Diss. ETH No. 19907

Data Stream Processing in Complex Applications

A dissertation submitted to the
ETH ZURICH

for the degree of
Doctor of Sciences

presented by
KYUMARS SHEYKH ESMAILI
Master of Science in Software Engineering, Sharif University of Technology
born 26 January 1981
citizen of Iran

accepted on the recommendation of
Prof. Dr. Donald Kossmann, examiner
Prof. Dr. Peter M. Fischer, co-examiner
Prof. Dr. Renée J. Miller, co-examiner
Prof. Dr. Bernhard Seeger, co-examiner

2011

Abstract

With a massive increase of available data sources, their high data rates and requirements for fast response, we have witnessed the emergence of data stream processing as a new data management paradigm. Many Stream Processing Engines (SPEs) have been built in recent years and several business applications have started to successfully use data stream management systems.

However, as industry gears toward using data stream processing in increasingly complex applications, existing approaches and systems are no longer sufficient. In this thesis, we identify, generalize and solve the challenges arising from such complex applications, based on a real-world use case, compliance monitoring in Service Oriented Architectures (SOA). In particular, we make the following contributions toward making data stream processing a viable solution for complex streaming applications:

1. *Establishing Schema for Data Streams and Exploring its Applications.* Static metadata (aka Schema) plays an important role in design and validation of data. We identify the most relevant static metadata aspects for data streams, and formally specify them in a model named Stream Schema. We also present a mechanism for validation of data streams against Stream Schema and analyze its runtime and space complexity. Experimental evaluations confirm the usefulness of semantic optimization techniques based on Stream Schema. Additionally, we show that applicability of Stream Schema extends beyond optimization gains. It broadens the static analysis of stream queries and changes the way that stream applications are modeled.

2. *Fine-grained Provenance Management on Data Streams.* Precise evidence of origin and processing of data (aka provenance) is a crucial part of many complex, data-centric applications. We investigate the requirements of fine-grained provenance on data streams using real-life use cases and determine the challenges, in particular caused by the volatility of data. To solve this problem, we analyze existing provenance computation methods, establish the formal foundations using an algebra, and show the details of the implementation. Numerous optimizations to reduce

the overhead of provenance computation and to enable on-demand computation of provenance make it possible to use fine-grained provenance in environments with high performance requirements. We show this by an extensive set of experiments.

3. *A Framework to Model Continuous Query Lifecycle and Query Modification Model.* Precise descriptions of lifecycle operations of a data stream query are necessary to provide exact semantics not only for "regular operations", but also for "boundary conditions", in particular when modifying a query to fit the new requirements. We establish a punctuation-based framework and a methodology that can formally express arbitrary lifecycle operations on continuous queries. It is based on input/output mapping functions and a limited set of punctuations such as Start and Stop control elements. Using this framework and methodology, we derive all possible variations of query modification, each offering a distinct combination of correctness guarantees. We also show how this framework and model can be integrated into state-of-the-art SPEs with fairly minor effort.

In order to avoid SPE-specificity, all solutions proposed in this thesis have been built upon abstractions and formalizations. Furthermore, all of them have been implemented and experimentally validated, in some cases, using more than one stream processing platforms

Zusammenfassung

Die massiv steigende Anzahl an Datenquellen, deren hohe Datenraten, sowie die Anforderung an schnelle Reaktionszeiten haben zur Etablierung der Datenstromverarbeitung (*Stream Processing*) als eigenständiges Datenverwaltungsparadigma geführt. Im Verlauf der letzten Jahre wurden zahlreiche Systeme zur Datenstromverarbeitung (*Stream Processing Engine*, SPE) entwickelt, welche bereits erfolgreich in vielen Bereichen der Industrie eingesetzt werden.

Im Zuge immer komplexerer Anwendungen sind existierende Ansätze und Systeme jedoch nicht länger adäquat. Diese Doktorarbeit identifiziert, generalisiert und löst die neu entstandenen Herausforderungen anhand eines Beispiels aus der Praxis, der Überwachung von Richtlinien (*Compliance Monitoring*) in service-orientierten Architekturen (SOA). Dabei werden insbesondere die folgenden Beiträge zur besseren Anwendbarkeit von SPEs in komplexen Anwendungen erbracht:

1. *Schema für Datenströme und seine Anwendungen.* Statische Metadaten (Schema) spielen eine zentrale Rolle in der Modellierung und Validerung von Daten. Die vorliegende Arbeit identifiziert die relevanten Aspekte statischer Metadaten bezüglich Datenströmen, und formalisiert diese in Form eines Modell namens *Stream Schema.* Für dieses Modell wird ein Validierungsalgorithmus beschrieben und bezüglich Speicher- und Zeitkomplexität untersucht, welcher die Korrektheit eines Datenstroms hinsichtlich eines benutzerdefinierten Schemas überprüft. Eine analytische und experimentelle Evaluierung zeigt den Nutzen solcher Schema bei der semantischen Optimierung von Datenstromanfragen. Weitere Anwendungsgebiete umfassen die statische Korrektheitsanalyse von Datenstromanfragen sowie neue Entwurfsansätze für Datenstromanwendungen.

2. *Feingranulare Provenance(Abstammung/Herkunft) auf Datenströmen.* Ein genauer Nachweis der Herkunft und Verarbeitungskette von Daten (*Provenance*) ist wichtiger Bestandteil vieler komplexer Anwendungen. Die vorliegende Arbeit untersucht die Anforderungen feingranularer Provenance im Kontext von Datenstromanwendun-

gen anhand konkreter Szenarien, und identifiziert die besonderen Herausforderungen dabei, insbesondere die Flüchtigkeit von Daten. Es werden existierende Ansätze zur Berechnung von Provenance betrachtet, die formalen Grundlagen von Stream Provenance auf Basis einer speziellen Algebra gelegt, sowie eine Implementierung von Stream Provenance inklusive zahlreicher Optimierungen beschrieben. Wie anhand zahlreicher Experimente nachgewiesen wird, ermöglichen diese Optimierungen den Einsatz feingranularer Provenance in Anwendungen mit hohen Leistungsanforderungen.

3. *Rahmenmodell zur Beschreibung von Lebenszyklus und Modifikation von Datenstromanfragen.* Präzise Modelle von Lebenszyklusoperationen einer Datenstromanfrage sind notwendig um exakte Semantik nicht nur im "regulären Betrieb" sondern auch in den "Übergangsbereichen" solcher Anfragen zu erhalten, insbesondere bei der Modifikation einer laufenden Anfrage. Diese Arbeit präsentiert ein Rahmenmodell auf Basis von Markierungen (*Punctuations*), mithilfe dessen beliebige Lebenszyklusoperationen formal ausgedrückt werden können. Dieses Rahmenmodell beruht auf Abbildungsfunktionen zwischen Eingaben und Ausgaben sowie einigen wenigen grundlegenden Markierungen wie Start und Stop. Das Rahmenmodell wird in der Folge für verschiedene Lebenszyklusoperationen angewendet, insbesondere für die Modifikation von laufenden Anfragen. Dabei werden alle Varianten der Anfragenmodifikation vollständig abgeleitet, welche jeweils bestimmte Korrektheitskriterien hinsichtlich des Eingangs- und Ausgangsstroms garantieren beziehungsweise nicht garantieren. Eine vollständige Implementierung zeigt, wie dieses Rahmenmodell mit geringem Aufwand in existierende SPEs integriert werden kann.

Um die beschriebenen Verfahren weitestgehend anwendbar zu machen, wurde für die gesamte Arbeit eine einheitliche Formalisierung entwickelt. Alle Verfahren bauen auf dieser Formalisierung auf, und wurden für diverse existierende SPEs – fallweise für mehrere SPEs – umgesetzt und evaluiert.

Acknowledgments

I would like to avail the opportunity to express my gratitude to several people who helped me accomplish the work presented in this dissertation.

First and foremost, I am in deep gratitude to Prof. Peter Fischer for his continuous support and guidance. His insightful comments and relentless dedication have deeply influenced this dissertation. He has also been a constant source of moral support and encouragement throughout my PhD studies. This dissertation would not have been the same without him.

My profound admiration also extends to Prof. Donald Kossmann. He made it possible for me to join Systems Group and provided me with invaluable advices and scientific intuition whenever I needed. I am also grateful for the work and availability of Prof. Nesime Tatbul. Her instructive comments on some of my works undoubtedly helped me with my achievements.

I am also indebted to my colleagues at Systems Group, in particular: to Boris Glavic, Tahmineh Sanamrad, Silvio Kohler, and Christian Tarnutzur for their conceptual and technical contributions; to Philipp Unterbrunner for reading an earlier version of this dissertation; and to Jens Teubner for his helpful comments on my PhD defense presentation.

Finally, I would like to thank my friends in Zurich: Akhi, Alex, Anne-Sophie, Araz, Arya, Boris, Emre, Farhad, Farzaneh, Giorgos, Hanieh, Jana, Masoud, Maysam, Morteza, Nemat, Nihal, Parivash, Peyman, Philipp, Pravin, Roozbeh, Sareh, Simon, Tahmineh, Tudor, Yunting, and Ziad. They made the years of my PhD studies a fun experience and an unforgettable time.

To the loving memory of my late cousins

Xolamsên

and

Şoriş

Contents

Chapter 1

Introduction

1.1 Background and Motivation

With the proliferation of dynamically generated data, we have witnessed the emergence of data stream processing as a new data management paradigm. Stream processing has proven to have a wide spectrum of practical applications with varying requirements such as real-time financial analysis, network traffic monitoring, sensor-based tracking. As a result, a significant number of academic (e.g., Borealis [15], STREAM [74], TelegraphCQ [31]) and commercial (e.g., StreamBase [11], Truviso [13], IBM InfoSphere Streams [8], MS StreamInsight [18]) stream processing engines (SPE[1]) have been built to meet these needs.

However, these systems and proposals only address the fundamentals of data stream processing. They fall short of meeting the requirements of complex streaming applications. To be precise, while analyzing our real-world use case, compliance monitoring in Service Oriented Architecture (SOA), we have identified three major shortcomings in the state-the-of-art of data stream processing[2]:

Use of Static Metadata: Metadata specifying structural and semantic constraints are invaluable in data management. They facilitate conceptual design and enable the checking of data consistency. They also play an important role in semantic query optimization, that is, optimization and processing strategies that are often highly effective, but only correct for data conforming to a given schema. While the use of metadata is a well-established technique in relational and XML databases, the same is not true for data streams. The ex-

[1]Throughout this thesis we use the terms SPE and DSMS (data stream management system) interchangeably.

[2]For usecase-specific problem statements, see Chapter 2.

isting work mostly focuses on the specification of dynamic information (e.g. constraints on arrival rates) have long been exploited for optimization [85]. However, beyond a few limited proposals (including K-Constraints [25] and Gigascope [36]), structural and semantic constraints on stream data have not been exploited in a systematic way.

Furthermore, due to the growing interest in developing declarative query languages for data stream processing, the need for schema knowledge, as its crucial complementary counterpart, arises.

Provenance Tracking: Tracking provenance, exploring which input data led to a given query result, has proven to be an important functionality in many domains such as scientific data management, workflow systems [39] and relational database systems [33]. Previous techniques have traditionally been classified according to their granularity: *Coarse-grained* provenance tracks dependencies between input and output data at a very abstract level (e.g., streams), whereas *fine-grained* provenance does so for individual data items in the input's data collections (e.g., tuples or attribute values).

Surprisingly, in the area of data stream management systems, there has been little work beyond *coarse-grained* provenance (e.g., tracking the sensor sources from which a data item originates [86, 68]). Recently, Huq et al [58] have proposed to achieve *fine-grained* stream provenance by augmenting *coarse-grained* provenance with timestamp-based data versioning, focusing specifically on query result reproducibility at reduced provenance metadata storage cost. Still there have not been attempts to identify the challenges involved in providing stream provenance support, and address these challenges in a scalable way.

Query Lifecyle Model: In contrast to one-time queries in relational databases, continuous queries in SPEs can run for unpredictably long time periods over infinitely long data streams. During their lifetime due to requirements of application semantics or resource constraints of the system these queries may go through different lifecycle states. Examples of such states are Stopped, Running, and Modification.

Previously, there have been few works on stopping and restarting of long-running queries in data warehouses [65, 30, 32]. Due to the inherent differences between streams and traditional warehouses and also their limited scope, these solutions are not sufficient to use in SPEs.

1.2 Contributions

In this thesis we make the following contributions:

- **Establishing Schema for Data Streams**: after reviewing various possibilities of describing data stream properties and exploring the general design space, we compile a relatively small but powerful list of stream description elements. At a more coarse-grained level, we proposed Stream Schema, a formal framework which not only provides precise definitions for individual constraints, but also offers a well-defined composition scheme, which makes it possible to describe data streams recursively and comprehensively. Furthermore, we have presented a mechanism for validation of data streams against Stream Schema and analyzed its complexity. On the practical side, we have shown how Stream Schema can be integrated into the existing stream processing models and sketched two implementation alternatives for Stream Schema Validator.

- **Exploring Applications of Stream Schema**: Once a schema for streams is established, we explore its applications. On top of validation and type annotation, we present three other areas that can benefit from Stream Schema:

  - Optimizations: Stream Schema can be exploited to pipeline the query execution and to partition the stream data. It also enables state reduction and query rewrites.
  - Static analysis of stream queries: using the Stream Schema knowledge, the set of runnable as well as non-executable expressions can be extended.
  - Modeling of streaming applications: since Stream Schema captures data consistency and structural constraints, it can greatly simplify the queries in streaming applications resulting in increased decoupling and reuse.

  These applications are experimentally demonstrated using two case studies.

  A paper reporting on the Stream Schema and its applications has been published [48] in the Proceedings of the 13th International Conference on Extending Database Technology (EDBT'10).

- **Building a Stream Provenance Management System and Exploring the Tradeoffs.**: We first investigate the requirements of fine-grained provenance on data streams using real-life use cases and determine the challenges, in particular caused by the volatility of data. Then, we analyze existing provenance computation methods, establish the formal foundations using an algebra, and show the details of the implementation. Numerous optimizations to reduce the overhead of provenance computation and to enable on-demand computation of provenance make it possible to use fine-grained provenance in environments with high performance requirements. We show this by an extensive set of experiments.

The general description of our solution has been published [52] in the Proceedings of the Database Systems for Business, Technology, and Web (BTW'11) Workshops. A detailed report including our algebra, experiments, and tradeoffs has been submitted to the VLDB Journal.

- **Establishing a Framework to Model the Lifecycle of Queries**: We establish a framework that can formally express arbitrary lifecycle operations on the basis of input-output mappings and basic control elements (such as query start or query stop) which are inserted into the input stream. We also devised a step-by-step methodology to model complex lifecycle operations using basic control elements.

  Finally, we propose a general architecture which allows integrating our framework into existing SPEs with minimal changes.

- **Modeling Query Modification**: we leverage our modeling framework to first define correctness criteria for query modification and then derive all possible variations of query modification. Each of these variations provides a different level of correctness guarantees and performance. The results of two sets of experiments to identify the key performance tradeoffs of the query modification variations are also reported.

  A paper describing our modeling framework and query modification model has been published [47] in the Proceedings of ACM SIGMOD Conference 2011.

1.3 Thesis Structure

The reminder of this thesis is organized as follows. We start by giving an overview of MASTER, our SOA monitoring use case, in Chapter 2. Chapter 3 introduces abstractions and formal definitions for the two fundamental concepts in data stream processing: data streams and stream queries (or, interchangeably, continuous queries). These definitions will be used in other chapters of this thesis.

In Chapter 4, we introduce Stream Schema, our proposal to capture and structure static stream metadata. A suite of optimizations enabled by Stream Schema as well as other applications of Stream Schema are presented in Chapter 5.

We explain our solution for the stream provenance management problem in Chapter 6. There, we also present Ariadne, our provenance-aware stream processing engine.

Our general framework to model query lifecyle is discussed in Chapter 7 and a detailed example of it is given in Chapter 8, where we leverage this framework to model query modification.

In Chapter 9, after presenting our ideas in previous chapters, we summarize the thesis. There, we also point out how our ideas have been integrated into the official deliverables of MASTER. At the end of this thesis, we outline ongoing work and discuss possibilities for future research.

Finally, Appednix A encompasses the complete query texts of the MXQuery implementation [29] of LR Benchmark [23]. These queries are (partly) referred to in Chapters 5 and 8.

Chapter 2

SOA Monitoring

In this chapter, we first (in Section 2.1) briefly introduce MASTER [9], an European Union FP7-integrated project aiming at building a complex streaming application for compliance monitoring. Then (in Section 2.2) we look at the conceptual side of MASTER in general and three challenges which have been addressed in this thesis in particular.

2.1 MASTER Project

A Service Oriented Architecture (SOA) [19] provides a common platform that allows integrating services and components across organizational domains, reusing them in different business settings, and building applications through composing services. Because it enables flexibility and agility, SOA has been quickly adopted by software vendors, service providers, and businesses.

Businesses which adopt SOA platform, as for other architectures, still need to comply with applicable laws and regulations (i.e. COBIT, ISO 27001). From this perspective, SOA's main characteristics such as abstract service interfaces, distributed ownership, and cross-domain operations introduce new challenges for the implementation of compliance controls and the assessment of their effectiveness.

Taking these aspects into the consideration, MASTER [9] (short for Managing Assurance, Security, and Trust in sERvices) aimed at solving problem of compliance monitoring in SOAs.

In following, we will give an overview of MASTER and its prototype implementation.

2.1.1 Big Picture

To solve the compliance problem in SOA, MASTER proposes [71] an architecture that extends SOA. This architecture is depicted in Figure 2.1.

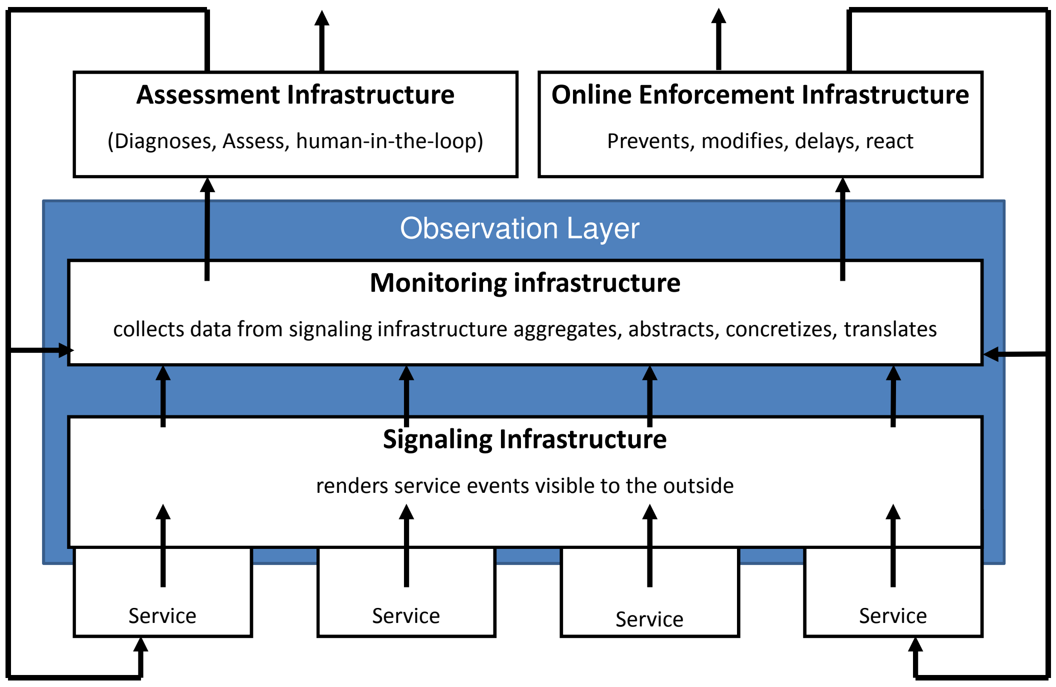

Assessment Infrastructure
(Diagnoses, Assess, human-in-the-loop)
Online Enforcement Infrastructure
Prevents, modifies, delays, react
Observation Layer
Monitoring infrastructure
collects data from signaling infrastructure aggregates, abstracts, concretizes, translates
Signaling Infrastructure
renders service events visible to the outside
Service
Service
Service
Service

Figure 2.1: *Compliance Architecture Proposed by MASTER [71]*

MASTER distinguishes between an observation layer that hooks into the SOA and the services to extract raw events and aggregates them to complex events, and an enforcement layer that provides analysis and reporting facilities and decision support on the events as well as automated enforcement where possible. The observation layer consists of two parts. First a signaling component that deals with events caused by services. Second an aggregation component that performs monitoring along with the generation of complex events.

The assessment and enforcement components receive input from the observation layer and feed back into the services. The difference between these two components is that enforcement emphasizes on automation of reaction and the adaptation of controls, while assessment focuses on reporting, knowledge management, and data warehousing with respect to the events.

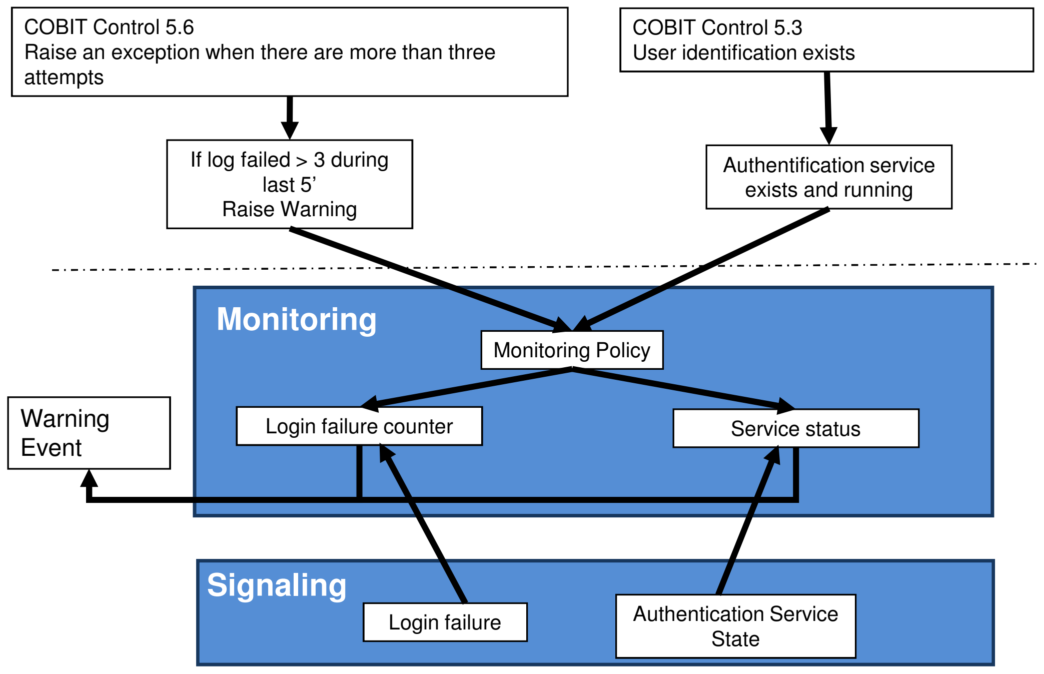

COBIT Control 5.6
Raise an exception when there are more than three attempts
COBIT Control 5.3
User identification exists
If log failed > 3 during last 5'
Raise Warning
Authentification service exists and running
Monitoring
Monitoring Policy
Warning Event
Login failure counter
Service status
Signaling
Login failure
Authentication Service State

Figure 2.2: *A MASTER Scenario: Credit Card Pin Code Validation*

2.1.2 Simplified Example

To further explain the MASTER architecture, we use a simplified compliance example. This example will be frequently referred to in the next section, where we point out a set of challenges in SOA monitoring.

Imagine that in a given SOA setup, there are multiple instance of a running ATM authentication service, which is realized through a certain number of message exchanges on the Enterprise Service Bus (ESB). Moreover, assume that there is a security control objective which states

if there are 3 failed login attempts within 5 minutes, block the card.

Figure 2.2 depicts this scenario. The signaling infrastructure collects the raw authentication events and forwards them to the monitoring layer. In turn, the monitoring layer performs filtering and aggregation operations to generate complex events. Once there are 3 failed-login attempts in 5 minutes, a complex event is produced which triggers a reaction in the enforcement layer that consequently blocks the ATM card.

2.1.3 Prototype Implementation

Our team in MASTER project was concerned with the observation layer (signaling and monitoring) problems. In parallel to tackling the conceptual issues, we also contributed to the prototype implementation of MASTER. We started by conducting a thorough analysis [63] of available implementations of Enterprise Service Buses (ESB) which form the backbone of any SOA. Based on this study, we chose Apache ServiceMix [1]. We then enriched ServiceMix with signaling capability which allows exposing internal dynamics of ServiceMix including messages exchanges and changes in the service registry. Details can be found in Deliverable D4.1.2 [3] of MASTER.

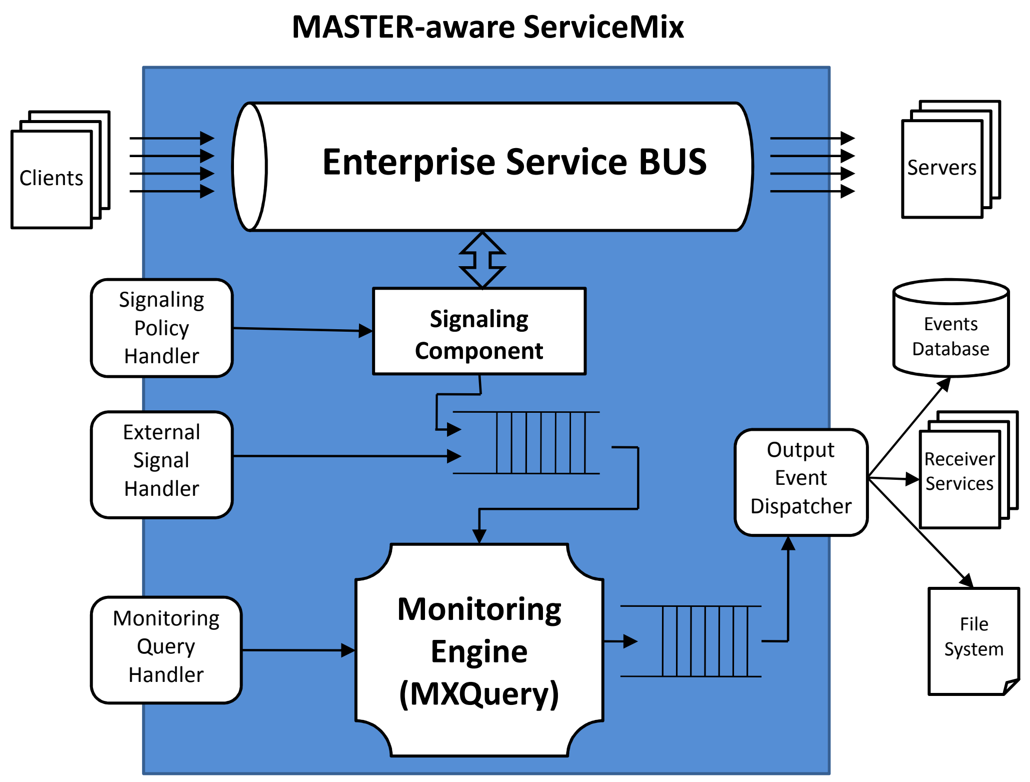


Figure 2.3: *Monitoring Infrastructure of MASTER*

On the monitoring side, as explained in Deliverable D4.1.3 [4], we embedded MX-Query [10] into ServiceMix to cater for complex event processing requirements.

Figure 2.3 depicts the overall view of our MASTER monitoring infrastructure. The topmost level of this architecture illustrates the core functionality of ServiceMix which is decoupling external clients and servers from each other bye exchanging messages on its bus. By adding the Signaling Component and through the Signaling Policy Handler interface, we have enabled SOA administrators to intercept and and extract messages on the bus. These signaling events are pushed into the input queue of the Monitoring Engine.

The Monitoring Engine, MXQuery, is the central element in this architecture. It processes multiple events with the objective of extracting complex, meaningful events. To this end, it employs techniques such as patterns, event hierarchies, and other relationships between events such as causality, membership, and timing. The Monitoring Query Handler interface allows interacting with the Monitoring Engine and managing the queries. This engine works on the event queue fed by the signaling layer, and puts complex events into an output queue. The Output Event Dispatcher interface allows dispatching these output events to a variety of sinks (receiver endpoints, event databases, or the local file system).

2.2 The Conceptual Side of MASTER

2.2.1 SOA Monitoring as a Streaming Application

The conceptual part of MASTER [71] describes SOA monitoring as *expressions over streams of events* and therefore an application of Stream Processing Engines (SPEs). Furthermore, it advises to use the XQuery programming language (with focus on its streaming feature [29]) to achieve this goal.

Returning to our ATM authentication example, an excerpt of the event stream reported by the signaling infrastructure would look like Listing 2.1.

```
<event type="login-failure" uid="511" time="10:00"/>
<event type="login-success" uid="101" time="10:01"/>
<event type="login-failure" uid="511" time="10:03"/>
<event type="login-failure" uid="511" time="10:04"/>
```

Listing 2.1: *A Stream of Login Events in XML Format*

Additionally, the aforementioned monitoring policy would look like Listing 2.2.

```
forseq  $w in $eventseq sliding  window
  start  curItem  $x  when $x/@type eq "login-failure"
  end    curItem  $y  when $y/@time - $x/@time gt 5 min
where   count ($w[@uid = $x/@uid]) > 2
return  <alert type="break-in attempt" account={$x/@uid}>
```

Listing 2.2: *Break-in Attempt Detection in XQuery*

Once the monitoring system detects a *break-in attempt*, it issues an alert which through enforcement mechanisms will result in blocking the card by the ATM machine.

2.2.2 Challenges

There are a number of conceptual challenges in tackling the SOA monitoring problem. Based on their generality, three of these challenges were selected [46] to be addressed in this thesis (for a list of other challenges, see Section 9.2). The first of these challenges is driven by a non-functional requirement while the other two arise from functional requirements. In the following, by the means of our ATM example, we will shortly explain each of these three challenges.

Efficient Processing of Event Streams

Monitoring subsystems often have strict performance constraints. In particular, information on compliance violations need to be reported within acceptable time frames, which can range from seconds (for human observation) to milliseconds (for automatic enforcement). These requirements need efficient processing of queries; not only to meet QoS requirements and minimize resource usage, but also to offer scalability in terms of queries and signaling sources. To cater for such performance-critical circumstances, optimization techniques play a key role.

Such optimizations generally exploit the existing implicit or explicit metadata about the data streams and the queries working on them. Although there has been a body of work on using the dynamic metadata to optimize streaming queries, the same does not hold for static metadata. The need to take advantage of static metadata becomes more apparent once one takes into account the highly-structured nature of MASTER data streams (interleaving sequences of relatively large XML messages with well-defined schemas).

For instance, in our ATM example, one can observe the following properties of the main input stream:

- the stream itself is a combination of some heterogeneous substreams (i.e *authentication events* stream, *cash withdrawal events* stream, *reporting events* stream, etc)

- the timestamp attribute of events generated by an ATM machine is strictly non-decreasing

- most of event streams are combinations of homogeneous substreams (since there are multiple instances of certain services running)

Hence, developing a framework for specifying stream constraints is perceived as a important step toward more efficient data stream processing. In Chapter 4, we define Stream Schema to capture such static constraints and properties. Later, in Chapter 5, we show the applicability of Stream Schema.

Tracing the Provenance of Complex Output Events

The provenance management feature in information systems allows tracing the data items exchanged between steps, including the original source data and the operations that were performed on the data. It provides the functionality to determine where data comes from and how it is transformed. In case of MASTER, provenance information is necessary for both in-depth analysis and evidence providing, which are required to prove compliance to regulations. More concretely, for a particular monitoring event, provenance information enables users to know:

- what signaling events have caused the event's creation (i.e. the actual three failed login attempts' details)
- what signaling sources have been involved in event's creation (the geographical location of the corresponding ATM machine)
- what monitoring queries were involved in generating the event (i.e. the query in Listing 2.2)

There is a sizable amount of work on data provenance management in relational databases, but due to the fundamental differences between relational and streaming systems, those techniques are not directly (if at all) applicable to data streams. Hence, *provenance management on data streams* remains, to a large extend, an unsolved issue. In Chapter 6, we explain our solution to solve this problem.

Dynamic Modification of Monitoring Queries

Monitoring policies of MASTER are embodied in continuous queries. Queries can be in the running state for possibly very long periods of time (in some cases, for years). However, throughout these periods, it may become necessary to modify them, due to, for instance:

- changes in laws, standards, or internal regulations (in the ATM example, relaxing the security control objective by increasing the maximum number of failed tries to 4)
- changes in high-level management interests (alerts should be accompanied by sufficient evidence)
- changes in the schema of the signaling events and streams (newly installed ATMs may provide richer metadata)

Here, the main challenge is how to perform the transition from an old version of the query to a newer version which reflects the changes. Such transitions, need to have a clear definition for semantics of query modification. After investigating this problem it becomes clear that we first need to have well-defined semantics for finer-grained lifecycle operations such as query stop and start. To this end, in Chapter 7, we have taken a bottom-up approach and proposed a formal framework and a methodology, which allow defining clear semantics for query lifecycle operations. Subsequently, in Chapter 8, we will leverage this framework to model query modification.

Chapter 3

Foundations of Data Stream Processing

Since no agreement on data models and processing models for data stream systems exists [60], in this thesis we have tried to assure generality of our solutions by basing them on abstractions and formalizations which are not pendent on any streaming systems. At the same time, we provide refinement and integration mechanisms to guarantee applicability of our ideas in state-of-the-art processing models.

In this chapter, after introducing a simple stream processing example in Section 3.1, we will provide clear definitions for data streams (in Section 3.2) and continuous queries (in Section 3.3). In the next chapters of this thesis, we will use these simple example and basic definitions and, if necessary, extend them accordingly.

3.1 Running Example

Our running example is $Q1$, which uses a tuple-based sliding window of size 3 and slide 2, applying a sum operation on the window (Figure 3.1).

3.2 Data Streams

A stream S is an unbounded sequence of *stream elements*, where each stream element has a *tuple* part and a *position*. The tuple part, similar to tuples or rows in relational databases, is set of attribute values which complies with a predefined schema. The position assures

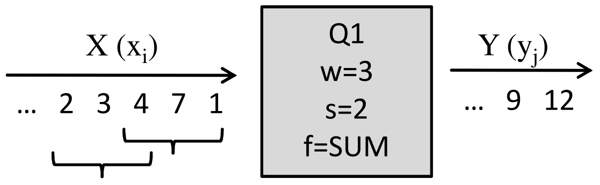


Figure 3.1: *Running Example - Q1*

the total order among the stream elements. In formal terms

$$E : \mathbb{N} \times T$$

in which T is the set of all (relational) tuples. As an example, the input stream in Figure 3.1 contains the following stream elements:

$$\mathrm{E}_1 = (1{,}1),\ \mathrm{E}_1 = (2{,}7),\ \mathrm{E}_1 = (3{,}4),\ \mathrm{E}_1 = (4{,}3),\ \mathrm{E}_1 = (5{,}2),...$$

Moreover, we define the following stream element utility functions and will use them in future definitions and proofs.

$$indexOf(E) : S \longmapsto \mathbb{N}$$
$$elementAt(i) : \mathbb{N} \longmapsto S$$

For instance

$$indexOf(E_3) = 3$$
$$elementAt(3) = E_3$$

In this thesis, unless it's necessary to have the full representation of stream elements, we use the tuple part of stream element to denote it.

3.3 Stream Queries

Stream queries (also known as continuous queries), in contrast to one-time queries in relation databases, are issued once but run (potentially) forever. A stream query takes a stream X as input and produces a stream Y as output. At any time point t, the answer to a continuous query Q is based on the elements of its input stream X seen up to t, and this answer is updated as new stream elements continue to arrive on X, following the monotonicity definition in [66]. In our abstraction, each continuous query Q is defined by a unique *query identifier* and a *query expression.* As a convention, we use the notation of x_i to indicate input stream elements and y_j to indicate output stream elements, where x and y correspond to the tuple parts, and i and j correspond to the positions ($i, j \in \mathbb{N}$), respectively.

In following we first present how we have abstracted away the logic of queries into a pair of mapping functions. Then after describing operator-level mapping functions we show how they can be composed to build mapping functions of complete queries.

3.3.1 Query Mapping Functions

We use the notion of *mapping functions* as an abstraction of the details of the query expression. Mapping functions are defined on of a pair of streams (input and output stream) and establish a relationship between a single element in one stream to a set of elements in the other. We specify two mapping functions, which capture the data dependencies established by the query expression of a given continuous query Q:

- *depends(y_j)*: $E \longmapsto \{E\}$[1] Given an output data element y_j, returns the sequence of all input data elements that y_j depends on.

  $$depends(y_j) = \{x_i | x_i \in X \text{ where } Q(X) = y_j\}$$

  Notice that the *depends(y_j)*if fact defines the Why-Provenance (see Chapter 6).

- *contributes(x_i)*, $E \longmapsto \{E\}$: Given an input data element x_i, returns the set of all output data elements which x_i has contributed to.

  $$contributes(x_i) = \{y_j | x_i \in depends(y_j)\}$$

We illustrate the query mapping functions in Figure 3.2, on query $Q1$ of our running example. Note that the mapping functions are not only driven by the query expression, but also the *starting position*, which is their reference point in the streams. For example, in Figure 3.1, a mapping starting at the second input element instead of the first one would generate a different output stream (12 instead of {14,11}).

[1]To keep our definitions simple, we use the set notation to represent a sequence of elements. In these "sets", the positional information of stream elements defines the order.

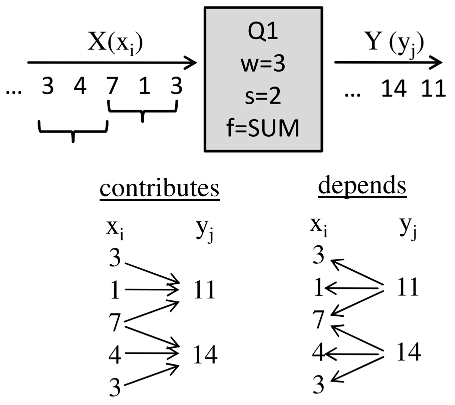


Figure 3.2: *Mapping Functions of Q1*

3.3.2 Query as Composition of Operators

Since operators provide the building blocks for complete queries, we now decrease our abstraction level to that of operators, analyze each operator and then compose them in order to achieve mapping functions for complete queries.

Conceptually, mapping functions apply to operators in the same way as they apply to entire queries, defining on which input data item an output items depends and vice versa. We can thus complement our definition of mapping for an operators OP.

$$\mathit{dependsOP}(y_j) = \{x_i | x_i \in X \text{ where } OP(X) = y_j\}$$

$$\mathit{contributesOP}(x_i) = \{y_j | x_i \in \mathit{dependsOP}(y_j)\}$$

Mapping functions of individual operators can be derived from their formal definitions. Moreover, operator mapping functions can express the distinction between stateless and statefull operators:

- *Stateless* operators (e.g., selection, projection) perform their computation on one tuple at a time. More formally, for a stateless operator each output stream element y_j depends on exactly one tuple:

$$\forall y_j \in Y, \ |\mathit{depends}(y_j)| = 1$$

- *Stateful* operators (e.g., window-based operators, pattern matching) perform their computation possibly on multiple tuples at a time. More formally, for a stateful operator some (if not all) output stream elements y_j depend on more than one tuple:

  $$\exists y_j \ \in \ Y, \ |\mathit{depends(y_j)}| > 1$$

  As we will see further in this thesis, in contrast to the relative simplicity of stateless operators, handling statefull operators requires special attention.

Queries are composed of these operators, forming a query plan. In this composition, mapping functions are transitive, since the output items of one operator form the input items for the next.

In other words, given a query Q with $dependsQ()$ and operators OP_k with $dependsOP_k()$, if

$$Q = OP1_1||OP_2||...OP_K$$

then

$$dependsQ(yj) = dependsOP_1(dependsOP_2(...dependsOP_K(yj))))$$

Using induction and exploiting the transitivity property of *depends function* this above composition rule can be proved . A similar statement holds for composition of *contributes functions*.

Given this method to derive the mapping functions from formal operator specification and the composition rules for mapping functions, we now can determine the overall mapping functions of sequential query plans. More complex operators and query plans such as trees or DAGs follow the same approach, but need extensions on the definition of the mapping functions and the composition.

Chapter 4

Stream Schema: Static Metadata for Data Streams

4.1 Introduction

4.1.1 Motivation

Metadata specifying structural and semantic constraints, is invaluable in data management. It facilitates conceptual design, and enables checking of data consistency. Metadata also plays an important role in semantic query optimization; that is, optimization and processing strategies that are often highly effective, but only correct for data conforming to a given schema. While the use of metadata is well-established in relational and XML databases, the same is not true for data streams. The existing work mostly focuses on the specification of dynamic information. Constraints on arrival rates for instance, have long been exploited for optimization [85]. However, beyond a few limited proposals (including K-Constraints [25] and Gigascope [36]), structural and semantic constraints on stream data have not been exploited in a systematic way.

The need for capturing and structuring static stream metadata extends beyond semantic query optimization. In fact, since the schema knowledge is a crucial complementary counterpart for declarative query languages, the growing interest in developing declarative query languages for data stream processing [31, 12, 29, 43] has also highlighted the need for developing schema for data streams.

In this chapter, we present *Stream Schema*, our proposal for modeling structural and semantic constraints on data streams. The following chapter is then is dedicated to applications of Stream Schema.

4.1.2 Contributions

The main contributions of this chapter are:

- exploring the possibilities of describing stream constraints and identifying an expressive set of the common constraints;
- formalizing these constraints and their combination into a framework named Stream Schema;
- presenting mechanisms for correctness checking and validation of Stream Schema along with complexity analysis and a discussion on the practical aspects;
- explaining how Stream Schema can be integrated into a wide range of stream processing models.

4.1.3 Outline

The rest of this chapter is structured as follows. In Section 4.2 we present Stream Schema. This presentation is followed by an explanation of how data streams can be validated against Stream Schema in Section 4.3. In Section 4.4, we show how Stream Schema can be exploited in a number existing stream processing models. After an overview of related work in Section 4.5, we conclude the chapter in Section 4.6.

4.2 Stream Schema

In this section, we propose an equivalent of database schema in the Relational Model for data streams. This proposal is named *Stream Schema*. For generality reasons, in designing Stream Schema we make very few assumptions. Our formal stream model uses a totally ordered sequence of items and does not assume any specific value ordering (e.g., a timestamp attribute).

Stream Schema, at a very high-level, is consists of a small number of Stream Schema elements, each capturing one type of stream semantic constraints, and a recursive combination scheme which states how the individual Stream Schema elements can be composed. As explained in Section 4.2.1, the selection of the Stream Schema elements has been based on an exploration of the design space for describing sequences of items. The actual Stream Schema elements are presented in Section 4.2.3 using a running example introduced in Section 4.2.2. Finally, the combination scheme is presented in Section 4.2.4.

4.2.1 Design Space

A data stream is a *sequence* of *items*. Each of these two aspects can be a rich source of static metadata and therefore need to be considered in designing schema for data streams. Since item-level schema is a relatively well-established concept in data management systems, in Stream Schema we adopt the state-of-the-art item schema (and make it our first Stream Schema element) and instead focus on the more coarse-grained stream-level aspect of data streams which has not be sufficiently investigated so far.

There are two main classes of formalism available to describe the stream-level aspect: a) *Inter-item relationships*; and b) *Sequence decomposition*. Below, we explain each class in more details.

Inter-item relationships are captured in several ways:

1. *value (order) relationships between items*, such as a total order over the domain of certain attributes [36];
2. *temporal logic*, such as LTL, to express that particular properties will hold always, will hold at some point, or will hold until some point; this is often used in specification and validation of complex systems [72]
3. *grammars*, that describe permitted sequences of items (e.g., the child nodes in an XML document). To deal with infinite sequences, ω-grammars [82] have been defined.

These three approaches share a certain amount of overlap (and in certain cases equivalence, e.g., LTL and ω-regular grammars [82]).

Sequence decomposition, in turn, has not received a great deal of attention. Generally speaking, a data stream can be decomposed along two dimensions:

1. *item values*, or in other words, logical partitioning;
2. *item ranges*, or in other words, reoccurring patterns.

It should be noted that in nearly all sequence or stream-oriented query processing languages, operators with the corresponding semantics are available, since *item value* decomposition has a correspondence to *group by*, whereas item range decomposition corresponds to *window* or *pattern* operators.

For the reasons of validation complexity, we excluded general integrity constraints: uniqueness or functional dependencies among element values in a single stream may require possibly extremely large amounts of state, as all different occurrences may have to be recorded. Foreign key relationships are only possible among multiple streams, which we have postponed for future work, as it involves model-specific join algorithms.

In summary, Stream Schema includes the following elements (each corresponds to one type of constraints on streams):

- Item-level schema
- Logical Partitioning
- Pattern specification
- Inter-item relationships

The detailed and formalized definition of these elements is summarized in Table 4.1 and explained in Section 4.2.3. We give examples of each element using the Linear Road Benchmark just to keep the examples simple. However, we stress that our model is much more general than Linear Road as illustrated in Sections 4.4, 5.5 and 5.6.

4.2.2 Running Example: Linear Road Benchmark

To illustrate our approach, consider Linear Road which is a popular benchmark for data stream management systems [23]. Linear Road specifies a schema for its benchmark, albeit in an informal way as no stream schema models were available when the benchmark was created. Linear Road describes a traffic management scenario in which the toll for a road system is computed based on the utilization of those roads and the presence of accidents.

Both toll and accident information are reported to cars; an accident is only reported to cars which are potentially affected by the accident. Furthermore, the benchmark involves a stream of historic queries on account balances and total expenditures per day. The input data stream for Linear Road is constrained to contain only four types of tuples: position reports and three different types of historical query requests. Moreover, position reports are associated to a specific vehicle and the reports for each vehicle are constrained to follow a specific pattern called *vehicle trip.*

4.2.3 Individual Elements of Stream Schema

There are four types of Stream Schema Elements. Below, we explain these elements one-by-one. For each element we introduce it, present its formal definition, give an example from the LR benchmark, and finally discuss its potential uses.

Const.		Formal Definition	Example from LR
Item Schema		$I\!S : (N, \mathcal{A})$ $\mathcal{A} : \{A_i\}$ $A_i(I\!S) = V_i$ $A_i(I) = v \in V_i \cup \mathit{NULL}$	For Item Schema of position report stream (P): N: P_{IS} $\mathcal{A}$: {TIME,VID,SPD,XWay,LANE,DIR,SEG,POS}
Partitioning	By Str.	$S \xrightarrow{\mathcal{A}'} S_1, S_2$ $\forall I \in S \begin{cases} I \in S_1 & \text{if } \forall A_i \in \mathcal{A}', A_i(I) \neq \mathit{NULL} \\ I \in S_2 & \text{otherwise} \end{cases}$	The input stream (S) along : $\mathcal{A}'$: {TIME,VID,SPD,XWay,LANE,DIR,SEG,POS} S$_1$: position report stream (better known as P) S$_2$: rest (mixture of three query streams)
	By Val.	$S \xrightarrow{A_p,\ n} S_1, S_2, ..., S_n$ $\forall I \in S, I \in S_i$ if $A_p(I) = v_i$ where V_p=$\{v_1, ..., v_n\}$ is the finite domain of A_p	Position report stream (P) along: A_p: VID and $n = \lvert\text{VID}\rvert$ S$_i$: the position report stream of the vehicle with VID(I) = vid$_i$
Pattern/ Repetition		$P ::= F \mid F'^{**'}$ $F ::= FF \mid F'\lvert'F \mid F'^{*'} \mid F'^{+'} \mid '('F')' \mid E \mid \epsilon$ and element E is defined by restricting the Item Schema E: $I\!S_{(V_i \leftarrow \{v_i\})}$	Vehicle trip pattern for a particular vehicle (vid) $(L_0(L_1\mid L_2\mid L_3)^*L_4)^{**}$ Where L$_j$ is defined by restricting $\text{P}_{I\!S}$ $\text{P}_{I\!S_{(\text{VID}\leftarrow\{\text{vid}\},\text{LANE}\leftarrow\{\text{j}\})}}$
Next Const.		$c(A_n(\mathit{current}), A_n(\mathit{next}))$ where *current* and *next* are any two adjacent items: $\mathit{current} = I_i \leftrightarrows \mathit{next} = I_{i+1}$ and comparison function c is defined over domain of A_n	On the main stream: $\leq$ (TIME(*current*), TIME(*next*))
Disord.		$\forall i, j \in \mathcal{N}$: $< (i + k, j) \Rightarrow \quad < (A_o(I_i), A_o(I_j))$ where stream is in *ascending* order on accessor A_o and k is the upper bound of disorderedness	On the main stream: A_o: TIME $k = 0$
Combination		The combination is modeled by a *fixed-point combinator*: $F(\chi) = (\mathcal{C} \times \mathcal{P})^* \times (\mathcal{T}_{NFA} \cup (\mathcal{P}^* \times \mathbb{N} \times \chi)^*)$ It is tree-like recursive structure in which stream-level *next-constraints* can be placed at **any** node, pattern-level *next-constraints* at **leaf** nodes, and *disorder* specification only at **root** node.	Combination of all important constraints on input stream of LR is depicted in Figure 4.1.

Table 4.1: *Formal Definition of Stream Schema Constraints*

Item Schema

A data stream is composed of items (also called tuples or events). Items can be specified in any data model, e.g., relational [16, 24, 22, 36, 40, 2] , XML [29], object-based models [7]. Given this heterogeneity, Stream Schema is designed to work with any item model that supports access functions, denoted by A, (be they relational attributes, elements, XPath expressions, methods, etc.) For the sake of simplicity, we use the term 'attribute' instead of 'access function' in rest of this chapter.

We will use IS to refer to an item schema which will have a name N and a set of attributes $\mathcal{A}$. Assuming that I refers to an item, we say that $I \models IS$ if I conforms to the schema IS. We will use V (or sometimes V_i) to refer to the domain of an attribute. The value *NULL* is a special value where

$$\forall V, \quad NULL \notin V$$

Domains may be infinite or finite. Additionally, a set of *comparison functions* C is defined over each domain. In formal terms

$$C : V \times V \rightarrow \mathbb{B} \cup \{NULL\}$$

in which $\mathbb{B}$ is the boolean domain.

As an example from Linear Road, the position reports have a relational item schema which name it P_{IS}. It has the following set of attributes: time (TIME), vehicle identifier (VID), and speed (SPD), expressway (XWay), lane (LANE), direction (DIR), segment (SEG), and position (POS). An instance of P_{IS}, a position report item p, is shown below

$$\mathrm{p(123,23,50,1,3,0,23,4)} \models \mathrm{P}_{IS}\mathrm{(TIME,VID,SPD,XWay,LANE,DIR,SEG,POS)}$$

A simple application of item schemas is to ensure item structure and domain integrity, e.g. checking that all attributes are present, and the observed values are in the required domain. Item schemas also play an important role in optimizing the storage of items and in optimizing predicate evaluation on item values. We refer the reader to the existing literature on how to do this [34, 54, 49, 79].

Logical Partitioning

Two logical partitioning constraints are considered in Stream Schema: 1) partitioning by item structure; and 2) partitioning by attribute value.

Formally, partitioning by item structure splits the original stream S into two substreams S_1 and S_2, using a set of attributes $\mathcal{A}'$. Items on which all attributes of $\mathcal{A}'$ are defined (i.e., non-NULL) go into S_1, all other items go into S_2.

As an example of partitioning by item structure, Linear Road's input stream is a combination of four different streams (one containing only position reports, and the other three query streams). To support SPEs that can only handle streams with a single item schema, Linear Road defines a schema with the union of all attributes of all the different substream, 14 in total. Attributes not needed for a particular type are given the value NULL. For example, for the position report stream the attributes Type, TIME, VID, SPD, XWay, LANE, DIR, SEQ, POS are non-NULL. Each of these substreams has its own item schema, and possibly also other constraints, e.g., pattern or next-constraints (see below). In contrast, In SPEs that support heterogeneous item schemas, this could have been expressed explicitly using multiple item schemas.

Partitioning by attribute value is defined using an attribute A_p (with finite domain V_p) and a partitioning bound $\boldsymbol{n}$. This constraint creates at most $\boldsymbol{n}$ substreams with the same schema, one for each value of the A_p attributes. In general case, instead of A_p, we can have a set of attributes $\mathcal{A} = (A_1, ..., A_k)$ with domain $V_1 \times V_2 ... \times V_k$.

As an example, the position reports stream can be partitioned based on the VID value ($A_p = VID$). If there are at most $|VID|$ different vehicles, then $|VID|$ substreams are created.

Multiple alternative partitionings of the same stream are often possible. For example, one could split the position report not only by vehicle id, but also by highway (XWAY) and direction (DIR).

Partitioning is useful for both optimization purposes (to partition data and query plans) and for structuring. The overall stream might not be suitable to express additional constraints, but the partitioned stream might be.

Patterns

In many cases, a stream (or a logical partition of a stream) can be broken into item ranges (also called subsequences), which in turn can be described as a well-defined sequence of items, e.g., a web session log could be expressed as *login browse** *logout* in common regular expression language. The name *pattern* has been established in the literature for such structures. For Stream Schema, patterns are of finite length. They can repeat infinitely often, but the instances may not overlap. Each of these repetitions therefore defines a subsequence, and their repetition defines a possibly infinite stream.

We use a regular language F for patterns, and designate the repetitions as P. The regular language is defined over items that may satisfy an item schema or restrictions (selection conditions) on an item schema. Compared to languages in pattern queries (e.g., SASE [17] or Cayuga [40]), this language has two simplifications that stem from our need for a description language and not a query language: 1) Only contiguous patterns can be specified, to ensure that the whole stream is described and no items can be ignored 2) All language constructs related to *matches* (e.g., length, next start, result generation) are not needed, since we do not allow overlapping matches, and want to describe the whole stream.

The grammar of the our pattern specification language is given in Table 4.1. It should be noted that the $**$ repetition indicator represents the possible infinity of a stream. The definition of the *variables* in patterns is done by placing predicates on types and/or values of an item. As an example, in the Linear Road benchmark, there are some patterns in the input streams, including the following taken from the specification [23].

> A set of vehicles, each of which completes at least one *vehicle trip*: a journey that begins at an entry ramp on some segment and finishes at an exit ramp on some segment on the same expressway.

The vehicle trip pattern (of Linear Road) for a specific vehicle (with VID = vid) is described as follows

$$(\mathrm{L}_0(\mathrm{L}_1|\ \mathrm{L}_2|\mathrm{L}_3)^*\mathrm{L}_4)^{**}$$

Where L_j is defined by restricting $\mathrm{P}_{\mathcal{IS}}$ (the position report item schema defined above)

$$\mathrm{P}_{\mathcal{IS}_{(\text{VID}\leftarrow\{\text{vid}\},\text{LANE}\leftarrow\{\text{j}\})}}$$

The constraints in a pattern can be used to optimize pattern queries and semantic windows. As one example, pattern specifications in the query can be simplified by using knowledge of existing patterns in the data. In addition, pattern information can be used to *unblock* operators, i.e., permitting a blocking operator on infinite data to produce results (see Section 5.3).

Inter-Item Relationship

In many streams, attribute values in different items have a well-defined relationship. For example, we may be able to define an ordering on attribute values. In certain streaming models, an ordering is hard-coded for timestamps, but ordering constraints among other attributes cannot be specified. In Stream Schema, we generalize these orderings between attribute values and capture them using *next-constraints*. A *next-constraint* is defined based on a comparison function $\boldsymbol{c}$ over an attribute A_n of two adjacent items called *current* and *next*.

The scope of validity for such a constraint may be the whole stream, a logical substream, or a single repetition of a pattern. For the sake of simplicity, we illustrate the pair of adjacent items in latter with *pnext* and *pcurrent*.

For example, the value of the TIME attribute in Linear Road's input stream is always non-decreasing, and this can be expressed with a next-constraint on the whole stream as follows:

$$\leq (\text{TIME}(\textit{current}), \text{TIME}(\textit{next}))$$

In addition, in Linear Road, the position of a vehicle is non-decreasing or non-increasing (depending) while it stays on the same highway and direction. Such a constraint can again be specified by a *next-constraint*, but one whose scope is limited to a single pattern repetition (the pattern of a single vehicle on a highway).

Such ordering constraints are useful for query optimization, for example, to unblock operators (similar to punctuations [84]), rewrite semantic windows or patterns, and also for semantic correctness checks to determine if semantic windows close.

Disorder

Even though data stream models assume a total (or at least partial) order in the data stream, real-life data streams often do not conform to this assumption. For example, due to the impact of network delays or the lack of strict time synchronization between different sources, items may arrive out of order.

In Stream Schema, similar to Ordered-Arrival Constraint [25], we use a parameter $\boldsymbol{K}$ to express bounded disorder. This static description is an upper bound, in practice dynamic statistics may provide a more precise bound. In the Linear Road case there is no disorder given, but it could be easily envisioned that car position reports from different segments might be delayed, thus creating disorder in the stream.

Knowing a bound on the amount of disorder has been traditionally used to determine the size of a buffer required to restore the order, but more recent work takes disorder more into the account for specific operators and provides related optimizations [70, 67].

4.2.4 Combination Scheme

Having explained the individual elements of Stream Schema, we now need to determine which combination provides a good balance between expressiveness and complexity.

We allow the nesting of logical partitioning (item value decomposition) without limitation, since this is simple to validate even in the presence of other constraints. We also permit inter-item relationships (i.e. next-constraints) at any level of nesting. For patterns

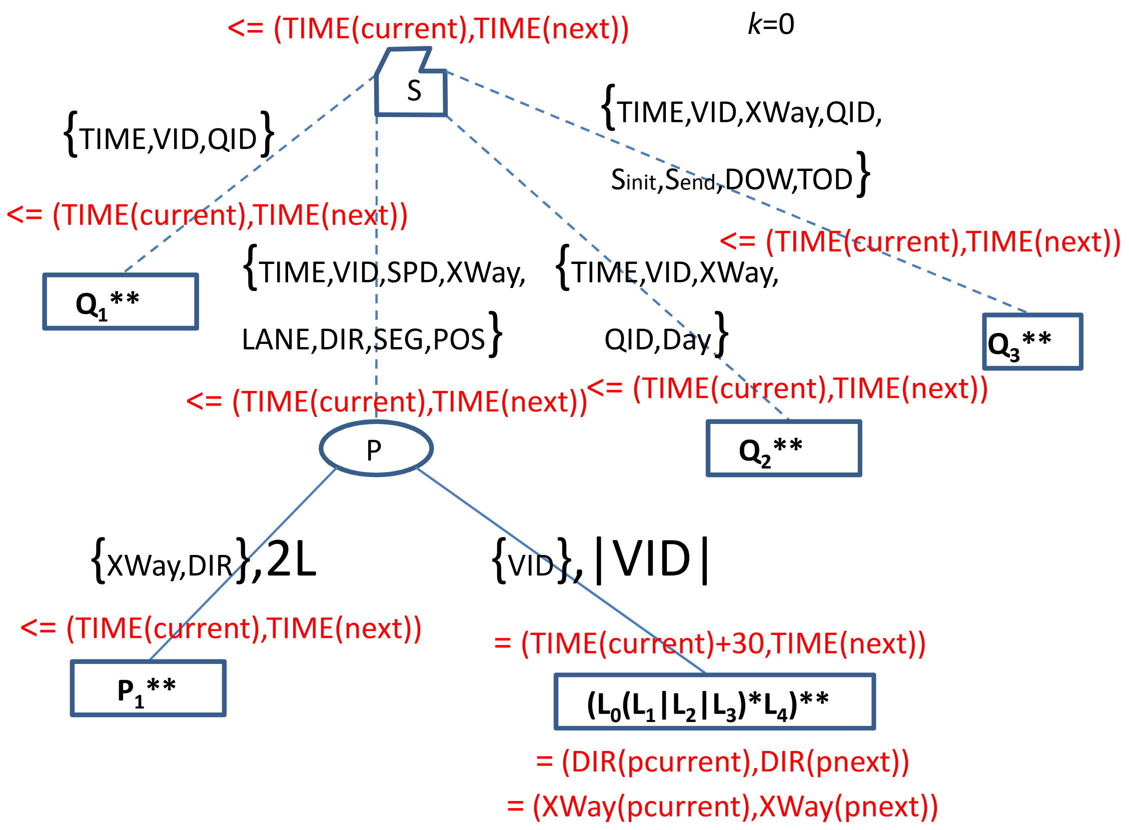

<= (TIME(current),TIME(next))
k=0
S
{TIME,VID,QID}
{TIME,VID,XWay,QID, Sinit,Send,DOW,TOD}
<= (TIME(current),TIME(next))
<= (TIME(current),TIME(next))
Q1**
{TIME,VID,SPD,XWay, LANE,DIR,SEG,POS}
{TIME,VID,XWay, QID,Day}
Q3**
<= (TIME(current),TIME(next))
<= (TIME(current),TIME(next))
P
Q2**
{XWay,DIR},2L
{VID},|VID|
<= (TIME(current),TIME(next))
= (TIME(current)+30,TIME(next))
P1**
(L0(L1|L2|L3)*L4)**
= (DIR(pcurrent),DIR(pnext))
= (XWay(pcurrent),XWay(pnext))

Figure 4.1: *Linear Road Stream Schema*

(item range decomposition), we are more restrictive: they may not overlap, and they are only allowed as terminals in the nesting of the logical constraints. Finally, we allow a single *disorder constraint* at the root node. This design avoids the complexity of having to combine the substreams in validation, reducing the verification effort. As we will show in Section 5.2, this restricted form, nonetheless, allows important optimizations for queries containing patterns and semantic windows.

This combination scheme results in a tree-like structure which we formally model using a *fixed-point combinator*[1] (see Table 4.1) . In this tree, every node represents a stream (or a substream) and every edge represents a *partitioning* constraint. The parent node of this edge is the stream which the partitioning constraint holds on and the child stream is result of the partitioning. In case of partition *by structure*, this is the substream with non-NULL values, and in case of partition *by value*, this substream is the representative

[1]A fixed-point combinator (http://en.wikipedia.org/wiki/Fixed_point_combinator) is a higher-order function that computes a fixed point of other functions

of the whole class of partitions.

It is noteworthy that since partitioning may place adjacent pairs of items into different partitions, the next-constraints will not necessarily hold over the partitioning (in other words, it is not inherited by substreams), thus new next-constraints may be needed to express the relation after the partitioning.

As an example, Figure 4.1 visualizes the constraint tree on Linear Road data stream. Reading the figure top-down, there is a next-constraint for non-decreasing time and a zero disorder constraint on the complete stream S. This stream can be partitioned into four different substreams (P,$Q1$,$Q2$,$Q3$) one position reports stream and three query streams, based on the existence of a set of attribute values. The position report stream P can further be partitioned by either VID values, or by XWay and DIR. The root stream has a non-decreasing TIME attribute which also holds for many of its substreams. After partitioning along the VID attribute, the resulted stream has a more precise next-constraints (stating the fact that a particular vehicle, emits its position report every 30 seconds)

After all these partitionings, the leaves of the tree contain item and pattern descriptions. In the case of the vehicle trip pattern L, there is a next-constraint that only holds within an instance of the pattern: during a trip, a vehicle will stay on the same highway and direction, segments and positions will either be non-decreasing or non-increasing, depending on the direction.

The Stream Schema specification provides a high level of expressiveness and flexibility, but also minimizes the potential for conflicting definition and the cost needed for validating.

4.3 Validation of Stream Schema

In this section, we first explain how the correctness of individual Stream Schema elements should be checked. Then validation is formally described using *prefix validation*. We also give an analysis of the space and time complexity. An overview of practical aspects concludes this section.

4.3.1 Checking Correctness for Individual Schema Elements

Item Schema

Given an item schema IS, an item I in a (sub)stream satisfies the item schema constraint IS if for all attributes A_i of the item schema, $A_i(I) \in V_i$, in particular $A_i(I) \neq NULL$.

Partitionability

A (sub)stream satisfies a partitioning constraint if the constraint assigns every item to a partition (so there are no items omitted). Hence, a partitioning by structure constraint is, by definition, valid since every item is assigned to either S_1 or S_2. A partitioning by value constraint is valid if the size of the domain of A_p has a finite upper band and if for all I in the (sub)stream on which the constraint is applied, $A_p(I) \neq NULL$.

Pattern and Repetitions

A stream satisfies a pattern constraint if an automaton representing the pattern accepts the item stream (prefix). Such an automaton can be created from the pattern specification by translating the finite part of the pattern (represented by F in Table 4.1) into a finite state machine.

For the repetitions (F^{**}), additional edges are added from accepting states to states reachable from the starting state, marked with the pattern starting symbols.

It should be noted that the pattern validation formalism closely corresponds to Büchi-automata [82], the default implementation of ω-grammars over infinite sequences: It handles infinite iterations over all well-defined set of accepting states (i.e. the representation of F). The difference is in the interpretation of correctness: we detect incorrectness also over finite sequences, while a Büchi-automaton only works with infinite sequences.

Next-Constraints

A stream satisfies a next constraint if for every consecutive pair of items I_1, I_2, the comparison $c(A_n(I_1), A_n(I_2))$ is evaluated *true*. A next-constraint specified within a pattern F only needs to hold for items within the same pattern instance, and a next-constraint specified with a partition only needs to hold in that partition.

Disorderedness

Stream Schema does not require an ordering relation. However, one may be specified by a next-constraint. In the presence of a total order specified over a particular attribute (e.g., a timestamp attribute), a disorder constraint of k may also be specified. A stream is valid (satisfies the disorder constraint) if an ordered sequence can be generated by sorting the items inside a sliding window of size k.

Combination

Constraints can be combined recursively to form a tree of constraints as illustrated in Figure 4.1. Constraint checking can be done recursively by checking constraints bottom-up through the tree. In fact the combination is correct as long as all of its building blocks are correct.

4.3.2 Prefix Validation

Since any newly arriving item could violate a given schema, complete validation of an infinite stream is not possible. Therefore stream validation is based on validating the current item using the prefix validation result and prefix validation state. Since disorder is orthogonal to all aspects of validation, we first define the validation algorithm for stream data without a disorder constraint, and then extend the definition and analysis for disordered streams.

In order not to store the complete prefix, we define a special data structure that captures only the information necessary for validation. This data structure is also recursive, and mirrors the structure of a stream closely.

- Partitioning: we define a recursive data structure containing the nested stream data; in addition we need to store the information on the partition decision. The number of substreams can be derived from the number of (nested) states.

- Pattern: the automaton state for a single repetition of a pattern, e.g., all active states in an Nondeterministic Finite Automaton (NFA).

- Next-constraints: for all attributes of the constraint, we store the previous value. For pattern-repeating next constraints, values are reset at the end of a pattern instance.

Using the prefix validation result, the prefix validation state and Stream Schema, the arrival of a new item produces a new validation result and state. The validation is performed by checking the item schema, the next constraints, and then either checking the pattern (leaves) or the partition constraint and then recursively the nested substream definition(s).

4.3.3 Validation Mechanism

Recalling the formal definition from Table 4.1, a Stream Schema instance M has a name and a definition:

$$\mathcal{M} = \mathcal{N} \times \mathcal{D}ef$$

In which $\mathcal{D}ef$ is modeled using a fixed-point combinator:

$$F(\chi) = (\mathcal{C} \times \mathcal{P})^* \times (\mathcal{T}_{NFA} \cup (\mathcal{P}^* \times \mathbb{N} \times \chi)^*)$$

Here, F defines a set of next-constraints ($\mathcal{C}$), and either a pattern definition ($\mathcal{T}$) or a partitioning and a nested stream definition ($(\mathcal{P}^* \times \mathbb{N} \times \chi)^*$). For notational simplicity, we use $A^* = \bigcup_{n \in \mathbb{N}} A^n$.

Using F, Def can now be defined as the smallest set so that

$$Def = F(Def)$$

in an explicit form

$$Def = \bigcup_{n \in \mathbb{N}} F^n(\emptyset)$$

Moreover, to do the validation, we need to establish the contents and the structure of the runtime state. It mirrors the schema definitions, since each definition has a particular amount of state, and all partitions require nested state.

$$G(\eta) = \mathcal{V}^* \times (\mathcal{S}_{NFA} \cup (\mathbb{N}^* \times \eta)^*)$$

and $\mathcal{ST}$ as the smallest set so that

$$\mathcal{ST} = G(\mathcal{ST})$$

.

The validation function for a an item and previous state can now be defined as:

$$sval : \mathbb{B} \times \mathcal{I} \times \mathcal{ST} \times \mathcal{DEF} \to \mathbb{B} \times \mathcal{ST}$$

$$sval(b, next, st, def) \mapsto (b', st') :$$

In short, for $type(Def) = pattern$, all next-constraints need to hold, as well as the pattern t_i. For $type(Def) = partition$ all next-constraints need to hold, as well as all possible partitioning alternatives, and the recursive checks for each of the partitions.

The prefix validation of a stream can now be defined by recursion:

$$b_0, St_0 = true, ((), init)$$

$$b_i, St_i = sval(b_{i-1}, \mathcal{S}_i, St_{i-1}, Def)$$

An empty stream is valid and has a pattern state with no next values and initial state of pattern. The validation a stream with a new item is determined as the validity of the prefix (with a computed state), and the validity of the new item given the state.

A finite stream is valid if the complete prefix is (stream-)valid and the all elements of the pattern state are in an accepting state.

$$|S| = N \wedge sval(b, S_N, Def, V) \mapsto (true, V') \wedge \bigcup_{t \in t(Def)} t(V') \in t.accept$$

4.3.4 Validation Complexity

Based on the formalization in the previous section, a number of properties on the complexity of stream schema validation can be established

Theorem 4.3.1 *The space needed for Schema validation is finite, if the set of recursively nested schema definitions is finite, regardless if the validated stream is finite or infinite*

Proof By definition of $\mathcal{ST}$, $\mathcal{ST}$ without considering the recursive state contains a finite number of values. If $type(St)='partition'$, a finite number of nested states is contained, otherwise there are no nested state. The number of nested schema definitions and thus states is limited by requirement of the theorem and also in practice.

Theorem 4.3.2 *cost of checking a new item without recursion is polynomial.*

Proof If there is no recursion (no partitioning) then the cost of checking is $O(|NC|) + O(|activestates|)$, where $|NC|$ and $activestates$ denote the number of next-constraints and active pattern states respectively.By definition of $\mathcal{D}ef$, there is a linear number of rules for each type and the cost of checking a next-constraint as well as choosing the relevant edges for a state in the pattern automaton can be considered a constant.

Theorem 4.3.3 *The cost of checking a new item with recursion is $O(n^m)$, where $m \approx log_n(|\bigcup Def|)$ (nesting depth)*

Proof The cost of check one level of recursion is $O(n)$ (previous theorem), and at each partitioning a single partition is chosen. Thus for m levels of nesting, the cost will be $\prod_{m} O(n) = O(n^m)$

4.3.5 Validation in Presence of Disorder

Checking a disorder constraint requires additional overhead in terms of space and computational complexity. We define two variants on how this validation can be performed: 1) with a known ordering relation (as a parameter to validation, expressed as a next-constraint); and 2) with no known ordering relation.

The first variant can be implemented by checking/restoring order according to the ordering relation, then performing the ordered variant of validation. The additional space required is linear to k, and the additional cost is the cost of sorting within sliding window of k.

For the second variant, the arrival position of items in the stream needs to be kept to work over partitions. To perform the validation, all permutations allowed by k need to be generated in order to check if at least one matches all constraints specified in the schema, and this enumeration needs to be performed for each newly arriving item. To check *next* and *partitioning* constraints, it is sufficient to keep k values around, and the effort for enumeration is $k!$. For *pattern* specifications, the situation is more complicated, since the permutations might affect the whole pattern instance, thus requiring state to be kept for the full instance of the pattern including all the k-permutations. This is similar to the solution in [70].

4.3.6 Practical Issues

In many use cases, explicit validation of streams is not needed, since the stream constraints are guaranteed to hold by the source producing the stream. Similarly, in a distributed stream processing setting, only untrusted data needs to be validated, which may be only a portion the streams used. Nonetheless, for situations in which validation is required (e.g., untrusted input), it should be possible to perform it without significant overhead. The formal definition shows that this is possible.

When a validation failure is detected in a stream, we may terminate processing of the stream. Such an approach might not always be desirable, as it limits the ability of a SPE to deal with unexpected data. One possible alternative is to treat validation as a normal stage in query processing (just as pattern matching), and allow the programmer to capture failures and subsequently disable schema-based optimizations that are not valid anymore in an on-the-fly manner. Alternatively, the SPE could relax the schema or trigger schema evolution to a new version that reflects the changed properties.

In terms of development, clearly building a stream validator from scratch is always an option. In following subsections, we propose two other alternatives.

Validation Using Existing Validation Framework

We implemented [83] a large part of Stream Schema based on the Xerces XML parser and validator [14], since it already provides most of the operations and data structures needed for the validation of Stream Schema. We extended XML schema with the new Stream Schema constraints, and changed the XML parser so that it can consume a root sequence instead of a root element. Each item in this sequence is first validated against the set of item schemas using the standard XML schema validation mechanisms.

The existing operators in Xerces were then re-used to express the stream constraints. The current implementation does not support checking nested schema definitions.

Validation Using Continuous Queries

As an alternative, Stream Schema can be translated into a continuous query, since the operations required for stream schema validation match closely the set of commonly available expressions and operators in SPE and CEP systems. If a matching operator is not available, such a system would not benefit from the optimizations in this area (e.g., Aurora does not have pattern matching, so checking and using pattern information does not provide benefit). For such systems, a subset of Stream Schema, without patterns can still be useful.

4.4 Integration into Processing Models

The next step after defining stream schema is to embed it into the specific data and processing models of SPEs. Each of these systems uses a somewhat different model, but for all of the systems we have evaluated, a straightforward integration is possible (with one exception that we highlight at the end of the section).

To perform this embedding, the following steps are needed: 1) The abstractions of *item*, *item schema* and *attribute* are mapped to their concrete counterpart in each SPE; 2) The *stream data model* needs to be checked for compatibility; 3) *Implicit schema constraints* of a SPE need be expressed in Stream Schema; 4) *Existing Schema-like capabilities* of a SPE need to be checked against the capabilities of Stream Schema.

Now we discuss details of each of these steps for a number of well-known processing models.

1. In relational systems (Aurora/Streambase [16], CQL [24], the SQL pattern extension [22], Gigascope [36], Cayuga [40], CCL [2]), a stream of homogeneous items are used, where *items* are flat relational tuples with attributes. For XQuery streaming [29], a stream can be heterogeneous (individual items validate against different XML schema definitions), where items are atomic values or XML nodes, accessible by XPath expressions. For such systems, we would need to define an accessor function (or attribute) for each valid XPath expression.

   SASE [17] and ESPER [7] also use heterogeneous streams, with flexible item-schema models and access paths. All these item-oriented aspects cleanly map to our formal model.

2. For the *stream data model*, most models assume a totally ordered sequence of items as a basis (which can be defined by next-constraints in Stream Schema) with some relaxations to this ordering: CQL uses a sequences of batches [24] as its stream model, in which the stream has a partial ordering on a timestamp value and the items with the same timestamp do not have any order among them. This can be defined in Stream Schema using a next-constraint with the $\leq$ relation, instead of the $<$ relation used for defining a total order. Other approaches use k-constraints [25] or Slack (Aurora) to give a bound to the degree of out-of-orderedness (with the same semantics as our disorder constraint).

3. Many stream processing models define *implicit timestamp attributes.* In Stream Schema, these implicit constraints can be expressed using an item schema and next constraints capturing the appropriate order of values. Since some systems (SASE, Esper, XQuery) do not require timestamps, we chose not to make timestamps a required part of Stream Schema.

4. While most SPEs provide some schema-like definitions, the stream-oriented aspects of these schemas are often restricted to some ordering properties and several dynamic properties (e.g., arrival delays). A relatively closer match is the possibility to define a stream using a query (Esper and Coral8). In this approach, the query specification (filter, pattern) would imply similar schema constraints as our Stream Schema.

StreamBase/Aurora provides a schema-like operator specification `(On A, slack, GroupBy` B_1, B_N`)`, expressing an order on an attribute A, limited disorder `slack` and the possibility to group by the attributes specified in the `Group By` clause. All of these constructs can be mapped to Stream Schema (next-value, disorder, partition by value).

Gigascope uses ordered attributes (representing timestamps, sequence numbers, etc.) of three different types: a) strictly/monotonically increasing/decreasing; b) monotone non-repeating; and c) increasing in group. Ordering a and c are expressible as next-constraint (with partitioning for c). The precise definition of b) cannot be derived from the informal definition in the Gigascope paper. If the purpose is to express non-repetition of values in an infinite sequence, this cannot be validated, since all values need to be kept. Otherwise we could also express it as a smaller ($<$) next constraint.

4.5 Related Work

Research on schema-based optimizations dates back into the early '80s and such techniques have been successfully exploited in relational [34] and object-oriented [54] query languages. For XPath/XQuery, there has been work not only for persistent XML [49], but also for streaming XML [79]. These approaches are a subset of what is expressible in Stream Schema, since they focus on the contents a single item or document, not a possibly infinite stream.

In developing Stream Schema, we have chosen to focus on common structural constraints that are useful in semantic checking of continuous queries. In following, we overview a few other approaches to designing and using metadata in stream processing. Our decision to focus on structural constraints was largely influenced by this related work.

K-Constraints [25] is the work most closely related to ours. K-Constraints tell a certain behavior of the stream using constraints within a limited range, denoted by the adherence parameter ***k***. They introduce three types of constraints: 1) Referential Integrity Constraint (for joining streams), 2) Ordered-Arrival Constraint (on single streams), and 3) Clustered-Arrival Constraint (on single streams). Then, one can exploit the constraints to reduce run-time state for a wide range of continuous queries.

Yet again, this is orthogonal to our work, since k-Constraint address relationships between streams, unorderedness within streams and can be seen as dynamic constraints. We do not give an absolute range in terms of tuple count where our constraints are valid. Our constraints describe different connections, like dependencies between items by next rules or keys. What our solution does not do compared to k-Constraints, is that we describe the individual streams in an aggregated stream, whereas k-Constraints can describe dependencies between different streams. On the other side, we can describe streams more fine grained by telling exact dependencies between consecutive items, instead of just telling that the item being sought will appear in the range of the next k elements, if it appears. Additionally we have introduced number of new constructs (i.e. patterns, stream keys, stream combination, etc) which can describe different aspects of data streams.

Some previous work have looked at capturing constraints for sensor data and other applications where the data may be error-ridden or have a highly dynamic structure. Such data may not conform to any strict constraints. To handle such applications, the use of soft constraints has been proposed [64].

Finally, we have chosen to focus on structural constraints within a single stream since this is already a very rich (and under-studied) area. Constraints between streams have been studied by others [38, 25, 53, and others] although these solutions also tend to depend on specific processing models, join algorithms and dynamic properties. For example, Golab et al. [53] extended the SQL DDL to define three stream integrity constraints (Stream Key, Foreign Stream Key, and Co-occurrence), which are defined across streams for particular time windows. The main goal of these constraints is to reduce the cost of joins between streams using join elimination and anti-join elimination.

4.6 Conclusions

It is well-known in data management that static metadata, if they are explicitly specified, can be used to check data consistency, improve application modeling, and permit new forms of semantic query optimization.

In this chapter we introduced Stream Schema, our formal framework to capture static stream metadata. It contains a small set of schema elements and a recursive methodology to combine them together. As we will show in next chapter (Section 5.2), Stream Schema has considerable power and can be exploited for a wide range of optimizations, static analysis and constraint separation in modeling streaming applications.

As with any schema formalism, there is a clear trade-off between the expressive power of the schema and the cost of validation (checking whether a stream conforms to a schema). In this chapter, we also presented *prefix-validation*, a mechanisms to validate data streams against Stream Schema. Through complexity analysis we proved that if the set of nested schema definitions is finite, the space needed for validation is finite and cost of checking a new item is exponential to nesting depth.

Finally, by taking a close look at a representative set of existing SPEs, we explained how Stream Schema can be embedded to a wide range of stream processing models.

As avenues for future work, we foresee two natural extensions: first, integrating relationships between streams [53]. To investigate if and how foreign key relationships can be suitably expressed. Second, completing our validation implementation and further investigating alternative ways to react to violations of constraints within a stream.

Another challenge is exploring ways to build Stream Schema. Stream Schema may be designed manually, or potentially discovered using stream mining [50] or pattern mining. Alternatively, they could be derived from business process or workflow descriptions. Both of these areas deserve further investigation as they will improve the sophistication and usability of stream systems.

Chapter 5

Applications of Stream Schema

5.1 Introduction

5.1.1 Motivation

It is well-known in data management, that semantic constraints, if they are explicitly specified, can be used to check data consistency, improve application modeling, and permit new forms of semantic query optimization, specifically the application of optimizations that are correct (and potentially highly efficient) over data satisfying a given set of constraints.

While the use of metadata is well-established in relational and XML databases, the same is not true for data streams. The existing work mostly focuses on the specification of dynamic information such as constraints on arrival rates, have long been exploited for optimization [85]. However, beyond a few limited proposals (including K-Constraints [25] and Gigascope [36]), structural and semantic constraints on stream data have not been exploited in a systematic way.

In this chapter, based on Stream Schema knowledge, we present a set of semantic query optimization strategies that both permit compile-time checking of queries (for example, to detect empty queries) and new runtime processing options. Furthermore, we show how Stream Schema can greatly enhance programmability of stream processing systems.

5.1.2 Contributions

The main contributions of this chapter are:

- a suite of runtime optimizations enabled by Stream schema

- an analysis of how Stream Schema can be used in the static analysis of queries to simplify (or minimize) the queries

- a proposal to improve the programmability of data streams by encoding parts of the constraints in Stream Schema.

- two case studies (one centered on Linear Road Benchmark and the other on an RFID-based supply chain application) implemented in two different SPEs demonstrating the above uses of Stream Schema.

It should be noted that stream validation (and type annotation) is another important application of Stream Schema, but since we have already discussed this topic in previous chapter, we do not repeat it here.

5.1.3 Outline

The rest of this chapter is structured as follows. Since the most well-known application of schema knowledge is enabling optimizations, we start out with Stream Schema-enabled optimizations for data stream processing in Section 5.2. Next, we explain how Stream Schema can be used in in static analysis of streaming queries, as well as in modeling streaming applications in Sections 5.3 and 5.4 respectively.

Two cases studies (Sections 5.5, 5.6) and their relevant experiments show the applicability and benefits of Stream Schema on different models, workloads and implementations.

Finally, after giving an overview of related work in Section 5.7, we conclude this chapter in Section 5.8.

5.2 Stream Schema-enabled Optimizations

The constraints provided by Stream Schema are applicable to a large range of operators and expressions. In this thesis, we focus on optimizations based on the *stream* aspects of Stream Schema; optimizations based on item schema specifications are similar to existing schema-based optimizations [34, 54, 49, 79] and will not be discussed here.

Stream Schema provides the metadata to perform optimizations, so an important class of optimizations enabled by stream schema is not *new* in a strict sense, but in fact well-understood in terms of their mechanism and benefit. For example, rewriting a window's type from sliding to tumbling in order to use a simpler evaluation mechanism with less CPU and memory cost [29]. The important contribution of stream schema is to formally capture whether an optimization is applicable. It also opens up the possibility of automatically applying these optimizations and systematically considering (and comparing) different possible optimizations within a stream optimizer.

In the following, we present an overview of the classes of optimizations for which Stream Schema is beneficial. While explaining each class, if applicable, we will point to existing works in the database and data stream communities, which have used similar techniques. Later, in Sections 5.5 and 5.6, we will discuss selected optimizations in more detail and show the benefits experimentally.

5.2.1 Pipelined Execution

When strictly following the definition of semantic windows[1], e.g., *forseq* [29], such a window could be a pipeline breaker: the items bound by such a window can only be processed when the end condition has been successfully evaluated. For many streaming applications, this behavior is undesirable, since all following operations (such as aggregation) can only be started after the window has been closed, and thus an additional amount of latency and memory consumption is incurred. In addition to other preconditions purely decidable at the language level, two important conditions need to be fulfilled to allow this optimization: 1) every open window will be closed eventually, 2) no window fully subsumes another one, in other words, windows will be closed in the same order they were opened. A detailed analysis on this optimization is given in the LR case study in Section 5.5.

5.2.2 Stream Data Partitioning

The volume of data that needs to be processed in a real-time SPE can easily exceed the resources available on a centralized server. A well-known approach to tackle this problem that has been used in Distributed SPEs is data stream partitioning. This approach requires the splitting of resource-intensive query nodes, into multiple nodes each working on a subset of the data feed [35].

Johnson et al. [61] propose a solution to partitioning a stream query workload based on a set of queries. Their approach includes two steps: 1) finding a partitioning scheme which is compatible[2] with the queries; 2) using this scheme to transform an unoptimized query

[1]Also known as predicate-based windows.

[2]Partitioning set P is compatible with a query Q if for every time window, the output of the query is equal to a stream union of the output of the Q running on all partitions produced by P.

plan into a semantically equivalent query plan that takes advantage of existing partitions.

The unspoken assumption of the Johnson et al. work is that the data (not just the queries) are actually partitionable by the schemes produced in their first step. The partitioning constraints of Stream Schema can be used to determine which (if any) of these schemes can actually be used, thus making this approach fully automatable. In the first case study (Section 5.5), we will show how this optimization technique can improve performance of the LR implementation.

5.2.3 Window/Pattern Optimizations

Here we present two types of optimizations for window and pattern queries:

- The pattern description feature of Stream Schema can be used to simplify the overlap in the specification of window or pattern instances. Specifically, if one can determine that a new window (or pattern) can only start after the previous one has been completed, then one may be able to optimize the processing of windows (or patterns). To be more concrete, here are two examples from different frameworks:
  - *Sliding* and *landmark* windows are more expensive compared to *tumbling* windows, since the number of open windows is potentially much higher, and more checking may be needed[3]. Using the information specified in Stream Schema, a query written using landmark or sliding windows can be rewritten into a tumbling window. An example of this rewriting is given in the LR case study later on.
  - One of the constructs in the SQL's Pattern Matching extension [22] is the `SKIP TO` clause which determines where we should start looking for the next match once the current one has been completed. Two common options are `NEXT ROW` and `PAST LAST ROW` meaning we should start from the row after the next row or from the last row of the current pattern respectively. Using a pattern constraint over the stream, one can rewrite the query to have more efficient `SKIP TO` options. For example, if the Stream Schema defines a stream as repetition of the pattern `AB*C` and the query matches the pattern `AC`, we can safely replace the value of the `SKIP TO` option with `PAST LAST ROW`. The new query is semantically equivalent with the original one, but cheaper because it avoids performing hopeless matches.

[3]In terms of number of overlapping windows: landmark > sliding > tumbling.

- Removing existing structure from window/pattern specifications to only check what a schema does not already provide. The following is an example for pattern matching systems: Complex Event Processing (CEP) systems usually use Finite State Machines (FSMs) for pattern matching [40, 17, 43]. Using the information of the patterns already present in the stream, it is possible to simplify the query FSM by fully or partly decomposing it, a technique commonly used in other areas [42]. Such decompositions can improve the performance of the CEP systems by reducing the number of states or relaxing the transition rules. For example, if we know that incoming items already comply with some patterns in the stream's schema, rechecking these sub-patterns is unnecessary. Figure 5.1 depicts this optimization for the query `ABABA` over the stream `(AB)*`.

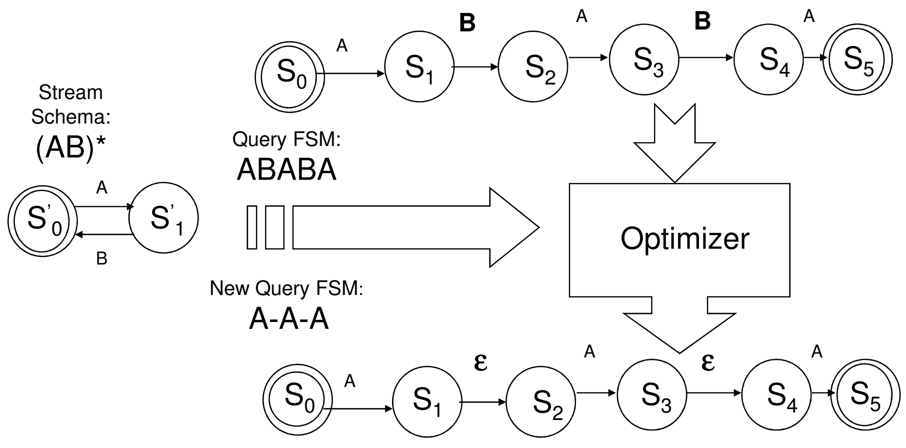


Figure 5.1: *Pattern FSM Decomposition*

5.2.4 State Reduction

Many stream operators (e.g., join, group-by, and sort), maintain state in order to generate the correct results [27]. Most of the optimization techniques for stream processing aim at reducing the number of maintained states to minimize the memory/disk cost. These techniques in general fall into one of two categories: 1) avoid keeping items at all: that is, avoid materialization; and 2) purge states after certain evaluation steps. In the following, we present some guidelines on how Stream Schema provides hints to make these techniques applicable.

Avoiding materialization

In SPEs, newly arrived items are fed into the open windows to determine if they will contribute to the output results. If there is a way in which we can make sure that an incoming item will not contribute to the output result, we can safely drop that item, avoiding unnecessary resource allocation.

Stream Schema can help in different ways to ease making such decisions. The following are some examples for a stream join operator:

- differences in partition by value bounds: if two streams involved in the join have a partitioning constraint on the join attributes, and the $\boldsymbol{n}$ (the maximum number of partitions) is not the same, one can drop the items with the missing values from the respective stream (since they will never match match any tuple from the other stream)
- only items which comply with the next-constraints of both streams have a chance to successfully participate in a join.
- if any of the streams involved in the join is heterogeneous and some of its item schemas do not include the join attribute, they can be eliminated

State purging

In some cases, Stream Schema constraints allow an operator to purge parts of its state. For example, the join operator needs to keep track of a number of items, as there might be matches for them in the future. If based on, for example, a monotonic next-constraint in the stream schema, one can make sure that certain items will never appear again, the join processor can purge the state it has been keeping for those items [44].

5.2.5 Join Cardinality Reduction

Different cardinalities in the join operator impose different levels of complexity and cost. Providing more precise information on the cardinality of the join allows the query optimizer to choose the right algorithm and implementation [56]. Stream schema can be considered as source of such information. For example, if the join attribute is monotonic over one (both) of the streams, uniqueness of the attribute can be deduced, resulting in one-to-many (one-to-one) join type instead of many-to-many join type.

5.3 Impact On Stream Processing Semantics

The presence of stream schema can have a profound effect on the semantics of operations as we show on several examples here.

5.3.1 Static Check for Non-Executable Expressions

The system can use Stream Schema to either output a warning or to abort execution in the following situations.

- *Non-executable predicate-based windows*: for example, imagine Web Server log where log entries for individual users comply with the pattern `login browse`* `logoff`. Now if a query defines a window to be closed on the occurrence of a `browse` item, this window might not close, since occurrence of the `browse` item is optional in the pattern specification. The system can therefore issue a warning.
- *Execution of blocking operators*: for example, if a blocking operator (e.g., a sort) is applied on a stream and from an analysis of the schema a system can determine that the execution may be infinite, a system can abort the operation (or issue a warning).
- *Empty results*: for example, if the pattern query $\mathtt{AC}^{+}\mathtt{B}$ is applied over a stream with the schema of $\mathtt{(AB)}^*$, a *no-result* warning can be issued.

5.3.2 Extended Set of Runnable Expressions

If the input stream of stream engine satisfies certain schema constraints, the engine might be able to change a blocking operator into a non-blocking one (and in doing so, make the operator *runnable*). As a simple example, a blocking sort operator may be removed from a query plan, if the stream is known to comply to a schema that guarantees the same order. A more detailed example is given in the Linear Road case study (Section 5.5).

5.4 Decoupling Streaming Applications

Stream Schema can be used to simplify modeling and developing streaming query applications. Looking at the state of the art, one can conclude that streaming queries are written in a very explicit manner: all possibly relevant predicates and expressions are directly expressed in the query (to ensure correctness), and also often manually arranged (to achieve high performance). By doing so, predicates from two domains are mixed: 1) predicates to describe the desired behavior; and 2) predicates to capture semantic constraints.

The use of Stream Schema enables a different approach. Queries can be written to extract the desired results only. There is no longer any need to provide constraints regarding data consistency and/or structure as part of the query itself.

This separation of query and structural constraints allows for significant improvements in how streaming applications can be developed:

- *Simpler queries*: queries express only the data to be retrieved and can thus be more easily reused over different streams
- *Simpler domain* or *semantic constraints*: constraints need to be declared once and can be re-used for multiple queries
- *Decoupling development of query and schema*: so that both can be designed and evolved separately

In our second case study (Section 5.6), we will give an example of this modeling approach.

5.5 Case Study I: Linear Road Benchmark

To check the expressiveness of our schema proposal and determine the usefulness of its applications in stream processing, we used the Linear Road Benchmark [23] implementation in Streaming XQuery [29]. We introduced LR benchmark in Section 4.2.2. It specifies different scale factors L, corresponding to the number of expressways in the road network. The smallest L is 0.5. The load increases linearly with the scale factor.

It should be noted that XQuery is an interesting target for Stream Schema, since its data model does not have any stream-oriented implicit constraints, it uses semantic windows, and allows arbitrary nesting of expressions. The schema for Linear Road has already been given in Figure 4.1, so we will omit it here. Currently, no optimization framework for a data stream system is known to exist, hence the optimizations are discussed at a formal level and implemented by manually adapting the queries.

The MXQuery [29] implementation of Linear Road benchmark uses a combination of continuous XQuery expressions and dedicated stream stores to express the streaming queries, as shown in Figure 5.2. In total, 7 threads were used, 4 driving the output stream, 3 for intermediate results. The full queries are given in Appendix A.

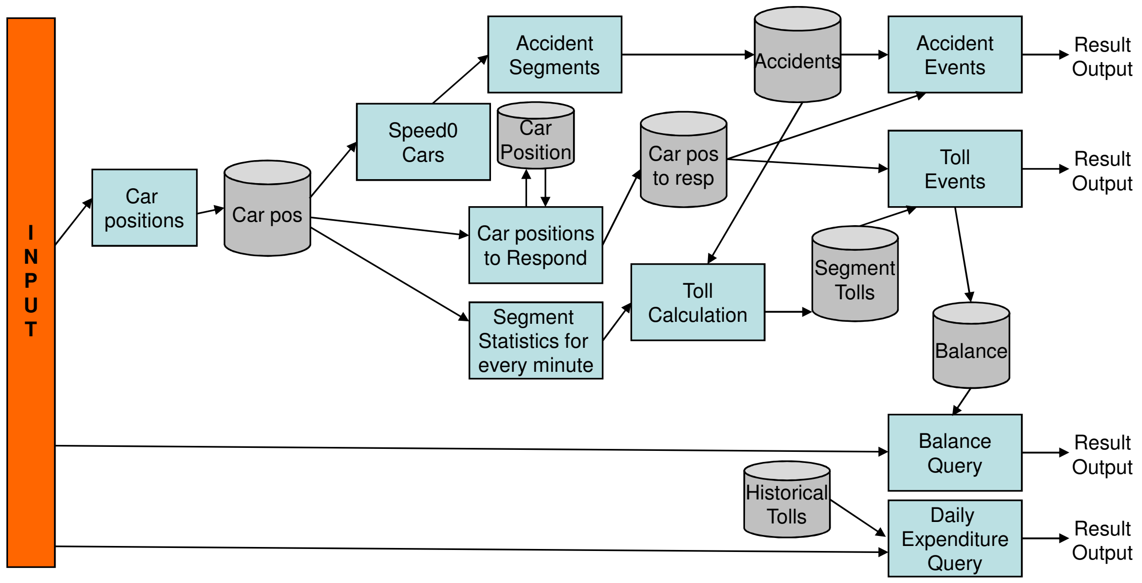


Figure 5.2: *MXQuery Linear Road Implementation [29]*

5.5.1 Optimizations

Query Rewrites for Pipelining Execution

As described in the optimization section, pipelined window execution is an important factor to reduce latency and memory consumption. For semantic windows, an important precondition is that windows will be closed in the same order that they were opened, since otherwise the execution would be blocked until the *complete* window is ready. In case of sliding and landmark windows additional information is required which can be derived from the stream schema. For example, assume the window expression of the Accident Detection Query, shown in Listing 5.1.

```
forseq $w in $ReportedCarPos sliding window
  start curItem $s_curr, prevItem $s_prev
    when $s_curr/@minute ne $s_prev/@minute
  end curItem $e_curr, nextItem $e_next
    when ($s_curr/@minute +2) eq ($e_next/@minute)
```

Listing 5.1: *An Excerpt from the Accident Detection Query*

With two propositions, we show that windows are ordered and hence we can pipeline the results.

Proposition 5.5.1 *In Accident Detection Query (Listing 5.1), windows will be opened in* *a strictly increasing time order.*

Proof By definition of sliding windows in XQuery, for each incoming item, at most one window will be opened. Now we show that these windows are strictly ordered with respect to time attribute of their first element. The start condition of the window specifies that a window should be opened if the time attribute is different between two adjacent elements in the stream (inequality denoted by *ne* operator). In addition, the Stream Schema description states a $\leq$ relationship between the time attributes in the stream, meaning that the only difference of time attribute values can be an increase. As a result of both query and schema constraints, the desired order is guaranteed to hold.

Proposition 5.5.2 *In Accident Detection Query (Listing 5.1), windows will be closed in the same order they were opened.*

Proof We prove by contradiction. Consider two arbitrary windows w and w' in which w was opened *before* w'. According to the window start condition in the query this means

$$\texttt{\$s\_curr}_{w}\texttt{/@minute < \$s\_curr}_{w'}\texttt{/@minute}$$

now assume that w is closed *after* w', meaning

$$\texttt{\$e\_next}_{w}\texttt{/@minute > \$e\_next}_{w'}\texttt{/@minute}$$

but from the LR Stream Schema (Figure 4.1) we know that the values of *minute* attribute is non-decreasing, hence

$$\texttt{\$s\_curr}_{w}\texttt{/@minute + 2 = \$e\_next}_{w}\texttt{/@minute}$$
$$\texttt{\$s\_curr}_{w'}\texttt{/@minute + 2 = \$e\_next}_{w'}\texttt{/@minute}$$

this is a contradiction.

Along these lines, we can draw similar conclusions for other queries and therefore guarantee the safety of pipelining.

Data Partitioning

As we described before, the position report stream of the LR benchmark can be considered as a combination of multiple position report sequences from different expressways and different directions. Depending on the nature of the continuous queries over the LR input stream, it might be possible to partition this stream along the XWay and DIR dimensions, and to process the queries independently and in parallel.

Among the LR continuous queries, the Account Balance Query is the only one which performs computation over more than one expressway or direction. Therefore, we can easily parallelize the execution of the other queries. In below, we show this for two representative queries[4]:

- *The Accident Detection Query*: it uses a group-by and stream keys as part of the grouping predicates, so this is trivially parallelizable. *Toll Calculation* is analogous.
- *The Accident Notification Query*: for each incoming position report that has fulfilled the notification preconditions, it retrieves the accidents for the *same* expressway and the *same* direction and then notifies the vehicle about accidents in its neighbor segments (if any). Therefore, this query can also be executed for each stream key value independently.

Moreover, the Car Positions To Respond Query is just a simple filtering module to select those reports which need to be responded to (reporting a non-leaving segment crossings) and it can be also parallelized along expressways and directions. Consequently, as highlighted in Figure 5.3, a significant part of the query plan can be parallelized.

5.5.2 Experiments and Results

In order to validate the optimizations spelled out in this case study, we re-created the experimental setup of Botan et al. [29]: All experiments were run on a dual-CPU AMD system with single-core (pipelining experiment) and dual-core (partition experiment) Opteron 2.2 GHz processors and 6 GB RAM. A Sun 1.6_10 64-bit JVM with a heap size of 3 GB respectively 5 GB was used.

The type system of MXQuery was extended to handle Stream Schema stream definitions in a similar manner as regular XML Schema user-defined datatypes. The compile-time checks for blocking queries were extended to incorporate the ideas outlined in Section 5.3. On the optimizer, we added the rules for the schema-driven rewrites. Since the queries used were carefully tuned and chosen to take advantage of the implicit schema knowledge, we created a baseline using semantically equivalent queries that do not use schema knowledge.

[4]The actual XQuery expressions of these queries can be found in Appendix A.

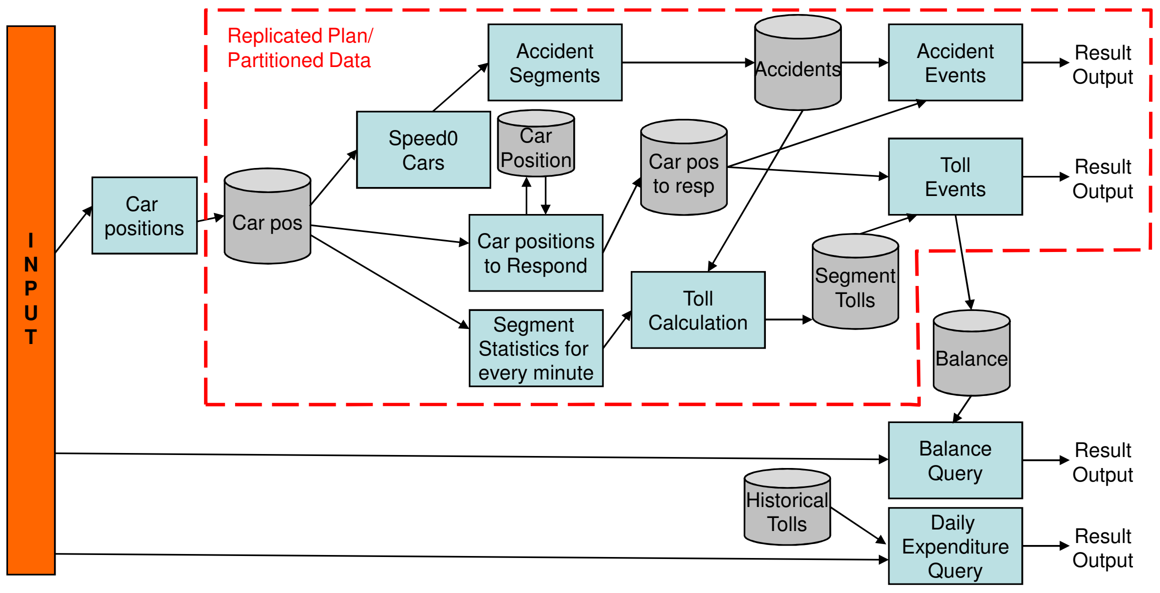


Figure 5.3: *MXQuery Linear Road Implementation, Extended with Data Partitioning Information*

The experiments were aimed at highlighting aspects of Stream Schema that are not well-covered by related approaches, therefore we always kept accurate item schema information even for the baseline measurements.

Pipelining Experiment

For Linear Road, most window constraints are on the attribute 'minute', and the resulting computations on the window contents to calculate statistics, accidents and tolls all need to be performed when the value of the minute attribute changes. Consequently, without pipelining, the response time requirement of 5 seconds is violated at the minute change moments (due to the sudden computation spike), even though unused processing capacity is available during the minutes. Schema information can be used to enable pipelining in window processing (see Section 5.5.1), and thus alleviate the issue. In the experiments, this effect was clearly visible: While running the queries without the schema information (and thus without pipelining) the highest scaling factor we reached was $L = 2.5$, using Stream Schema we were able to scale to $L = 3.5$.

Stream Partitioning Experiment

A second experiment was geared toward partitioning the stream in order to parallelize the processing. When the results of a query or a set of queries can be computed independently for each value of a partitioning attribute, the workload can be distributed over multiple cores or machines. As explained in Section 5.5.1, the workload of linear road can be partitioned along the XWay and Direction attributes. Since the level of parallelism present in the original setup was only enough to saturate 2 cores on the experimental platform (and 4 cores being available), the stream was split into two substreams with the equivalent query plans, sharing only the balance store. On this 4-core machines, $L = 5.0$ was reached with partitioning, while $L = 6.0$ was missed, since the maximum observed response time was 8 seconds.

These experiments give a first set of indications that utilizing stream schema knowledge -whether implicitly in low-level or hand-tuned code, or explicitly by using schema descriptions and a query optimizer for declarative queries- can have a significant impact on performance.

5.5.3 Schema-Driven Executability of XQuery Expressions

Since continuous XQuery uses predicate-based windows, it is, in many cases, non-trivial to statically determine whether an open window will ever be closed or not. This is important, since the output of the window may be consumed by blocking operators, e.g., aggregations. Using LR's stream schema , this can be guaranteed. For example, let us focus on the window expression in the Stopped Cars Query (Listing 5.2). This query finds vehicles with speed equal to zero.

```
forseq $w in $ReportedCarPos sliding window
  start curItem $s_curr, prevItem $s_prev
    when $s_curr/@minute ne $s_prev/@minute
  end curItem $e_curr, nextItem $e_next
    when $e_curr/@minute ne $e_next/@minute
```

Listing 5.2: *An Excerpt from the Stopped Cars Query*

Each window produced by this expression will stay open as long as the incoming reports have the same value for the minute attribute, thus one needs to make sure that the value of the minute attribute cannot remain unchanged forever.

Proposition 5.5.3 *Every open window in Stopped Cars Query (Listing 5.2) will be eventually closed.*

Proof The maximum number of reports in a particular minute is $2 * |VID|$, since every vehicle emits two reports per minute. The next report emission (regardless of the source vehicle) belongs to the next minute, consequently closing the window.

The argument for other windowing queries is analogous.

5.6 Case Study II: Supply Chain

The second case study focuses on application of pattern specification feature of Stream Schema for query optimization and application design.

5.6.1 RFID-based Misrouted Item Detection

Assume a supply chain system (e.g., a car manufacturing factory) which attaches RFID tags to manufacturing items (e.g., car parts) to keep track of these items while they are being distributed among different routes. For example, all *gearboxes* should go to destination number *14*. Each item takes a specific route, and each route is equipped with a number of RFID readers. These RFID readers form a tree, as depicted in Figure 5.4.

Typically, continuous queries are used to detect misrouted items. For instance, if a gearbox has ended up at destination 8, this must be detected and reported. Such queries commonly use patterns to describe valid destinations. As an example, the pattern $AB^{*}C$ where A is the entrance reader, B is any intermediary reader, and C is the correct destination reader for a given car part.

With the help of metadata, the performance of such queries can be improved. In fact, in many cases, detecting a misrouted item is possible before it reaches its final destination. Consider readers tree in Figure 5.4. Instead of checking the type of the items at leaf nodes R_6, R_8, R_{10}, R_{14}, the check can be done right after branches R_4, R_7, R_9, and R_{11}, respectively, or at even higher-level branches. For example, as soon as a part that should be going to R_{14} reaches R_3, the system can report the error. To this end, the structure of the tree can be provided to SPEs or CEPs through Stream Schema, making it a structure-aware item tracking.

RFID Readings Schema

The item schema for the RFID readings stream (R) is defined as

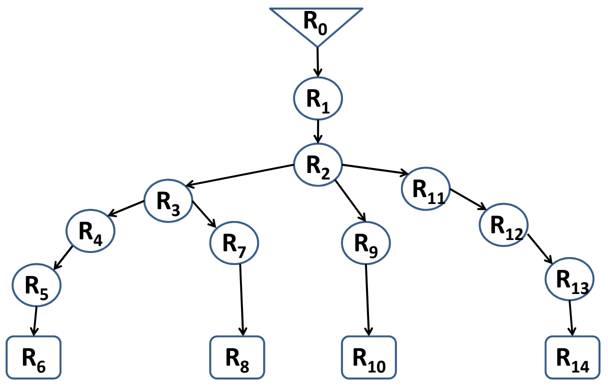


Figure 5.4: *Supply Chain with RFID Readers*

N: R_{IS}
$\mathcal{A}$: {ReaderID,TagID,ItemType,TIME}

and combination of constraints on the main stream of RFID readings is depicted in Figure 5.5. Items in this stream are homogeneous and the TIME values of the readings are non-decreasing (without any disorder). Notice that several readings may arrive at the same TIME (in a batch). The stream can be partitioned along the TagID attribute. Each

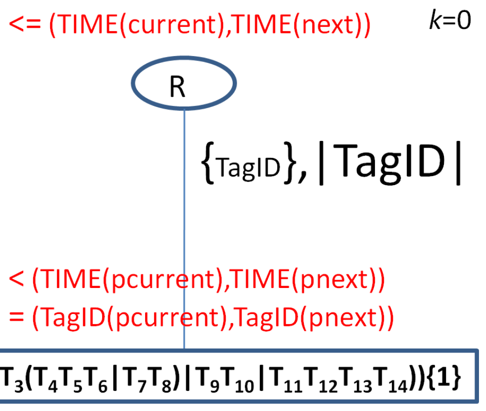

<= (TIME(current),TIME(next))
k=0
R
{TagID}, |TagID|
< (TIME(pcurrent),TIME(pnext))
= (TagID(pcurrent),TagID(pnext))
(T0T1T2(T3(T4T5T6|T7T8)|T9T10|T11T12T13T14)){1}

Figure 5.5: *RFID Readings Stream Schema*

partition corresponds to a particular TagID (tagid) and has a finite number of readings[5].

Each of these partitions conform to a simple pattern constraint, shown at the only leaf of the constraint tree. In this pattern, T_i is defined as below

$$T_i\text{: } \mathrm{R}_{IS_{(\mathrm{TagID}\leftarrow\{tagid\},\mathrm{ReaderID}\leftarrow\{R_i\})}}$$

[5]At most the depth of the reader tree.

For each partition, TIME values are strictly increasing since a particular item is sensed by only one reader at any point in time.

5.6.2 Query Rewrites for Early Detection of Misrouted Items

Assume that the rightmost leaf of the reader tree in Figure 5.4 (ending in R_{14}) is the destination for gearbox. A pattern query for detecting misrouted items to this reader is shown in Listing 5.3 which is written using the MATCH_RECOGNIZE [22] extension of SQL.

```
SELECT InitialS, EngineS, RoutingTime, MatchNo
FROM Readings
  MATCH_RECOGNIZE (
    PARTITION BY TagID
    MEASURES A.ReaderID AS InitialS,
                   C.ReaderID AS EngineS,
                   C.Timestamp − A.Timestamp AS RoutingTime,
             MATCH_NUMBER AS MatchNo
    AFTER MATCH SKIP PAST LAST ROW
    ALL MATCH
    PATTERN(A B∗ C)
    DEFINE A AS (A.ReaderID = "R0")
           B AS (B.ReaderID != "R14")
           C AS (C.ReaderID = "R14" AND
           C.ItemType != "gearbox")
);
```

Listing 5.3: *A Dejavu Query to Detect Misrouted Items*

As depicted in Figure 5.5, the Stream Schema specification encodes the possible paths as a pattern with one repetition. Having this knowledge of the reader tree structure, it is straightforward to find the right replacement for readers in route-checking queries. In fact, for each leaf reader, one can replace the reader with its farthest non-branching ancestor. In case of the above query, R_{14} will be replaced by R_{11} (and analogously, R_{10} by R_9, R_8 by R_7, R_6 by R_4).

5.6.3 Experiments and Results

In our experiments, we used the query described in the previous section. The length of a path was fixed at 30. We have generated readings for 1000 Tags which end up in the gearbox leaf. Misrouting probability was set to 0.02. In our experiments, we have varied the branching position (position where the ancestor of the gearbox leaf has sibling).

We have used DejaVu [43] to implement this case study. DejaVu is a declarative Pattern Matching System over live and archived streams. It is built on MySQL, an open-source database system, and implements the MATCH_RECOGNIZE clause [22] to define patterns with semantic windows. The DejaVu implementation uses finite state machines for the execution and internal representation of pattern queries. The open source memory measurement tool *Valgrind* was used to monitor the memory usage.

In this set of experiments, we measured the memory consumed by the windows, which mostly maintain partial matches. Early decision making allows the system to close the windows earlier, which means less memory consumption. As the results in the Table 5.1 show, the nearer to the root the branches are, the more memory one saves. In this table, the baseline is the case in which the query did not exploit the schema knowledge.

Branching Pos.	Memory (KB)	Saving (%)
2	8	99.4
5	189	87.3
15	711	52.4
25	1234	17.4
30 (Baseline)	1494 KB	0

Table 5.1: *Memory consumption in the Early Detection experiment*

The experiment clearly shows that rewriting the query using Stream Schema reduces the memory usage by keeping less number of pattern states.

Finally, it should be noted that in all variations of this experiment, the detection query remained unchanged and only the schema information was altered. Moreover, in case of changes in the topology of RFID readers, as as long as the entry and the destination readers stay the same, there is no need to change the query. This, if fact, demonstrates the decoupling benefit of using Stream Schema.

5.7 Related Work

There is a body of work on schema-based optimization in relational (i.e. [34]) and object-oriented (i.e. [54]) data management systems. They typically focus on goals like predicate addition and removal, join removal, and empty result detections. For XPath/XQuery, there has been work not only for persistent XML [49], but also for streaming XML [79]. As argued in Chapter 4, the item-level schema information exploited in these works is orthogonal to what is expressible in Stream Schema, therefore these optimizations are complementary to the techniques we outlined in this chapter.

Several approaches [85, and others] have considered how to specify and exploit dynamic behavior (such as arrival rates or dynamic delays) in stream optimization. These solutions are strongly dependent on the processing model. A well-known example is the work of Tucker et al. [84] on the use of *punctuation semantics* to optimize stream operators. A punctuation can be used to unblock operators or output partial results. There are also many proposals for stream constraints that are based on a specific processing model or the language of a specific system, the most common of which are types of window specifications [53, 38]. Such constraints are orthogonal to properties of the data stream itself, properties we call *static* to distinguish them from *dynamic* or processing-model dependent properties. In Stream Schema, we focus on the specification of static data behavior.

Constraints between streams have been studied by others [38, 25, 53, to name few]. These solutions tend to depend on specific processing models, join algorithms and dynamic properties. For example, Golab et al. [53] extended the SQL DDL to define three stream integrity constraints (Stream Key, Foreign Stream Key, and Co-occurrence), which are defined across streams for particular time windows. The main goal of these constraints is to reduce the cost of joins between streams using join elimination and anti-join elimination.

A query-aware stream data partitioning scheme has been proposed in [61]. The authors present methods for analyzing given query sets and choose the optimal partitioning scheme, and also show how to reconcile potentially conflicting requirements that different queries might place on partitioning. The tacit assumption of this approach is that the data (not just the queries) are in fact partitionable by the schemes produced in the first step. The partitioning constraints of Stream Schema can be used to determine which (if any) of these schemes can actually be used, thus making this approach fully automatable.

K-Constraints [25] capture three characteristics of the stream through specifying a limited range, denoted by the parameter K: 1) Referential Integrity Constraint (for joining streams), 2) Ordered-Arrival Constraint (on single streams), and 3) Clustered-Arrival Constraint (on single streams). Given these constraints, one can exploit the constraints to reduce run-time state for a wide range of continuous queries.

5.8 Conclusions

Schema information is invaluable in data management. It facilitates conceptual design and enables checking of data consistency. It also plays an important role in query optimization. In this chapter, we presented a set of different applications for Stream Schema and demonstrated a selected set of them in our two case studies.

We showed that Stream Schema can be used to query optimization. For example, to safely pipeline the execution and partition the data. It can also enable state reduction and query rewrite. Furthermore, we presented an analysis of how Stream Schema can be used in the static analysis of queries to extend the set of runnable as well as non-executable expressions. Finally, we exploited data consistency and structure constraints specified in Stream Schema to simplify the queries in streaming applications resulting in increased decoupling and reuse.

Investigating additional optimizations based on Stream Schema is part of ongoing and future work. This can possibly lead to considering additional stream constraints. Future work also includes combining the use of static metadata (Stream Schema) and a recent proposal [45] which utilizes dynamic substream metadata for runtime optimizations.

Chapter 6

Provenance Management on Data Streams

6.1 Introduction

6.1.1 Motivation and Use Cases

Tracking provenance, exploring which input data led to a given query result, has proven to be an important functionality in many domains such as scientific data management, workflow systems [39] and relational database systems [33]. Previous techniques have been traditionally classified according to their granularity: *Coarse-grained* provenance tracks dependencies between input and output data at a very abstract level (e.g., streams), whereas *fine-grained* provenance does so for individual data items in the input's data collections (e.g., tuples or attribute values).

Surprisingly, in the area of data stream management systems (DSMS), there has been little work beyond *coarse-grained* provenance (e.g., tracking the sensor sources from which a data item originates [86, 68]) despite the wide spectrum of use cases as we show below.

Ad-hoc human inspection: In monitoring and control of manufacturing systems, sensors are attached to machines and to key points along a supply chain as well as on the support infrastructure. Sensor readings are processed by a DSMS in order to detect critical situations such as machine overheating or low inventory. These detected events are then used for automatic corrections and also to notify the human supervisors who need to assess the relevance of these events. To do so, the human operators need to understand from which inputs these events where derived (i.e., the individual temperature readings). This requires *fine-grained* provenance for events. Because of the interactive nature of hu-

Use case	*Provenance generation*	*Provenance retrieval*		
	Relevant events & queries	Lifetime	Retrieval	Response times
Ad-hoc human inspection	all events (events of interest not known beforehand)	time-bound (minutes to hours)	iterative drill-down & point queries	(milli)seconds
Stream query debugging	selected queries & events	debugging session	lookup & replay & interactive drill-down	(milli)seconds
Indicator-based assurance	selected queries, all events	retention period for indicators	Point & Analytic queries	offline
Event warehousing	selected queries, all events	application-dependent	Analytic queries	offline

Table 6.1: *Provenance Use Cases and Requirements*

man inspection, the original events and their provenance become relevant only for short periods of time, but should be provided efficiently to enable interactive drill-down.

If the DSMS outputs a machine overheating alarm event, the user would want to understand which sensors measured high temperature values.

Stream query debugging & diagnosis: The high complexity of streaming queries requires support for diagnosing system behavior, up to the scope of events or even attributes. Provenance helps in exploring the computational steps and the data that led to an observed result and in understanding how errors have propagated. The scope of inspection can be limited to particular queries or events of interest.

Indicator-based assurance: Monitoring and control systems often adhere to strict accountability requirements and need to provide proof for correct operations, which are expressed as indicators. Provenance helps to establish the validity of these indicators by providing the input events and computations they are based on.

Event warehousing: Event warehousing is used to collect raw and derived event streams for mining and analysis. Provenance exposes how events became part of the warehouse. Full provenance needs to be captured to allow complex analysis over such data.

The above use cases, summarized in Table 6.1, clearly show that *fine-grained* stream provenance is a critical requirement in several important application domains, each of which pose a diverse set of requirements in terms of the generation, storage, and retrieval of provenance.

6.1.2 Challenges

In many stream-based use cases including the one above, data is essentially transient, rapid, time-ordered, and possibly unbounded, while queries can behave non-deterministically due to approximate processing or uncertain inputs These characteristics lead not only to stream-oriented data representations and processing models, but also more strict constraints on performance, both of which greatly affect how provenance should be managed. This involves a long list of issues from stream provenance semantics and representation to its computation, storage, and retrieval. A major challenge is to find a solution that balances the amount of data needed for correctly representing provenance with its efficient generation and scalable retrieval. More specifically, provenance management on data streams must deal with the following list of challenges:

- **Online and Infinite Data Arrival:** Data streams can potentially be infinite. This means that we do not necessarily have a full view on all items of a stream (i.e., some items may have not appeared yet). Moreover, it may be impractical or even impossible, to preserve all seen items for later processing. This property of data streams imposes severe constraints on provenance storage and retrieval in DSMSs.

- **Ordered Data Model:** In contrast to the set or bag model of relational databases, data streams are typically modeled as ordered sequences, requiring a provenance model that incorporates order. This ordering, however, can be exploited in providing optimized representations of provenance.

- **Window-based Processing:** Commonly used streaming operators like *aggregation* and *join* are typically processed in DSMSs by grouping tuples from a stream into windows and computing the result over the content of each window, essentially requiring to maintain stateful operators in the query plan. Stream provenance must deal with such windowing behavior in order to trace the outputs of such stateful operators back to their sources correctly and efficiently.

- **Low-latency Results:** Streaming applications have strict performance requirements where low latency should be maintained, even under very high data arrival rates. For example, in ad-hoc human inspection queries, the events of interest are relevant only for short periods of time and supporting interactivity in the system by providing low-latency results is important.

- **Non-determinism:** Some mechanisms applied by DSMSs to cope with issues like high input rates (e.g., load shedding or approximations [80]), unpredictable behavior

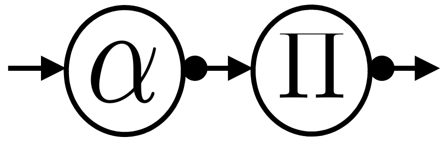


Figure 6.1: *The Continuous Query in our Running Example*

of input sources (e.g., delays or disorder [77]), and certain operator definitions (e.g., windowing on system time or operators with slack or timeout parameters [16]), may lead to non-deterministic behavior. The non-deterministic nature of some DSMSs severely restricts the use of some of the standard techniques developed for database provenance for these systems. For instance, query rewrite techniques used in traditional databases for computing provenance is not applicable in DSMSs, since most of these techniques requires reproducibility of query results to deal with operations like aggregation [51, 37].

In this chapter, we address all of the above challenges, by providing a novel, fine-grained, propagation-based stream provenance management approach which exerts minimal overhead in the system through a number of optimizations.

6.1.3 Running Example

Throughout this chapter we will be using a running example. This example is inspired by the Ad-hoc Human Inspection use case explained above.

Example 1 *Consider an assembly line (AL) in a factory which is equipped with multiple temperature sensors and a continuous query (q_1, depicted in Figure 6.1) to detect overheating in the assembly line. Temperature is measured by sensors at each second and an overheating is reported if the average temperature over a fixed time interval is above a threshold 'thr'. In q_1, the projection operator transforms the output stream of the aggregation operator into a stream of boolean values ('true' indicates that a part of AL is overheated).*

In the scope of this example, provenance can be retrieved for two purposes: 1) identifying the malfunctioning sensors (if an occurred overheating has not been reported), and 2) providing evidence of overheating (the individual temperature measurements) to human inspector.

6.1.4 Contributions

The main contributions of this work are:

- Modeling provenance for DSMSs considering the unique requirements including infinity, order, window-based processing and (in some DSMS) non-deterministic behavior.
- Exploring the different possibilities for computing provenance in a DSMS: *Operator Instrumentation*, *Inversion*, and *Query Rewrite*.
- Providing efficient techniques for compressed representation, computation, and retrieval of provenance in DSMSs.
- Implementation and experimental evaluation of the above techniques on *Ariadne*.

6.1.5 Outline

The rest of this chapter is organized as follows. First, we discuss the provenance design space in Section 6.2. Next, we present an algebra in Section 6.3 to formally define the semantics of stream provenance. The implementation of *Ariadne*, our provenance-aware stream processing engine is explained in Section 6.4. Optimizations and experiments are presented in Sections 6.5, 6.6 respectively. Finally, after reviewing the related work in Section 6.7, we conclude the chapter in Section 6.8.

6.2 Provenance Design Space

In this section we discuss (in Sections 6.2.1 and 6.2.2) alternative ways of how to extend a DSMS to support provenance. the requirements of use cases like the one presented above. In addition to enumerate approaches for generating and representing provenance in a DSMS, we also investigate the trade-offs implied by these approaches with regard to the *properties of the query network* and *frequency of provenance retrieval* in Section 6.2.3.

6.2.1 Provenance Computation

We consider three approaches for provenance computation. (1) Instrumenting the operators of the query network to propagate provenance information, (2) rewrite the query network to propagate provenance using the existing operators of the DSMS, and (3) computing inverses. Provenance computation can either be done *eagerly* by generating provenance while the query network is running or *lazily* be computing provenance when it is requested. Eager computation usually results in a constant overhead in runtime. Lazy computation incurs in higher storage requirements and is only applicable to deterministic networks.

Propagation by Operator Instrumentation

The key idea behind the operator instrumentation approach is to extend each operator implementation so that: (1) The operator is able to output provenance information in addition to outputting the normal operator results. For instance, after outputting one result tuple the operator could output additional tuples that store the provenance of this result tuple. (2) The operator is able to read and interpret provenance attached input tuples. (3) The operator knows how to generate provenance for an output tuple based on provenance attached to an input tuple. By extending all operators of the DSMS in this way, provenance can be computed for query networks that are DAGs of operators, because each operator "knows" how to interpret the provenance produced by the operators that generate its input streams. This approach does not change the structure of the query network for provenance computation if applied eagerly. Lazy provenance computation with this method requires temporary storage of input tuples and replay techniques to process the past inputs by an instrumented query network. Since the execution of the original query network is traced, most issues with non-determinism are dealt with in a natural way. For example, windows based on system time or any external factor (as studied in [28]) are recorded effortlessly, while the *Inversion* or *Rewrite* approaches cannot be used for such networks.

Operator instrumentation does not impose any restrictions on how provenance is represented. For example, provenance could be represented as complete input tuples, using one of existing graph models for provenance, or as sets of tuple identifiers (TID). Operator instrumentation supports provenance computation for parts of a query network by instrumenting a subset of the operators.

Propagation by Query Rewrite

Similar to relational provenance system such as *Perm* [51], *DBNotes* [26], or Orchestra [55], we can compute stream provenance by rewriting a query network q into a network that generates the provenance of q in addition to the original network outputs. However, for this approach to be applicable the query language of the DSMS has to be powerful enough to express the provenance computation for an arbitrary network expressed in this language. Query rewrite can be problematic if applied to non-deterministic networks with windowing operators. For instance, we can not propagate provenance information though operators like aggregation directly without changing the results of this operator (see [51] for a discussion on the topic). To compute the provenance of such operators, a modified copy of the sub-network that generates the input of the operator has to be created and the results of the original sub-networks and its provenance producing duplicate are joined together. If the sub-network uses a non-deterministic function or operator (e.g., a random number generator) in the two copies of the sub-network the function respective operator may produce different results. Thus, the join will fail to attach the correct provenance to each original result. Furthermore, under rewrite the size of the rewritten network grows exponentially in the number of aggregations. Unless the DSMS is extended with additional data types and functions that operate on these new data types, provenance representation is limited to what can be modeled using the natively supported data types of the DSMS.

Inversion

Inversion computes provenance by applying the inverse (in the mathematical sense) of an operator to an output of interest. Examples of invertible operators are join (without projection) and selection, because for these operators the input can be constructed from an output tuples. For non-trivial operations, no true inverse in the mathematical sense exist. Additional information such as the input data of the operator is needed to compute the provenance of an output tuple. For example, value-based windowing requires the reprocessing of (part of) the input stream and propagation techniques to determine which tuples have been grouped together into a window. For such operations, inversion effectively becomes lazy propagation. This approach has the same disadvantage as rewrite for non-deterministic query networks with non-invertible operator, because the replay may produce results different from the original execution. In principle inversion can be applied eagerly or lazily. However, since eager computation also requires replay in most cases, it has no real advantage over lazy for inversion.

6.2.2 Provenance Representation

We now discuss how to represent provenance information *internally* in a system and *externally* to a user.

External Representation for Retrieval

External representations should be informative enough to be interpretable by user. For instance, presenting a set of tuple identifiers as the provenance to the user is not very useful. Complete input tuples seem to be a better way to represent provenance information to the user. An output usually depends on many input tuples. The main problem to solve is to find a representation that enables us to query this information. Two options have been established in the related work: (1) Extend the data model and query language to support provenance [26]. For instance, Karvounarakis et. al. [62] present provenance as a graph and develop a query language tailored for these graphs. (2) Represent provenance alongside with the normal query results using the original data model. For instance, this approach was implemented in the Perm [51]. Using the original data model has the advantage that provenance can be queries using the existing query language of the system, but usually results in a denormalized representation that duplicates original results to “fit in” the provenance.

Internal Representation

Representing provenance as collections of complete tuples would result in a huge overhead for, e.g., queries that generate their outputs from large windows. To reduce the load on the DSMS, we can use a more compact internal representation during provenance computation. For instance, use tuple identifiers as placeholders for complete tuples. Using an internal representation that is different from the external representation, we face the problem that we need to transform provenance between these representations before exposing it to the user. This results in additional processing and may require temporary storage of additional information (e.g., preservation of input tuples).

6.2.3 Summary of Tradeoffs

Using a more compact provenance representation for internal purposes can reduce runtime and storage overhead of provenance generation, but induces additional storage and runtime cost to create the external representation exposed to the user. This approach requires the storage of input stream tuples to be able to access the complete tuples represented by *TID* collections in provenance retrieval. Given that computational overhead is critical in a DSMS and provenance retrieval is expected to be limited to only a subset of the output tuples, the overhead of temporarily preserving input tuples can be expected to be less than the overhead of propagating complete input tuples for non-trivial query networks.

Given the design space and application requirements outline above, a decision for a particular computation approach can be made along those lines: *Non-deterministic* queries mandate *operator instrumentation* and *eager* computation, because this is the only way to ensure correctness.

For deterministic queries, the most important factor is the provenance retrieval frequency. If provenance is requested often, *eager* computation is beneficial, as it has the lower overall cost and usually lower response times. In turn, if retrieval happens more rarely, *lazy* computation becomes more appealing. Determining on which original tuples to compute the provenance carries additional overhead.

Among the computation methods, *Operator Instrumentation* and *Rewrite* are applicable for both *lazy* and *eager*, *Inversion* by its very definition only for *lazy*, since an output tuple already needs to exist before it can be inverted. For all but the simplest queries, *Operator instrumentation* carries much lower runtime overhead than *Rewrite*, but requires changes to operators and possibly also to the optimizer of the system and its cost model. Pure *Inversion* is only applicable for a restricted set of operators and queries, otherwise requiring re-computation. Such re-computations effectively turns *Inversion* into a variant of *Operator Instrumentation*. As a conclusion, we can determine that *Operator Instrumentation* with a compact internal representation is the best general approach. The other approaches are useful as optimizations under certain conditions or if no changes to the query processor are possible.

6.3 Provenance Semantics

6.3.1 Overview

- We develop a sound, declarative notion of provenance stating which conditions the provenance should fulfill. These conditions are based assumption that one would intuitively expect to hold for provenance (Sec. 6.3.4).

- We represent provenance explicitly by extending the stream data model to contain tuples with attached provenance. These streams, called *provenance enhanced output streams* (PEO), are the basis of provenance representation in Ariadne (Def. 3).

- We introduce extended stream operators that generate PEOs instead of normal output streams. These operators serve as an algorithmic specification and have a natural implementation as we will see in Sec. 6.3.5.

Our formalization tackles the challenges of *Order* and *Windowing* by using a sequence-based formalism. *Infinity* is expressed explicitly in the data model and is handled in the query model by defining operators recursively over finite subsequences of an infinite stream. Provenance is defined over finite prefixes of a data stream. We have chosen not to handle *Nondeterminism* at the level of the formal model, since it would greatly complicate the semantics, if it is expressible at all. This problem is addressed by our implementation (Section 6.4). A formal provenance model requires a formal underpinning in terms of a data model and query language, e.g., like the relational algebra in databases. Since there is no generally accepted formal model for data streams, we first establish the necessary definitions for the data model (Section 6.3.2) and operator semantics (Section 6.3.3), using a recursive prefix-based model. We have tried to keep this model as generic and minimal as possible, so that it will apply to many existing DSMS. Even if not all operations of a DSMS map directly to the formalism, the provenance model itself can be easily adapted (e.g. to deal with different ways on how to determine window boundaries [28]).

6.3.2 Data Model

We model streams as (possible infinite) sequences of so-called *stream items*. Each stream stores items of a specific type. In our model we use three item types: *tuples*, *windows*, and *join-windows*. Tuples are lists of attribute values that conform to a given schema (attribute name and domain pairs). Windows are ordered sequences of tuples and are used to define stream operators that compute output items based on subsequences of their input stream. Similar, join-windows, storing two windows from different streams, are used in the definition of the join operator.

Definition 1 (Stream Item and Stream) *A* ***stream item*** *i is of a type $Type \in \{T, W, JW\}$ that defines its structure: A stream item of type T (a* ***tuple****) is an element $t = [tid, a_1, \ldots, a_n]$ from $\mathcal{T} \times D_1 \times \ldots \times D_n$ for a list of domains $D_1, \ldots, D_n$ and a set $\mathcal{T}$ of tuple identifiers. Let $\mathcal{T}(t)$ denote the identifier of tuple t which is required to be unique. We reserve the attribute name TID for the attribute that stores the $\mathcal{T}(t)$ value of t. A stream item of type W (a* ***window****) is a finite sequence of tuples denoted as $w = \ll t_1, \ldots, t_n \gg$. A stream item of type JW (a* ***join-window****) is a tuple $jw = [w_1, w_2]$ where w_1 and w_2 are windows. A* ***stream*** *S of type $Type$ is a, (possibly infinite) sequence of stream items of type $Type$ denoted by $\ll i_1, \ldots \gg_{Type}$ ($Type$ is omitted if clear from the context). We use $S[i]$ to denote the i^{th} element of stream S, and S_{Type} to denote that stream S is of type $Type$.*

For example, $S = \ll\ll t_1, t_2 \gg, \ll t_2, t_3 \gg\gg_W$ with $t_1 = [\mathcal{T}_1, 5]$, $t_2 = [\mathcal{T}_2, 7]$, and $t_3 = [\mathcal{T}_3, 12]$ is a stream of type W containing two windows; each of them containing two tuples. Fig. 6.2 summarizes the notations we use in the stream algebra and provenance definitions (some will be introduced later).

6.3.3 Stream Algebra

We now present an algebraic formalization of stream operators. A stream operator produces one or more tuple output streams from one or more tuple input streams. In the definitions we use some auxiliary functions presented in the following. Applying the head function H to a stream S returns the first item in the stream ($H(S) = S[1]$). The result of applying the tail function T to a stream S_{Type} is the original stream with the first element removed ($T(S) = \ll S[2], S[3], \dots \gg_{Type}$). Both head and tail are also defined to return resp. remove m stream items (e.g., $H(S,m) = \ll S[1], \dots, S[m] \gg_{Type}$). The concatenation of a stream item i and a sequence S or of two sequences S_1 and S_2 is defined as: $i \mid\mid S = \ll i, S[1], \dots \gg_{Type}$ and $S_1 \mid\rightarrow S_2 = \ll S_1[1], \dots, S_1[l(S_1)], S_2[1], \dots \gg_{Type}$ where $l(S)$ denotes the number of items in sequence S. For a tuple t, $t.\mathcal{N}$ is the tuple without its identifier.

We first present the definitions for *selection* and *projection* that directly operate on input tuples without grouping them into windows. Afterwards, we present two auxiliary classes of operators, called *windowing* and *join windowing*, that are used in the definition of *aggregation* and *join* presented in the following. In the operator definitions we use *new* to denote a function that generates new TID values for the output of an operator. To not loose generality we only require that *new* is deterministic (it generates the same values for the same input).

Selection: A selection $\sigma_c(I)$ on condition c filters out tuples from a stream that do not fulfill the condition c.

$$\sigma_c(I) = \begin{cases} [new, H(I).\mathcal{N}] \mid\mid \sigma_c(T(I)) & \text{if } H(I) \models c \\ \sigma_c(T(I)) & \text{else} \end{cases}$$

Projection: A projection $\pi_A(I)$ on a list of projection expressions A (attributes and application of functions) projects each input tuple on the expressions from A. In the definition $t.A$ denotes the projection of a tuple t on A.

$$\pi_A(I) = [new, H(I).A] \mid\mid \pi_A(T(I))$$

Windowing: A window operator is a function $\omega : I_T \rightarrow O_W$. I.e., groups tuples from a tuple stream into windows. As examples for a window operators we present *count-based*

S_{Type}	Stream S is of type $Type \in \{T, W, JW\}$
$[tid, a_1, \ldots, a_n]$	Item of type T (tuple)
$\ll t_1, \ldots, t_n \gg$	Item of type W (window) with $t_i \in T$
$[w_1, w_2]$	Item of type JW with $w_1, w_2 \in W$
$H(S)$	First stream item of sequence S
$H(S, n)$	Sequence containing the first n items of sequence S
$T(S)$	Sequence S with first element removed
$T(S, n)$	Sequence S with first n elements removed
$i \mid\mid S$	Sequence S with item i added at the beginning
$S_1 \mapsto S_2$	Concatenation of sequences S_1 and S_2
TID	Name of the attribute for tuple identifiers
new	Function that generates new tuple identifiers for the output of an operator
$t.A$	Project tuple t on expressions A
$t.\mathcal{N}$	Project tuple t on its data (remove TID and/or provenance attribute)
$t.\mathcal{P}$	Project tuple t on the provenance set
$q[O]$	Output stream O from query network q
$S \cap \mathcal{M}$	Remove elements from stream S that are not in M
$S \uparrow \mathcal{M}$	Remove elements from stream S that follow the last element from M
q_O	Sub-network of network q that contains only nodes that influence output stream O

Figure 6.2: *Notations*

windowing and *value-based* windowing. The count-based window operator $\mathcal{C}(c, s)$ groups c (called *count*) tuples from the input into a window and skips s (called the *slide*) tuples before opening a new window:

$$\mathcal{C}(c, s)(I) = \ll H(I, c) \gg \mid\mid \mathcal{C}(c, s)(T(I, s))$$

The value-based window operator $\mathcal{V}(x, r, s)$ groups all tuples into a window that have an x attribute value that is smaller than the x attribute value of the first item in the window plus the parameter r (called *range*). Windows are advanced by s:

$$\mathcal{V}(x, r, s)(I) = \sigma_{x \leq H(I).x+r}(I) \mid\mid \mathcal{V}(x, r, s)(\sigma_{x \geq H(I).x+s}(I))$$

Join Windowing: A join-windowing operator is a function $j\omega : I_T \times I'_T \rightarrow O_{jw}$ that groups inputs from two tuple streams into join-windows. For a join-window $jw = [w_1, w_2]$ we denote the access to window w_i by $jw.w_i$. For example, *value-based* join-windowing ($jv(x_1, x_2, r)$) groups each tuple t from the left stream with all tuples from the right stream that have an x_2 attribute value between $t.x_1$ and $t.x_1 + r$.

$$\begin{aligned} jv(x_1, x_2, r)(I, I') =& [\ll H(I) \gg, \sigma_C(I')] \\ & \mid\mid jv(x_1, x_2, r)(T(I), I') \\ C =& x_2 \geq H(I).x_1 \land x_2 \leq H(I).x_1 + r \end{aligned}$$

Aggregation: An aggregation $\alpha_{agg,\omega}(I)$ partitions its input into windows using the window function ω and computes the aggregation functions from $agg = (agg_1(a_1), \dots, agg_n(a_n))$ over each window generated by ω. Each aggregation function $agg_i(a_i)$ computes a single attribute value from all values of attribute a_i in a window w. We denote the application of an aggreation function $agg(a_i)$ to a window w as $agg(a_i, w)$.

$$\begin{aligned}
\alpha_{agg,\omega}(I) &= a(\omega(I)) \\
a(I) &= agg(H(I)) \mid\mid a(T(I)) \\
agg(w) &= [new, agg_1(a_1, w), \dots, agg_n(a_n, w)]
\end{aligned}$$

Join: The join operator $\bowtie_{c,j\omega} (I_T, I'_T)$ joins two input tuple streams I and I' by applying the join windowing operator $j\omega$ to I and I', and for each generated join-window jw outputs each combination of a single tuple from $jw.w_1$ with a tuple from $jw.w_2$ that fulfills the join condition C.

$$\begin{aligned}
\bowtie_{C,j\omega} (I, I') =& join(H(j\omega(I, I'))) \mid\!\rightarrow join(T(j\omega(I, I'))) \\
join(jw) =& joinl([\ll H(jw.w_1) \gg, jw.w_2]) \\
& \mid\!\rightarrow joinl([T(jw.w_1), jw.w_2]) \\
joinl(jw) =& \begin{cases} X \mid\mid Y & \text{if } [H(jw.w_1), H(jw.w_2)] \models C \\ Y & \text{else} \end{cases} \\
X =& [new, H(jw.w_1).\mathcal{N}, H(jw.w_2).\mathcal{N}] \\
Y =& joinl([jw.w_1, T(jw.w_2)])
\end{aligned}$$

Query Network: A query network Q is a DAG representation of a stream algebra expression q. Each node in Q represents one operator of algebra expression q. In the network Q, there is a directed edge between node x and y labeled with a if operator x uses the output stream of operator y as its ath input. For networks with multiple output streams, we use $q[O]$ to denote output stream O of the network.

Example 2 *In this example we generalize our running example query ($q1$) from Section 6.1.1 to demonstrate the query operator composition. We assume that there are three temperature sensors ($t_1 - t_3$) that output their measurements once a second as a stream T_i with a schema $[time : \mathbb{N}, temp : \mathbb{N}]$. A part of the assembly line is considered overheated, if the temperature averaged over two seconds is above a threshold of* 150. *Overheating is required to be measured once a second. Query q_{ex} presented below is used to output tuples that store the current overheating status of all three temperature sensors. We use attribute* *renaming (denoted by $\rightarrow$) to simplify the presentation.*

$$\begin{aligned}
q_{ex} =& \pi_{t_1,oh_1,oh_2,oh_3,}(\bowtie_{true,jv(t_1,t_2,1)} \\
& (q_1, \bowtie_{true,jv(t_1,t_2,1)} (q_2, q_3))) \\
q_i =& \pi_{avgtemp>thr\rightarrow oh_i,time\rightarrow t_i}(\\
& \alpha_{avg(temp)\rightarrow avgtemp,min(time)\rightarrow time,\mathcal{V}(time,2,1)}(T_i))
\end{aligned}$$

Assume the sensors generated the input streams as shown below, then q_{ex} outputs the stream O presented below.

$$\begin{aligned}
T_1 =& \ll [1,70],[2,166],[3,143],[4,161],[5,170] \gg \\
T_2 =& \ll [1,30],[2,40],[3,37],[4,60],[5,23] \gg \\
T_3 =& \ll [1,130],[2,152],[3,153],[4,151],[5,145] \gg \\
O =& \ll [1,f,f,f],[2,t,f,t],[3,t,f,t],[4,t,f,f] \gg
\end{aligned}$$

6.3.4 Declarative Provenance Semantics

We now present the contribution semantics (definition of provenance) applied by *Ariadne* that models the provenance of an output tuple t of a query q as a provenance set, the set of tuples from the input stream(s) of q that were used to derive t. The declarative definition of our contribution semantics captures assumptions about provenance that one would intuitively expect to hold. (i) The provenance of a tuple should produce this tuple and nothing else. (ii) Provenance should not include tuples that did not contribute to the output. These intuitions are captured by stating conditions over the result of evaluating a query network over subsets of its input streams. E.g., by removing all tuples from the input that do not belong to the provenance set of an output tuple.

To be able to state such conditions we define two types of reduced input streams: Intersection of a stream I with a set $\mathcal{M}$ is denoted by $I \cap \mathcal{M}$ and defined as:

$$I \cap \mathcal{M} = \begin{cases} H(I) \mid\mid T(I) \cap \mathcal{M} & \text{if } H(I) \in \mathcal{M} \\ T(I) \cap \mathcal{M} & \text{else} \end{cases}$$

The prefix $I \uparrow \mathcal{M}$ of a stream I according to a set $\mathcal{M}$ contains all tuples from the stream until the last (in order of the stream) tuple from $\mathcal{M}$:

$$I \uparrow M = \begin{cases} H(I) \mid\mid T(I) \uparrow M & \text{if } \exists t \in M : p_I(H(I)) \leq p_I(t) \\ T(I) \uparrow M & \text{else} \end{cases}$$

Prefix and intersection of a list of streams $\mathbb{I} = (I_1, \dots, I_n)$ with a set are defined as the list generated by applying prefix respective intersection to each stream in the list:

$$\mathbb{I} \cap \mathcal{M} = (I_1 \cap \mathcal{M}, \dots, I_n \cap \mathcal{M})$$
$$\mathbb{I} \uparrow \mathcal{M} = (I_1 \uparrow \mathcal{M}, \dots, I_n \uparrow \mathcal{M})$$

For an output stream O of a query network q we define q_O as the query network that contains all nodes that are reachable from O if we reverse the edges in q. I.e. the sub-network that contains only streams and operators that may influence the evaluation of O. Furthermore, $I \subseteq I'$ denotes that all items from I are contained in I'. Having defined prefix, intersection, and q_O we now present our declarative provenance definition.

Definition 2 (Provenance Sets) *The Provenance set $\mathcal{P}(q, \mathbb{I}, t)$ of a result tuple t from an output stream O of a stream algebra expression q over a list of input streams $\mathbb{I} = (I_1, \dots, I_n)$ is the minimal subset of all tuples from $\mathbb{I}$ ($Set(\mathbb{I})$) that fulfills the following conditions:*

$$q(\mathbb{I} \cap \mathcal{P}(q, \mathbb{I}, t))[O] = \ll x \gg \textit{ with } t.\mathcal{N} = x.\mathcal{N} \tag{1}$$
$$q(\mathbb{I} \uparrow P(q, \mathbb{I}, t))[O] = \ll \dots, t \gg \tag{2}$$
$$\forall \omega \in q_O : \omega(\mathbb{I} \cap P(q, \mathbb{I}, t)) \subseteq \omega(\mathbb{I}) \tag{3}$$

The conditions of Def. 2 capture the intuitive assumptions presented beforehand. Condition 1 guarantees that the provenance of t is sufficient for producing t and only produces t. This is done by evaluating q over the provenance, checking that only a single tuple x with the same attribute values as t is returned (the TID of x may be different from the one of t, because the operators use *new*). The second assumption of $\mathcal{P}$ to be minimal is expressed by Conditions 2 and 3. Conditions 2 checks that $\mathcal{P}(q, \mathbb{I}, t)$ is the provenance of tuple t and not of some other output tuple with the same attribute values at a different position in the stream. This is achieved by applying the query to input streams prefixes up to the last tuple in the provenance, and checking that the last output tuple in the output stream O is t (With the same $\mathcal{T}$ value, since TID assignment is same for replays). Condition 3 plays the same role for operators with windowing. It requires that replaying the provenance does only produce windows that are produced by the original evaluation of q. The interested reader can verify that conditions 2 and 3 are necessary on the following example:

$$\begin{aligned} \alpha_{sum(a),\mathcal{C}(2,1)}(&\ll [\mathcal{T}_1, 5], [\mathcal{T}_2, 5], [\mathcal{T}_3, 5], [\mathcal{T}_4, 5] \gg) \\ &= \ll [\mathcal{T}_5, 10], [\mathcal{T}_6, 10], [\mathcal{T}_7, 10] \gg \end{aligned}$$

Based on Def. 2 we define the *PEO* of q, a stream that contains original output tuples of q and their provenance.

Definition 3 (Provenance Enhanced Output Stream (PEO)) *For a query q over inputs $\mathbb{I}$ and an output stream O of q, the PEO $P(q, \mathbb{I}, O)$ is a stream with schema $[\mathcal{O}, \mathcal{P} : Set(\mathbb{I})]$ defined as:*

$$P(q, \mathbb{I}, O) = [H(O), \mathcal{P}(op, \mathbb{I}, H(O))] \mid\mid P(q, \mathbb{I}, T(O))$$

Example 3 *As an example for the provenance of a query network, reconsider the network q_{ex} from Example 2. Applying Def. 3 we generate the PEO for q_{ex} as presented below. To shorten the representation, we use the notation $t_{i:j}$ to refer of the jth tuple of input stream i.*

$$\begin{aligned} P(q) = \{&[1, f, f, f, \{t_{1:1}, t_{1:2}, t_{2:1}, t_{2:2}, t_{3:1}, t_{3:2}\}], \\ &[1, f, f, t, \{t_{1:2}, t_{1:3}, t_{2:2}, t_{2:3}, t_{3:2}, t_{3:3}\}], \\ &[1, f, f, t, \{t_{1:3}, t_{1:4}, t_{2:3}, t_{2:4}, t_{3:3}, t_{3:4}\}], \\ &[1, t, f, f, \{t_{1:4}, t_{1:5}, t_{2:4}, t_{2:5}, t_{3:4}, t_{3:5}\}]\} \end{aligned}$$

For instance, the provenance set of the first output tuple $o_1 = [1, f, f, f]$ of the network contains the first two tuples from all temperature readers. Feeding these tuples into the query network causes the windowing operators to generate the first windows as in the original query network. Thus, the result of the aggregation over these windows are the original first aggregation results. In consequence, the projections and joins generate the first original tuple o_1 and condition 1 of Def. 2 is fulfilled. Condition 2 holds too, because the tuples in provenance set of o_1 are the first tuples of the input streams, and, therefore replaying the input stream until these tuples generates the same result. The windows generated over the provenance are a subset of the windows generated by the complete input streams. In summary, all conditions of Def. 2 are fulfilled.

6.3.5 Provenance Generating Operators

The naive approach to compute a PEO would be to generate the provenance set for each tuple t after Def. 2 by enumerating all subsets of $Set(\mathbb{I})$ and for each one test if the conditions of the definition hold. This method is clearly not efficient enough to be applied in a DSMS. Hence, we developed new algebra operators that generate PEOs in a much more efficient manner, further optimizations are shown in Sec. 6.4.

Provenance Generators

$$\sigma_c^{PG}(I) = \begin{cases} [new, H(I).\mathcal{N}, \boldsymbol{\{H(I)\}}] \mid\mid \sigma_c^{PG}(T(I)) & \text{if } H(I) \models c \\ \sigma_c^{PG}(T(I)) & \text{else} \end{cases}$$

$$\pi_A^{PG}(I) = [new, H(I).A, \boldsymbol{\{H(I)\}}] \mid\mid \pi_A^{PG}(T(I))$$

$$\alpha_{agg,\omega}^{PG}(I) = pa(\omega(I))$$
$$pa(I) = [new, agg_1(a_1, H(I)), \ldots, agg_n(a_n, H(I)), \boldsymbol{Set(H(I))}] \mid\mid pa(T(I)$$

$$\bowtie_{c,j\omega}^{PG}(I, I') = pj(H(j\omega(I, I'))) \mid\rightarrow pj(T(j\omega(I, I')))$$
$$pj(jw) = pjl([\ll H(jw.w_1) \gg, jw.w_2]) \mid\rightarrow pj([T(jw.w_1), jw.w_2])$$
$$pjl(jw) = \begin{cases} [new, H(jw.w_1).\mathcal{N}, H(jw.w_2).\mathcal{N}, & \\ \boldsymbol{\{H(jw.w_1), H(jw.w_2)\}}] \mid\mid pjl([jw.w_1, T(jw.w_2)]) & \text{if } [H(jw.w_1), H(jw.w_2)] \models c \\ & \\ pjl([jw.w_1, T(jw.w_2)]) & \text{else} \end{cases}$$

Figure 6.3: *Provenance Generator Operator Types*

For each operator of our stream algebra we introduce two new versions that generate PEOs. (1) A *Provenance Generator* (*PG*) is an operator that generates the PEO for a single algebra operator *op* from the input stream of *op*. This class of operators is used to generate PEO for operators in a query network that access solely input stream of the network. (2) A *Provenance Propagator* (*PP*) propagates from a PEO of the input(s) of an operator *op* to generate the PEO of the output of *op*. *PP* class operators are used to generate the PEO for an intermediate node in a query network by passing on provenance information from the input PEOs of an operator that have been generated by *PG* or another *PP* operator. This means, *PG* and *PP* operators are used in combination to derive the PEO for a query network output by transforming q into what we call a provenance generating network $PGN(q)$:

1. Replace each operator *op* that accesses only input streams of q by its PG version.

2. For joins that access both an input and an intermediate stream, wrap the input stream by adding a PG projections on all attributes of the input stream's schema.

3. Replace all remaining operators in q by their PP version

Provenance Propagators

$$\sigma_c^{PP}(P) = \begin{cases} [new, H(P).\mathcal{N}, \boldsymbol{H(P).\mathcal{P}}] \mid\mid \sigma_c^{PP}(T(P)) & \text{if } H(P).\mathcal{N} \models c \\ \sigma_c^{PP}(T(P)) & \text{else} \end{cases}$$

$$\pi_A^{PP}(I) = [new, H(I).A, \boldsymbol{H(I).\mathcal{P}}] \mid\mid \pi_A^{PP}(T(I))$$

$$\alpha_{agg,\omega}^{PP}(I) = pa(\omega(I))$$
$$pa(I) = [new, agg_1(a_1, H(I)), \ldots, agg_n(a_n, H(I)), \bigcup_{\boldsymbol{i \in H(I)}} \boldsymbol{i.\mathcal{P}}] \mid\mid pa(T(I))$$

$$\bowtie_{c,j\omega}^{PP}(I, I') = pj(H(j\omega(I, I'))) \mid\rightarrow pj(T(j\omega(I, I')))$$
$$pj(jw) = pjl([\ll H(jw.w_1) \gg, jw.w_2]) \mid\rightarrow pj([T(jw.w_1), jw.w_2])$$
$$pjl(jw) = \begin{cases} [new, H(jw.w_1).\mathcal{N}, H(jw.w_2).\mathcal{N}, & \\ \boldsymbol{H(jw.w_1).\mathcal{P}} \cup \boldsymbol{H(jw.w_2).\mathcal{P}}] \mid\mid pjl([jw.w_1, T(jw.w_2)]) & \text{if } [H(jw.w_1), H(jw.w_2)] \models c \\ & \\ pjl([jw.w_1, T(jw.w_2)]) & \text{else} \end{cases}$$

Figure 6.4: *Provenance Propagator Operator Types*

In the definition of the PG and PP operators we use $\mathcal{N}$ to refer to all attributes of a tuple except for TID and $\mathcal{P}$.

Definition 4 (PG and PP) *The PG and PP stream algebra operators are defined as presented in Figure6.3 and Figure6.4 respectively.*

We briefly explain two operators and their PG and PP semantics here: For selection, the PG provenance of a tuple t is the tuple itself, shown as $\{H(I)\}$, in PP mode the provenance of the selected tuple,$H(P).\mathcal{P}$. For aggregation, the provenance of t is the set of all tuples in the window that generated t (in PG), respectively the union of their provenance (in PP).

Example 4 *Reconsider the query network q_{ex} from example 3. Applying the approach presented above to transform a network into its corresponding PGN the aggregations in q_i are replaced by PG operators, because they are the only operators in q_{ex} that directly access input streams. Consequently all other operators in q_{ex} are replaced by their PP version. As an example of how PEOs are generated by these operators we discuss the PEO generation in sub network q_1 in more detail. Note that we omit the tid values of tuples for simplicity.*

The PG aggregation $q_{agg} = \alpha^{PG}_{avg(temp)\rightarrow avgtemp, min(time)\rightarrow time, \mathcal{V}(time,2,1)}(T_1)$ *in* q_1 *computes the original aggregation results by using the windowing operator to compute the average temperature over two consequent tuple from* T_1*. The tuples from each window* w *are then attached to the result of the aggregation over this window. All tuples from a window belong to the provenance set of the tuples from an aggregation result* t*, because they generate* t *(condition 1), they are necessary to generate* w *(condition 3), and using the contents of a different combination of input tuples that generate the same aggregation result would violate condition (2). Below we present the result of evaluating* q_{agg}*.*

$$T_1 = \ll [1,70],[2,166],[3,143],[4,161],[5,170] \gg$$
$$q_{agg} = \ll [1,118,\{t_{1:1},t_{1:2}\}],[2,154.5,\{t_{1:2},t_{1:3}\}],\ldots \gg$$

The projection in $PNG(q_1)$ *that processes the output of the aggregation is a PP operator. It projects the PEO input tuples on the expressions from* A *and simply passes on the provenance sets of its input PEO. For an output* o *of a projection, the provenance set of tuple* i *that was projected on* o *clearly fulfills condition 1. It is also minimal and fulfills conditions 2 and 3, because it is attached to* i *in the PEO of the input stream of the projection.*

$$\pi^{PP}_{avgtemp>150\rightarrow oh_i, time\rightarrow t_i}(q_{agg})$$
$$= \ll [f,1,\{t_{1:1},t_{1:2}\}],[t,2,\{t_{1:2},t_{1:3}\}],\ldots \gg$$

Note that, by modelling provenance generation as two operator types, one for initial provenance generation and one for provenance propagation, we have the necessary means to compute provenance for only parts of a query network by deciding which operator is the first in a path to be replaced by its PG version (we refer to this operator as being a *p-hook*). Furthermore, the output of each PP operator *op* is the provenance of the sub-network between all *p-hooks* and *op*. We call operators that generate the PEO for a part of a network *p-sinks*.

6.4 Implementation

6.4.1 Overview

We now present the implementation of the formalism developed in the last section in our prototype system *Ariadne*. While the formal description gives us the means to correctly describe the provenance for a given query network and data stream, several problems had to be overcome in the implementation to achieve efficient computation and retrieval of provenance.

Based on the analysis we presented in Section 6.2, we have made the following decisions in the implementation of Ariadne: 1) internally, represent provenance as a set of tuple identifiers (*TID-Set*), 2) use *operator instrumentation* for provenance computation, and 3) use the original data model for external provenance representation. This combination provides both flexibility and efficiency in provenance operations. Using TIDs avoids the cost of propagating full tuples and enable further optimizations which we present in Section 6.5.Operator instrumentation extends the implementation of each operator by introducing two new operational modes in addition to the normal operation of the operator (called NO): the PG and PP modes as presented in Section 6.3.5. A query network q is set-up for provenance computation by setting the operational modes according to the $PNG(q)$, thus providing the flexibility to (re-)configure provenance computation at a fine-grained level. Since we propagate TID-Sets from a *p-hooks* until the end of the query network, we can easily represent source provenance. If incremental provenance between operators is needed (e.g. for debugging as in [75]), we can also accommodate this by placing *p-hooks* and PG operators at each stage.

Representing provenance to users by duplicating the data tuples is simpler and more general than an extended data model, and does not prohibit us from adding specialized provenance query operators in the future. In order to generate this external format from the internal TID-Set representation, we added a new operator (called *expand*) which produces several duplicates from each output tuple t and its attached TID-Set. To restore the complete input tuples for provenance retrieval, this TID information can joined with the original input tuples stored at the *p-hooks*.

Ariadne is an extension of the Borealis data stream management system [15], a distributed stream processing engine that is based on *Aurora* [16]. In Borealis, a query network is represented as a DAG. The nodes in this DAG represent the operators (a.k.a. boxes) and a directed edge from box x to box y indicates that the output of x is used as the input of y. Boxes operate on constant-size tuples with a fixed schema. The input streams of a box are buffered in so-called tuple queues. A box x dequeues tuples from its input queues and enqueues its output tuples to the queue(s) of its downstream boxes. We use the *connection points* functionality [15] of Borealis to store original tuples at *p-hooks*.

6.4.2 Representing and Propagating Provenance

Borealis uses queues to pass fixed-length tuples as uninterpreted chunks of memory between operators in the query network. We considered three alternatives to pass the variable-size TID-Sets between operators: (i) Modify the queuing mechanism to deal with variable-length tuples, (ii) Propagate TID-Sets through channels other than normal tuple queues, or (iii) Split large TID-Sets into fixed-length chunks which are then streamed over normal queues. We chose the third approach, because it is less intrusive than changing all code that depends on fixed-length or introducing a new information passing mechanism. Furthermore, we benefit from several optimizations in the engine that rely on tuples of fixed-size.

We serialize a TID-Set of a tuple t into a list of tuples, each storing multiple TIDs from the set, that are emitted directly after t. The very first tuple in this list has a small header that stores the number of TIDs in the set. This header is used by down-stream operators to determine how many provenance tuples have to be dequeued. Given that a TID in Borealis is 8 bytes and the tuple header is almost 90 bytes, we save around an order of magnitude of space (and number of tuples propagated) compared to using full tuples even if tuples have very small payloads. The actual savings depend on number and type of payload data fields in the regular tuples: If TS is the size of the tuples in a queue and $TIDS$ is the size of a TID value, then we store $TS/TIDS$ TIDs in each provenance tuple except for the first one, which stores $(TS - PHeader)/TIDS$ TIDs, where $PHeader$ is the size of the provenance header (explained later). Despite these savings, the TID-Set representation can still cause significant overhead due to the large amount of provenance that operations such as aggregation can create. We investigate compression methods to further reduce this overhead in Section 6.5.

6.4.3 Provenance Operator Modes

As outlined in the beginning of this sections, we extend the existing Borealis operators with modes to consume and propagate provenance. This extensions reflects the formal semantics, adding the following modes:

- Initially produce provenance (PG)
- Consume data with existing provenance, combine with newly generator provenance (PP).

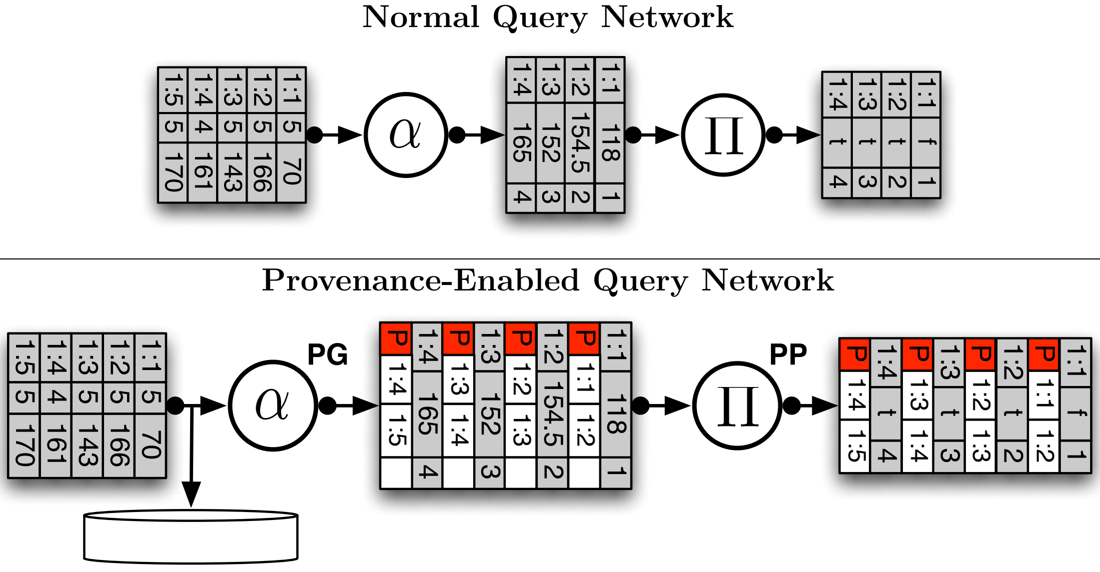


Figure 6.5: *Example for Provenance Computation*

An operator in PG-mode generates the TID-Set for each output tuple t and serializes it into additional tuples that are emitted directly after t. An operator in PP-mode processes each input tuple t by dequeuing t and all of the provenance tuples following t. Input tuple provenance is buffered until an output tuple is generated. All operators require similar functionality in terms of provenance handling. Therefore, we factored out common functionality such as dequeuing, enqueuing, and merging of TID-Sets to limit the changes to the original operator implementations.

6.4.4 The Expand Operator

As mentioned before, we enable provenance retrieval by providing an expand operator that transforms an output tuple t with an attached TID-Set into duplicates of t with a single TID attached to each duplicate. The expand operator simply reads all provenance tuples for an input tuple and repeatedly outputs copies of this tuple with an attached TID from the provenance TID-Set.

6.4.5 Input Tuple Storage and Retrieval

Borealis supports the storage of tuples from a queue in so-called *connection points* (CPs). CPs can be configured to preserve tuples for a given time interval. We use CPs to realize both preservation of input tuples and of output tuples with provenance (if required).

Example 5 *Fig. 6.5 presents the partial query network q_1 from our running example. The basic processing applied by this network is illustrated on the top of the figure: circles represent operators and the stacked squares represent queues (each tuple is one square). For convenience, each queue is shown with all the tuples that have flown through it. The same network is instrumented for provenance computation is shown at the bottom of the figure. The aggregation operates in PG-mode and the projection operates in PP-mode. The white tuples are the provenance tuples generated by the instrumented network. The red P, represents the provenance header. For example, the second tuple in the output of the aggregation stores the TID-Set* $\{1:1, 1:2\}$ *of the first output tuple of the aggregation. Note that in this example we assumed that both TIDs barely fitting into one provenance tuple. In reality, we can store many more TIDs in a single tuple. Note that a CP (the cylinder) is used to preserve the T_1 tuples for provenance retrieval.*

6.5 Optimizations

By its very definition, fine-grained provenance is potentially significantly larger than the result generated by normal query processing. This additional data can cause significant cost in computation and storage, as observed in many database [51] or workflow provenance systems. Providing provenance for data streams systems further aggregates this problem, mainly for two reasons:

1. Typical DSMS workloads rely heavily on aggregation to reduce downstream workload, reducing many input tuples to few output tuples. When requesting provenance for such aggregation queries, these saving are negated, as all input tuples are part of the provenance.

2. DSMSs treat data as transient, computing result on the fly and discarding data as soon as possible as to keep up with high data arrival rates. Consequently, the computation of provenance has to also be performed on the fly, possibly wasting significant resources if this provenance is never requested.

Given these challenges, we focus our optimizations on *compressing* provenance (as to reduce the overhead of computation) as well enabling on-demand provenance operations (as to avoid unnecessary provenance operations).

6.5.1 Provenance Compression

TID-Sets are already a quite compact representation of provenance information. They are efficiently serialized and propagated by our operator implementations. However, many workloads, in particular those involving large window-based aggregates lead to large TID-Sets, putting a lot of additional computation, storage overhead and queue operations to downstream operators. We study a number of methods for efficient TID-Set compression, ranging from generic data compression to methods which exploit data model and operator characteristics. An important consideration is the balance between the cost needed to perform compression/decompression operations and the savings achieved. In particular, choosing compression methods that do not require decompression for most operators will be important.

As we will see, these compressions work best under certain workload characteristics, necessitating an adaptive combination of them to provide effective processing for an arbitrary and changing workload.

Interval Encoding

Interval Encoding exploits the fact that windows produce contiguous sub-sequences of its ordered input sequences. Instead of storing each element of a TID-Set individually, it represent them as a list of intervals spanning the uninterrupted sequences of TIDs in the set. Under interval encoding, operators in PG-mode group contiguous TIDs into a single interval; operators in PP-mode try to merge intervals they read from their input. The merging can be efficiently done using existing well-known interval merging techniques [57].)

Interval encoding is most advantageous for queries involving aggregations over a long sequence of contiguous TIDs that are represented as a single interval. The worst-case takes place when each interval represents only a single TID (i.e., no aggregation operator). In practice, complex query networks which mix aggregations and other operators limit the applicability of this simple form interval encoding. As a result, combinations of interval encoding with other methods become more promising. Yet, interval encoding is an important foundation for lazy provenance computation, as we will show in Section 6.5.2.

Delta-Compression

Delta compression exploits the fact that windows with small slide sizes overlap to a large extent. Therefore, the TID-Set of a tuple may be encoded more efficiently by representing it as some delta to the TID-Set of its predecessor (by encoding which TIDs at the start of the previous set are left out and which TIDs are appended to the end). More precisely, we represent a delta tuples as difference to the last full tuple, not the previous delta tuple. While this approach has a potentially higher space overhead, it simplifies operations like filter and also prevents errors.

Delta compression requires additional buffering to keep the TID-Sets of past tuples to be able to reconstruct the TID-Set of the current one. Moreover, since it can create unnecessary overhead, there are a handful of tunable parameters to ensure its efficiency. Delta compression is most effective with significantly overlapping windows, almost regardless of the presence of other operators. With correct settings of the parameters, the overhead of using on unsuitable workload (e.g. no overlap) will not be significant.

Dictionary Compression

Given their potential size, TID-Sets and also collections of intervals can be compressed using standard compression techniques. We used LZ77 as it deals with flexible input sizes (not a block-based compression) and provides a good tradeoff between speed and effectiveness, but other methods are certainly possible as well.

Compression is only activated if the size of a TID-Set or interval collection exceeds a fixed threshold to avoid paying the price of compression for a small number of values. If dictionary compressed applied, this is indicated in the provenance header send in the first tuple of the set. This type of compression can reduce the load on the queues significantly for large TID-Set at the cost of additional processing to compress and decompress the TID-Sets.

Since generic compression methods do not take advantage of the specifics of the data model and operator semantics, its benefit/cost potential is significantly worse than using methods like interval or delta encoding. As experiments show, significant space savings are possible, but the computational overhead is typically offsetting these savings.

6.5.2 On-Demand Provenance Operations

As outlined above, working on transient data streams leads to an eager way of dealing with provenance, as each computation needs to be performed while the data is still available. Since provenance computations are expensive and not all provenance is always needed, this is wasteful and limits the application areas of stream provenance significantly. As a result, we propose three orthogonal approaches with an increasing amount of complexity in order to delay provenance operations until they are really needed:

1. Partial Instrumentation of the Query Network
2. Lazy Retrieval
3. Lazy Computation

Partial Instrumentation of the Query Network

In particular for complex networks, always having full provenance on everything is not necessary. In particular, there is often no need to compute provenance

- end-to-end, as only a suffix of the plan is of interest
- for irrelevant subgraphs, as only certain paths are of interest
- all of the time, as only certain periods are relevant.

Our PGN approach outlined in Section 6.4 already provides all the means to perform partial instrumentation: Controllable Modes (PG, PP) of provenance-enabled operators provide the flexibility to enable/disable provenance computation when needed, and -as experiments show- the overhead of keeping the provenance infrastructure inactive is negligible. The main limiting factor at the moment are the connection points to store the original tuples, as they cannot be controlled in a similarly fine-grained manner. We therefore need to make some advance decision where to place.

Lazy Retrieval

The second approach to enable lazy operations is to delay provenance retrieval. In our PGN-based approach we separate the computation of the relevant TID from operations on the actual tuples. This is in contrast to rewrite-based approaches (see Section 6.2.1), which eagerly joins full tuples not only at the end of a query network, but at many stages. We therefore need to pay this cost of retrieval once, but also have the opportunity to not pay it if the provenance results are not needed. By doing so, the cost of the *expand* operation, the tuple lookup and the join can be saved.

Lazy Computation

The third and most complex approach is actually avoid computing provenance until it is really needed, as to bring down the overhead imposed on normal query processing.

Actually doing this brings us into a quandary: in order to compute something on a set of results after it had been produced requires provenance, which is what we try compute lazily. Since *Inversion* is not generally applicable (see Section 6.2.1, we instead opted for an approach that records minimal provenance eagerly. More precisely, we record a superset of the actual provenance and use it to determine which tuples to store and use for the actual provenance computation. This provenance computation is performed using a copy of the actual network dedicated solely for provenance computation. While this approach may seem wasteful, its separation of normal query processing (with minimal provenance) and full provenance computation allows us to scale out with multiple machines and keeping the performance impact on normal processing at a minimum.

As a representation of superset provenance that is easy to compute and does not include too many spurious provenance candidates, we settled for *Covering Intervals*. *Covering Intervals* are an extension of the *Interval Encoding* compression technique presented in Section 6.5.1. As shown there (and confirmed by the experiments), interval compression is very efficient when there are contiguous sequences of TIDs. In practice, intermittent operators like filters tend to destroy such contiguous sequences, thereby limiting the applicability when computing exact provenance. However, the compactness of representation and low computation cost are maintained if we ignore "holes" being "punched" in these sequences and represent the complete sequence, a.k.a. *Covering Interval*. The runtime overhead of such an approach is low for almost all workloads, since we only need to propagate intervals. Given that these intervals have constant sizes (only 2 TIDs), we do not need to use separate tuples for provenance, but can piggyback the intervals on regular tuples, thereby significantly reducing the number of tuples to be propagated.

Covering Intervals are also an effective way to create minimal supersets, since the ordered nature of the data model and the monotonic nature of operators prevent a massive growth beyond the actually needed data.

6.6 Experiments

The goal of our experimental evaluation is to investigate the overhead of provenance computation with *Ariadne* and analyze the impact of the TID-Set compression methods on the performance and memory consumption.

6.6.1 Overview

In the following we explain the noteworthy aspects of our experimental study.

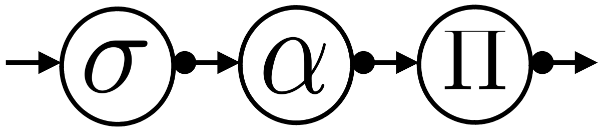


Figure 6.6: *Query Used in the Experiments*

Query

Our experiments are centered around the overhead of provenance generation when running a stream query depicted in Figure 6.6. This query is in essence q_1, our running example's query (from Section 6.1.3), preceded by a filter operator. In terms of functionality, this filter can be seen as an outlier remover. However, it serves a more crucial purpose in our experiments: to control the contiguity of TIDs flowing through the query network.

Performance Measures

As measures of overhead, we settled for the followings:

- **Completion Time**. It is defined as the difference between the arrival timestamp of the first input tuple and the leaving timestamp of the last output tuple. We measure completion time by utilizing the Borealis run-time statistics system: tuples receive a timestamp at arrival on the server, and a second timestamp when leaving the server, yielding a well-defined result for non-blocking operators. For aggregations and joins, we use timestamp of the last contributing tuple in the window, which is the default of Borealis.
- **Memory Consumption**. Another important resource to measure in provenance computation is the amount memory consumed by the system. There are two types of memory consumption in Ariadne:
  - *Tuple Queues*: each operator in Ariadne has a tuple queue per input stream. The active length of queues in runtime depends on the load, query, complexity, and amount of provenance data flowing through the query network. Consequently, measuring the average size of queues is an important indicator for memory overhead.
  - *Provenance Structure*: every provenance representation technique uses a different structure (internal to individual operators) to generate/propagate provenance. The amount of memory occupied by such provenance structures is the the second part of our memory consumption measurements.

To collect memory numbers, we designated an extra thread to sum up the sizes of all queues and provenance structures (in term of number of bytes) every 100 milliseconds and output the average at the end of execution.

Compared Methods

In all experiments, besides the *normal* (baseline, no provenance) and *Covering Interval* methods, denoted by 'noProvenance' and'coverInterval' respectively in the graphs, we consider various combinations of the following optimization techniques:

- the *naive* TID propagation method (denoted by 'single')
- the *interval* representation of TIDs (denoted by 'interval')
- the *adaptive* representation which decides between 'single' and 'interval' on the fly (denoted by 'adaptive')
- the *shared buffers* technique (denoted by 'buffer')
- the *delta compression* technique (denoted by 'delta')
- the *dictionary compression* technique (denoted by 'compress')

Setup

All experiments were run on a system with four Intel Xeon L5520 2.26 Ghz quad-core CPUs, 24GB RAM, running Ubuntu Linux 10.04 64 bit. Both the client (load generator) and the server are placed on the same machine and the client sends its input in very large batch sizes (100K tuples). We used this setup, because preliminary experiments indicated that the code that is responsible for retrieving inputs from the client and enqueuing them into the query network is a major performance bottleneck. In fact, the maximum load that can be handled by the retriever part is too low to place significant stress on actual query execution and, thus, covered up the overhead introduced by provenance computation. Sending a single large batch guarantees that query execution runs at maximum throughput which is needed to measure the "worst-case" overhead of provenance computation. Additionally, we ran separate experiments to tune the parameters of Ariadne/Borealis and our compression techniques.

Since the overhead of unused provenance code turned out to be negligible, we used *Ariadne* for both baseline (no provenance) and provenance experiments. Each experiment was repeated 10 times to minimize the impact of random effects.

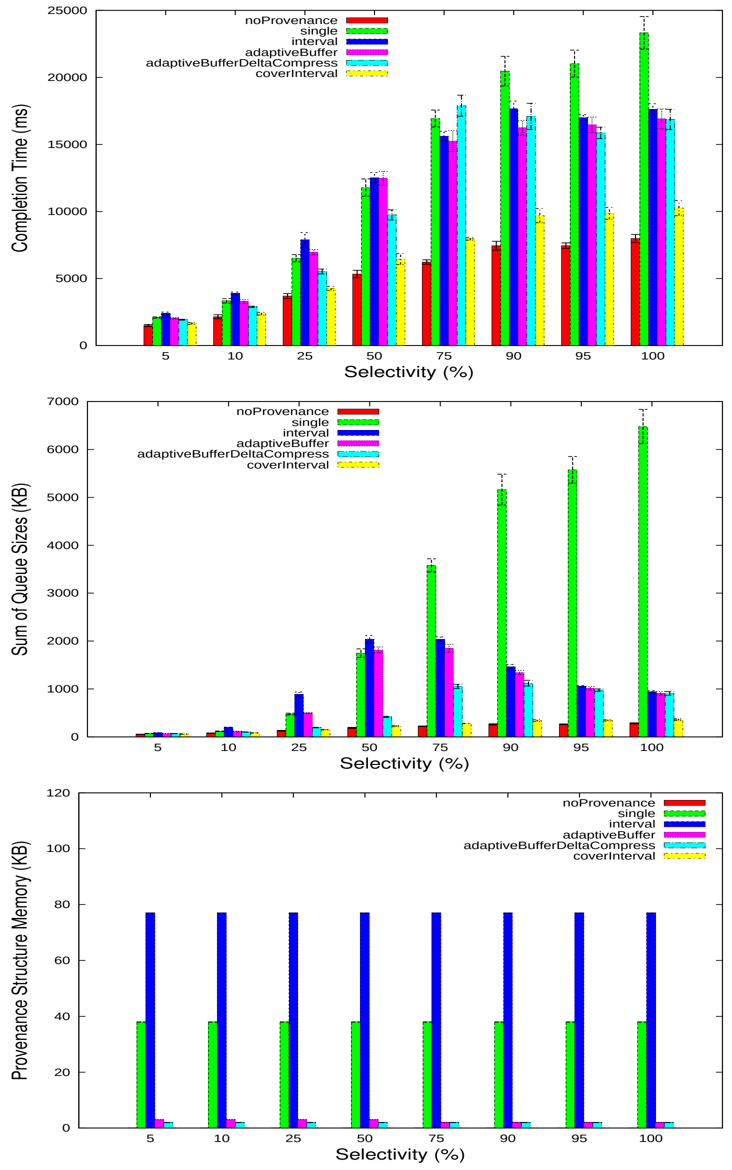


Figure 6.7: *Impact of Selectivity (Window Size = 100, Slide Size = 1)*

Experiments Plan

We investigated the important factors in driving the cost: selectivity, overlap, and window size. Below, we present and discuss the results.

6.6.2 Impact of Selectivity

The workload distribution directly affects the cost of TID-Set propagation. For example, in the extreme case of transparent filters (100% selectivity), all tuples in a window are contiguous and therefore only a pair of values (an interval) in sufficient to keep track of a TID-Set. In the first set of experiments, we varied the filter selectivity between 5% and 100%. Figure 6.7 shows the results.

For the baseline and naive propagation approach (single TIDs), the completion time consistently increases with the higher selectivity values because the load (i.e., number of generated output tuples) in the system grows constantly. For low selectivity filters, the gain of interval encoding is not much; in fact for the very low selectivity values it performs worse than single, because there are not enough contiguous TID-Sets to compensate for the extra TIDs that interval encoding needs (compared to single).

In terms of queue sizes, the naive approach (single TIDs) results in bigger queues for higher selectivity values while other methods reach their maximums at medium selectivity values (between 50% and 75%), where the load level is relatively high but TID contiguities are not yet sufficiently big.

Finally, the amount of memory occupied by the provenance structure is zero for normal (obvious) and covering interval (it piggybacks the required two values into the tuple header). Single and interval have constant and large values, due to the fixed number of TIDs they always maintain. Other methods have far lower memory consumption.

6.6.3 Impact of Overlap

When increasing the window slide size from 1 to 100 at a window size of 100, we observe that the completion time and the queue sizes consistently decrease (see Figure 6.8). The overall drop in all methods is due to the sharp decrease in the load level of the system (for instance, compared to slide size 1, slide size 2 produces 50% less number of output tuples). The logarithmic fashion of these two graphs can be explained by the fact that after slide size 10, the system is not stressed anymore and therefore the impact of provenance computation is not noticeable anymore.

The amount of memory consumed by provenance structures in the single and interval representation modes drop linearly as the number of open windows decreases for bigger slide sizes. Lastly, since big slide sizes result in very limited overlaps between open windows, they demonstrate the worst-case scenario for the buffer and delta techniques. In other words, maintaining those complex data structures does not pay back and they perform worse than the naive approach (single).

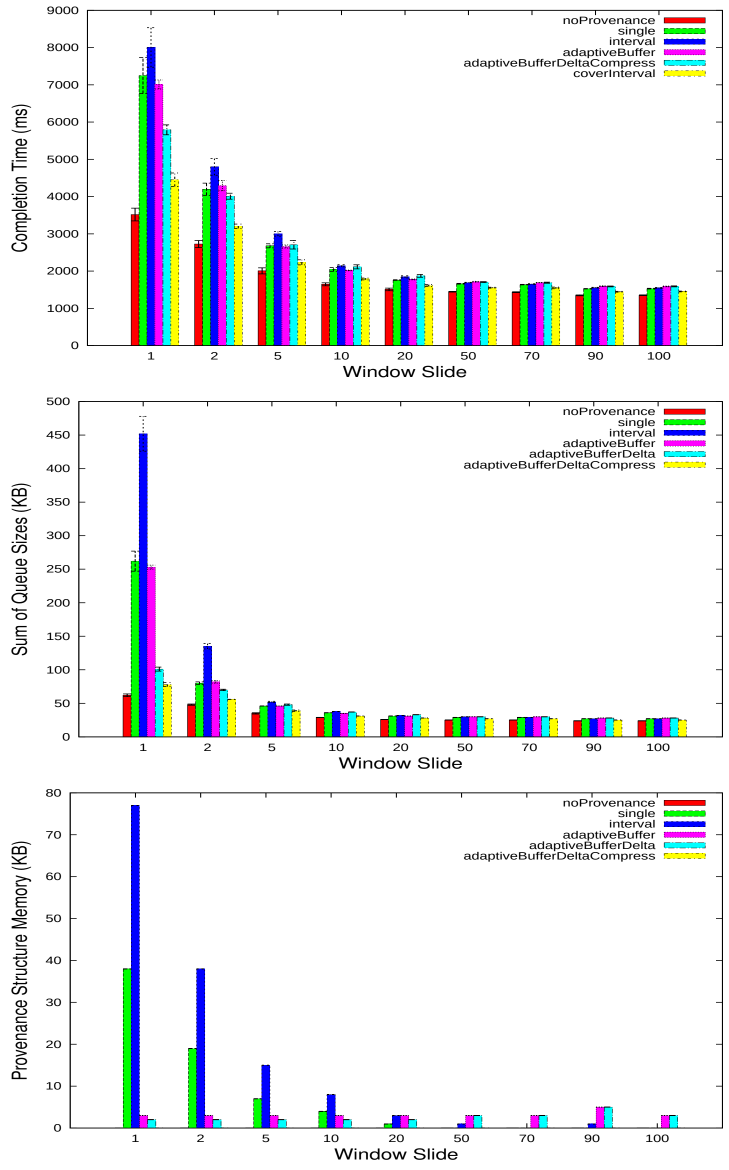


Figure 6.8: *Impact of Overlap (Selectivity = 25%, Window Size = 100)*

6.6.4 Impact of Window Size

We expect an increase of provenance cost when increasing the window size on the TID-Set implementation. Figure 6.9, varying window size from 50 to 2000 while keeping slide at 1, clearly shows this effect. Although for all methods the cost grows with the increased window sizes, but it is more pronounced in the big window sizes (i.e. larger than 200).

With regard to the queue sizes, the impact of the dictionary compression technique is quite visible in this set of experiments (in contrast to the previous two). In fact, the amount of provenance data produced by large windows exceeds the compression threshold

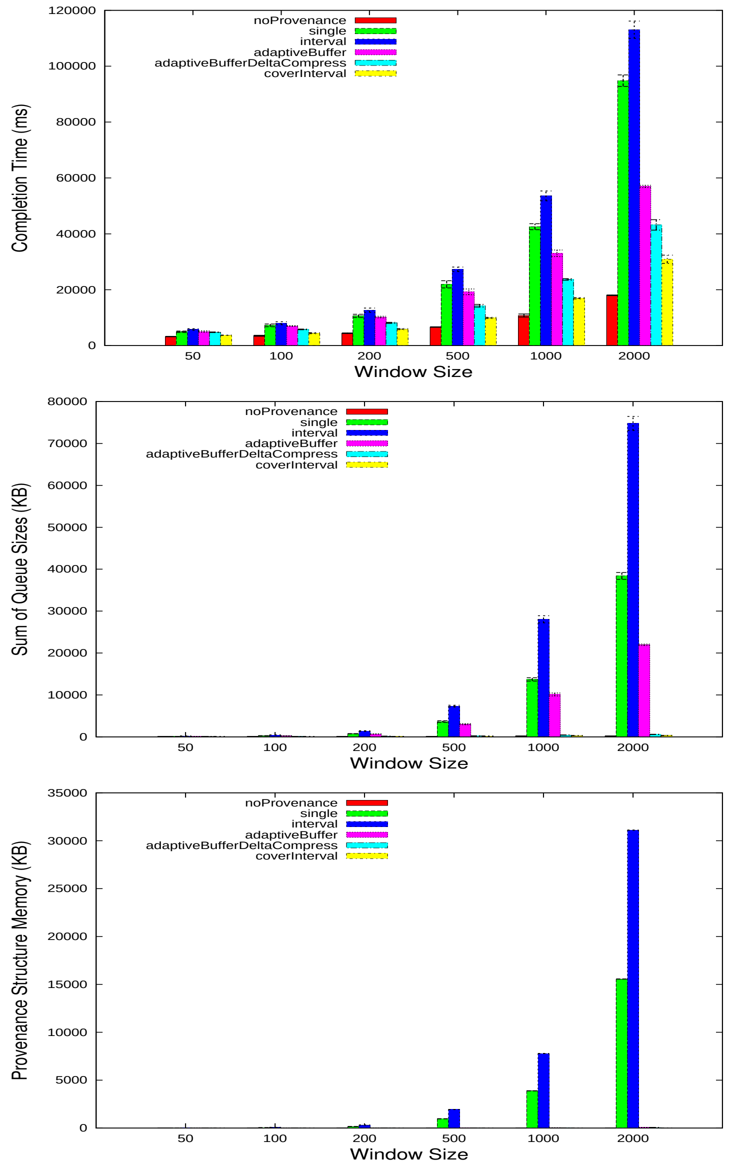


Figure 6.9: *Impact of Window Size (Selectivity = 25%, Slide = 1)*

and triggers the dictionary compression, resulting in considerably smaller queues.

6.6.5 Recording Cost for Lazy Computation

In all of the experiments presented so far, the cost of recording a minimum amount information for lazy provenance computation (through *replay*) is promisingly small. In fact, it increases the completion time by less than 50% in the worst case (within the scope of our experimental study) and has no memory overhead at all. As pointed out earlier in the chapter, this makes lazy provenance computation a favorable option in the environments that frequency of retrieving provenance information is low.

6.7 Related Work

Most of the research on provenance has been conducted by the workflow and database communities. [33] presents a survey of definitions of provenance (*contribution semantics*) for relational databases. Examples for provenance-enabled database systems are *Trio* [89], *Perm* [51], and *DBNotes* [26]. Most of the approaches for workflow provenance (with the exception of some recent work, e.g., [21])) handle tasks in the workflow as black-boxes, and, therefore consider all outputs of a task to depend on all inputs (see [39] for an survey of workflow provenance). Such provenance is not very helpful for the kind of use cases we are considering.

Huq et al [58] have proposed to achieve *fine-grained* stream provenance by augmenting *coarse-grained* provenance with timestamp-based data versioning, focusing specifically on query result reproducibility at reduced provenance metadata storage cost. Furthermore, a couple of recent workshop articles have discussed the need for low-overhead stream provenance modeling and collection for scientific stream processing [87][86]. In contrast to *Ariadne* these publications focus on coarse-grained provenance which not well suited for the use cases from section 6.1.1.

Wang et al. has proposed a model-based data provenance framework (namely, “Time-Value-Centric” (TVC) provenance) for medical sensor streams [88][73]. TVC describes provenance the provenance of an output tuple t as a set of rules that define which input tuples should included in the provenance as time intervals over input streams based on the timestamp of t. Each rule has a condition over the values of t or over the state of the operator that generates t. The provenance of t includes all time intervals from rules for which the condition is fulfilled on t. These rules have to be defined manually. Thus, the main difference to our work is that we focus on a set of operators with well-defined semantics and provide automated provenance generation for these operators without the need to define the contribution semantics of each operator manually for each query. Furthermore, it is unclear if TVC rules are powerful enough to deal with complex windowing functions efficiently. E.g., the number of rules necessary to describe the provenance of an output may grow to large to be a compact representation of provenance.

Stream provenances management shares some of its challenges and solutions with revision processing on streams as introduced by Cherniack et al. as part of the Borealis project [15][76]. In order to be able to process corrections to previously processed stream tuples, Borealis keeps archives of recent input data and replays portions of them. Provenance management, however, is more general, and can form a base mechanism for many applications including revision processing.

6.8 Conclusions

In this chapter, we have studied the problem of data provenance management in DSMSs. Based on a wide spectrum of use cases, we have identified that managing fine-grained, data item-level provenance information, is a fundamental requirement in DSMS and developed a contribution semantics for stream provenance. With *Ariadne* we have shown how stream provenance can be efficiently implemented and managed in a typical DSMS such as Borealis. Our experimental evaluation demonstrates that fine-grained provenance can be computed efficiently using operator instrumentation and optimized TID-Set representation It should be noted that although Borealis was used as a proof-of-concept platform, our techniques are general enough to be easily applied in other DSMSs as well.

In the following, we outline two interesting topics of ongoing and future work. First, we would like to enrich our experimental study by using more complex query networks e.g., queries with nested aggregations for which the size of provenance grow exponentially.

Second, we plan to empirically compare the performance of operator instrumentation and query rewrite methods. To this end, we can exploit the *connection points* of Borealis to store input tuples at the beginning of the query network. The stored tuples will be later retrieved and joined with the outputs of expand operator to produce the provenance-carrying output streams. Designing intelligent purging mechanisms for such storage points is an interesting research problem to solve.

Chapter 7

A Framework To Model Query Lifecycle Operations

7.1 Introduction

7.1.1 Motivation

In contrast to one-time queries in relational databases, continuous queries in SPEs can run for unpredictably long time periods over infinitely long data streams. During their lifetime, due to requirements of application semantics or resource constraints of the system, these queries may go through different stages in their lifetime. We call these stages, *query lifecycle states*. Transition between these states are triggered by lifecycle operations. Examples of such operations are query stop, start, pause/resume, migration, and modification.

Although there exists a general and informal understanding of these lifecycle operations, there is no formally-grounded and established agreement on their precise semantics, even for the most basic operations. As a matter of fact, there have been only few proposals on stopping and restarting of long-running queries in data warehouses [65, 30, 32], but these proposals, due to the fundamental differences between streams and traditional warehouses are not applicable to SPEs.

Furthermore, in case of complex lifecycle operations such as query modification, there can be multiple interpretations of what the desirable outcome should be. Exploring these variations and providing mechanisms to evaluate and rate them is another aspect of this problem.

In this chapter, we establish a framework that can formally express arbitrary lifecycle operations on the basis of input-output mappings and basic control elements (such as query start or query stop). In the next chapter, we will use this framework to present a query modification model.

7.1.2 Contributions

In this chapter, we have made the following contributions:

- propose a query lifecycle operation modeling framework. It has three main aspects: 1) uses input-output mapping functions to abstract away the query details; 2) formally define control elements and punctuate data streams with them to influence the behavior of queries; 3) introduces correctness criteria, a mechanism to evaluate the outcomes of different control elements and shows how they can be specified formally.
- propose a methodology that allows one to leverage our framework to model complex lifecycle operations.
- model query pause/resume using our framework and methodology.
- propose a general architecture to implement our framework.

7.1.3 Outline

The rest of this chapter is organized as follows. Section 7.2 introduces our framework and methodology. In Section 7.3 we model query pause/resume operation using this framework and methodology. In Section 7.4, we decrease the abstraction level and describe how our model can be refined from query-level to operator-level. There, we also investigate how real-life SPEs can be adapted to support our framework. This is followed by a general architecture for query lifecycle management in Section 7.5. After giving a summary of related work in Section 7.6, we conclude this chapter in Section 7.7.

7.2 Our Framework and Methodology

Our goal in this chapter is to create a framework which allows modeling query lifecycle operations. In order to achieve generality, precision, and deterministic behavior, this framework should not rely on semantics of specific operators, timing, or state, which are all notoriously hard to reason about (e.g., see [15] for an approach that encounters timing issues). To this end, our proposal abstracts away semantics and execution of queries into their data dependencies (using mapping functions introduced in Chapter 3) and annotate them with *punctuations* [84] which we call *Control Elements*. Control Elements are a special type of Stream Element (introduced in Chapter 3) that carries control metadata instead of regular data values. Control Elements do not directly take part in query processing, however, they convey important information to our framework regarding how the query should behave if encountering them.

Control Elements are punctuated into the input stream either by the user or by the system itself, depending on the use case. The order among data elements can be relaxed to a partial order (for models that process elements in groups, such as STREAM [74]), only control elements need to have a total order among themselves and with respect to data elements. For clarity of presentation, we assume total order among all *stream elements*. Moreover, within the scope of this chapter (and also next chapter) we primarily focus on queries with a single input and output stream, a restriction which we plan to remove in future work.

In the rest of this section, we describe the basic control elements, their interaction, correctness criteria, and finally our methodology to model complex control elements in a generic manner. Later, in Section 7.3, we use this framework and methodology to model query pause/resume operation. Chapter 8 will describe the complete execution of this methodology for query modification, another complex lifecycle operation.

7.2.1 Basic Control Elements

We establish a minimal set of basic control elements, which define basic lifecycle behavior and serve as building blocks for complex control elements. This set consists of control elements which take into the consideration the boundary conditions (namely, start and stop) of continuous queries. Understanding such conditions is crucial to understand many complex lifecycle operations such as query pause/resume, migration, modification[1].

In the following, we define the Start and the Stop Control Elements. We re-use our running example from Section 3.1 to give concrete examples. As depicted in Figure 7.1, $Q1$ uses a tuple-based sliding window of size 3 and slide 2, applying a sum operation on the window (Figure 7.1). Without limiting generality, we are using windows as a representative of stateful operators whose semantics exhibit non-trivial behavior on query lifecycle operations.

[1]It should be noted that this minimal set is not complete (meaning not all complex operations can be modeled using only this set). However, the extensibility of the framework allows adding more basic control elements to this set.

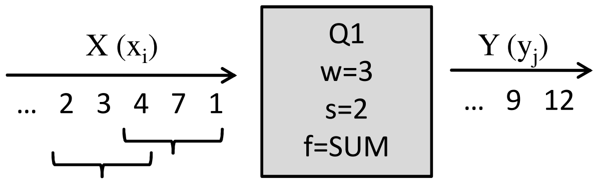


Figure 7.1: *Running example - Q1*

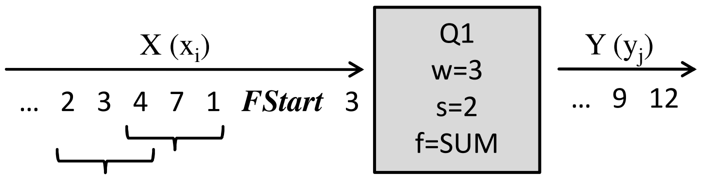


Figure 7.2: *Fresh Start on Q1*

Start

Upon encountering a start control element, a query Q will start producing output, otherwise the arriving inputs will be ignored. We draw distinction between three variations of Start elements:

- *Fresh Start (FStart)*: Assuming that X and Y represent the input and the output streams respectively, we denote this control element by x_{fstart} (and short **F** in figures), where $fstart$ is its position in the stream. Upon receiving an x_{fstart}, query Q will be started; i.e., the input data elements at positions after $fstart$ will contribute to the output. In formal terms, Y_{fstart} (the output stream for an input stream which contains x_{fstart}) is a substream of Y and defined as below:

  $$Y_{fstart} \;=\; \{y_j \;\in\; Y \mid \forall x_i \;\in\; \textit{depends}(y_j),\; i \;>\; fstart\;\}$$

  Figure 7.2 illustrates Fresh Start applied on $Q1$. It is important to note that a Fresh Start element restarts the *starting position* of the underlying query mapping functions (*depends(y_j)* and *contributes(x_i)*). As an example, applying Fresh Start after 1 instead of 3 in Figure 7.2 would shift the input sets of all windows by one position. This property makes Fresh Start unique among Basic Control Elements, because all other Basic Control Elements only *use* the mapping functions and do not modify them.

- *Cold Start (CStart)*: depicted as x_{cstart}, where *cstart* (short, **C**) denotes its index. Upon receiving x_{cstart} the query will start producing outputs which *exclusively* de-

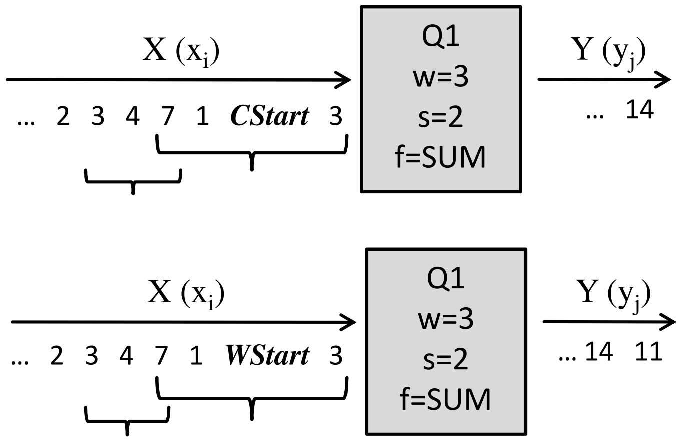


Figure 7.3: *Cold Start versus Warm Start*

pend on the input items arriving after x_{cstart}. In formal terms:

$$Y_{cstart} = \{y_j \in Y \mid \forall x_i \in \mathit{depends}(y_j),\ i > \mathit{cstart}\}$$

- *Warm Start (WStart)*: depicted as x_{wstart}, where *wstart* (short, **W**) denotes its index. Upon receiving x_{wstart} the query will start producing outputs with *full* or *partial* dependency on the input items arriving after x_{wstart}[2]. In formal terms:

$$Y_{wstart} = \{y_j \in Y \mid \exists x_i \in \mathit{depends}(y_j),\ i > \mathit{wstart}\}$$

Figure 7.3 illustrates the difference between the impacts of Cold Start and Warm Start Control Elements. Notice that both Cold Start and Warm Start Control Elements *use* the existing mapping functions and, in contrast to Fresh Start, do not *create* new ones.

Stop

Upon encountering a stop control element, a query Q will eventually stop producing output. We define two kinds of Stop elements:

- *Immediate Stop (IStop)*: We denote this control element with x_{istop} (short, **I**), where *istop* is its position in the stream. Upon receiving an x_{istop}, a query Q will be immediately stopped; this means that the input data elements having a position greater than *istop* will not contribute to the output. More formally:

[2]One can quickly deduce the fact that for a given query, if x_{cstart} and x_{wstart} are applied at the same position then Y_{wstart} always subsumes Y_{cstart}.

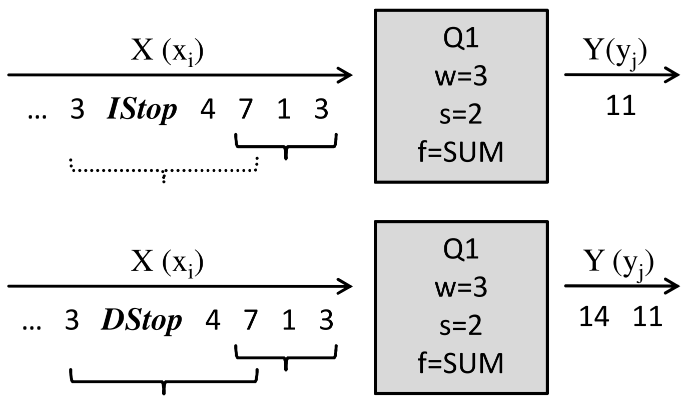


Figure 7.4: *Immediate Stop vs. Drain Stop on Q1*

$$Y_{istop} \;=\; \{y_j \;\in\; Y \mid \forall x_i \;\in\; depends(y_j),\; i \;<\; istop\;\}$$

- *Drain Stop (DStop)*: We denote this control element with x_{dstop} (short, **D**), where *dstop* is its position in the stream. Upon receiving an x_{dstop}, a query Q will be gradually stopped; meaning Q will continue to produce outputs which have dependencies on input data elements appearing before *dstop* and completing its partially produced output elements. More formally:

$$Y_{dstop} \;=\; \{y_j \;\in\; Y \mid \exists x_i \;\in\; depends(y_j),\; i \;<\; dstop\;\}$$

Figure 7.4 illustrates Immediate Stop and Drain Stop applied on $Q1$, showing that IStop discards the uncompleted window, whereas DStop finishes uncompleted windows, but does not open new ones and stops when there is no open window left. As we will show later in this chapter, both stop elements not only provide relevant semantics, but are also efficiently implementable on existing data stream systems.

7.2.2 Interaction of Basic Control Elements

A crucial part in defining the semantics of the query lifecycle operations is to understand the interaction of multiple control elements, both for the single query cases as well as for the interaction of multiple queries, which can be derived from the single query case. The interaction should maintain two design decisions established so far: (1) Control elements should become effective at the position they are specified, as defined in Section 7.2.1. (2) The order of control elements determines which control element is effective, superseding older ones.

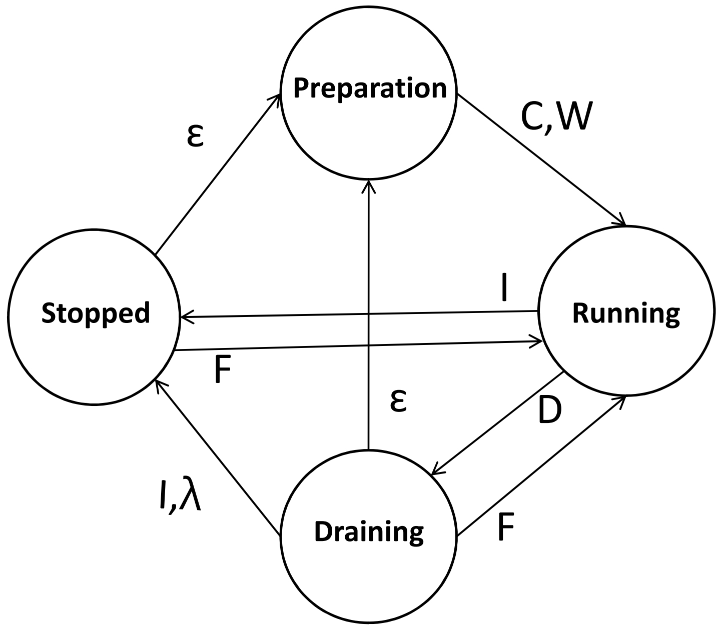


Figure 7.5: *Interaction of Basic Control Elements*

Figure 7.5 gives the states and the transitions, which correspond to the basic control elements. Initially, a query is in *Stopped* state, in which the *starting position* of the mapping function has not been set up. Using *FStart* (F), the mapping function is set up and the query transitions into the *Running* state, in which output is produced. An *IStop* (I) directly changes the state to the *Stopped* state, while an additional *Draining* state is needed for *DStop* (D) in which the pending output is produced. The transition from *Draining* to *Stopped* is not driven by any control element, but depends on data elements (denoted by λ). Conceptually, the λ-transition is similar to the notion of Timed Automata [20], yet not based on time, but on the progress of the underlying mapping function.

The *Preparation* state represents the cases in which the query keeps track of its mapping functions although it may not emit new outputs. This is needed to cater for control elements which need to "recall" the query history. Both *CStart* and *WStart* elements fall into this category. The ϵ-transitions in Figure 7.5 demonstrate the fact that as soon as the query goes into *Stopped* or *Draining* states it simultaneously goes to the *Preparation* state without consuming any data or control elements.

Finally, applying *FStart* in any state will re-initialize the mapping function and lead to the *Running* state, while applying *IStop* when *Draining* immediately stops the query.

7.2.3 Correctness Criteria

Having introduced the interaction diagram in the previous section, we saw that there can be more than one possibility to model basic lifecycle operators, let alone the more complex ones. Hence, we need to have an evaluation mechanism to compare and rate these variations. In our framework, this can be done by defining the required/desirable behavior of the target operation, through what we call *Correctness Criteria*.

Inspired by the approaches in system design and verification [65], we defined two general classes of Correctness criteria for lifecycle operations: *Safety* and *Liveness*.

Safety Criteria

A safety property expresses that "something bad will not happen" during a given execution [65]. We identified *Loss*, *Disorder*, and *Duplicates* as possible safety problems.

1. **Loss**: A common undesirable consequence of query lifecycle operations is losing some of the output elements. We formalize this concept as the set difference between *tuples* of the reference output stream (Y_{ref}) and those of the observed output stream (Y_{obs}), where Y_{ref} is the *ideal* output stream and Y_{obs} is the output stream that Q actually generates.

   We can formally express Loss as follows:

   $$Loss \ = \ [Y_{ref}] - [Y_{obs}]$$

   Here we use the [ ] operator, which turns a stream of elements into a bag of *tuples*, thus creating an unordered collection of the data parts. Note that in *Loss*, only the existence of *tuples* is considered. Their positions, which contain the order information, are captured by a separate safety criterion below.

2. **Disorder**: Order is a core property of data streams and consequently, disorder is another critical threat to safety in continuous query execution. An operation produces disordered output if at least one tuple in the observed stream does not have the exact same positional index as its corresponding tuple in the reference stream. In formal terms:

   $$Disorder \ = |\{y_j \in Y_{obs} : j - indexOf_{ref}(y_j) \neq 0\}|$$

   However, as we will see later in Chapter 8, for some operations we can further distinguish between different types of disorder.

3. **Duplicates**: An output stream produced by an operation contains duplicates, if there is at least a pair of output elements in the observed stream which have the exact same set of contributing input elements. More precisely

$$Duplicates = |\{(y_j \in Y_{obs}, y_{j'} \in Y_{obs}) : depends(y_j) = depends(y_{j'})\}|$$

Liveness Criteria

A Liveness property expresses that "something good must eventually happen" during a given execution [65]. We identified two complementary Liveness criteria for *Query Modification*:

1. **Termination**: A query Q is said to terminate if after receiving a Stop control element, the query eventually visits the *Stopped* state (in the interaction diagram 7.5). As an example, *DStop* without any restrictions on the semantics of the query expression may not terminate (e.g., closing condition of a semantic window never being satisfied).

2. **Progress**: A query Q is said to make progress if after receiving a Start control element, the query eventually visits the *Running* state (in the interaction diagram 7.5). Note that Progress is not necessarily a property that is observable in terms of the query output, since certain query semantics may prohibit the generation of output (e.g., a selection query whose condition is never satisfied).

This should be noted that these classes of correctness criteria are relevant for all complex lifecycle operations that use the Start and Stop control elements. Defining new basic control elements may require defining additional criteria.

7.2.4 A Methodology to Create Complex Control Elements

While the basic control elements can already support many use cases, the real benefit of this approach is that it provides a solid foundation to establish complex control and models. In order to do so, we propose the following methodology:

1. Formally define the new operation (e.g., *Pause/Resume*) on top of our mapping functions. This leads to new complex control element(s) (e.g., in case of query pause/resume, we name it *PR*), and extensions to our basic foundations if needed.

2. Define Correctness Criteria for the new operation (e.g., Safety, Liveness)
3. Derive the possible options for this complex control element by first combining the interaction diagrams of the individual queries -yielding a new interaction diagram with the power set of the states- and then transform the combined interaction diagram into a Finite State Machine (FSM). To this end, two steps need to be taken:
   (a) specifying the *Start* state and the *Accept* state
   (b) excluding the transitions which are not relevant for the target lifecycle operation

   Once the FSM is built, the language that it accepts, meaning all sequences of transitions that starts in the Start state and ends in the Accept state, contains all variations of the new control element. Parameters influencing the number of options are

   (a) the types of basic control elements considered,
   (b) the number of control elements (and state transitions) allowed,
   (c) their order, and
   (d) the distance between two basic control elements. this parameter considers the number of *data* elements between the Basic Control Elements. Although theoretically, distances of any arbitrary length are possible, in practice a small subset of these length are meaningful:
      i. zero length: Basic Control Elements follow each other directly (no data element in between)
      ii. in accordance to the preceding Basic Control Element (the drain distance for DStop, and the first output for FStart)
4. Evaluate the behavior of variants against the Correctness Criteria.

Using this methodology, we can make sure that all possible variations within the control element framework are covered. The next section will show how we apply the above methodology to model query pause/resume operation.

7.3 Example: Query Pause/Resume Model

Query Pause/Resume is a common lifecycle operation. Resource shortage is one of the most common reasons that forces the system to temporarily pause (suspend) execution of one or more of its running continuous queries. In such cases, as soon as the shortage is resolved, the paused queries should be resumed.

Here, following our methodology, we first precisely define what a pause/resume operation is, then after introducing the correctness criteria for this operation, we enumerate the possible variations of its execution and compare them in terms of guarantees they provide.

7.3.1 Formal Definition of Pause/Resume

Assuming that there is a query Q which is applied to a single input stream X and produces an output stream Y, the same input stream annotated by a *Pause/Resume* control element would produce another output stream which we refer to as Y^{pr} (*pr* is short for *Pause/Resume*). Intuitively, a Pause/Resume control element can be translated into a combination of Stop control element directly followed by a Start control element. Notice that although there is no positional difference between Start and Stop elements, the actual time difference between applying them can be arbitrarily long.

7.3.2 Correctness of Pause/Resume

In the following, we refine the correctness criteria definitions for pause/resume operation.

Safety Criteria

For safety criteria we only need to formally specify the Pause/Resume Reference Output Stream. It's the output stream produced by the query on the original unannotated input stream. In other words

$$Y^{ref} = Y$$

Having defined the Y^{ref}, all definitions given in Section 7.2.3 can be applied without further adaptation.

Liveness Criteria

Similarly, the Termination criterion implies that the paused query will eventually stop producing outputs and the Progress criterion implies the the query will eventually return to the running state.

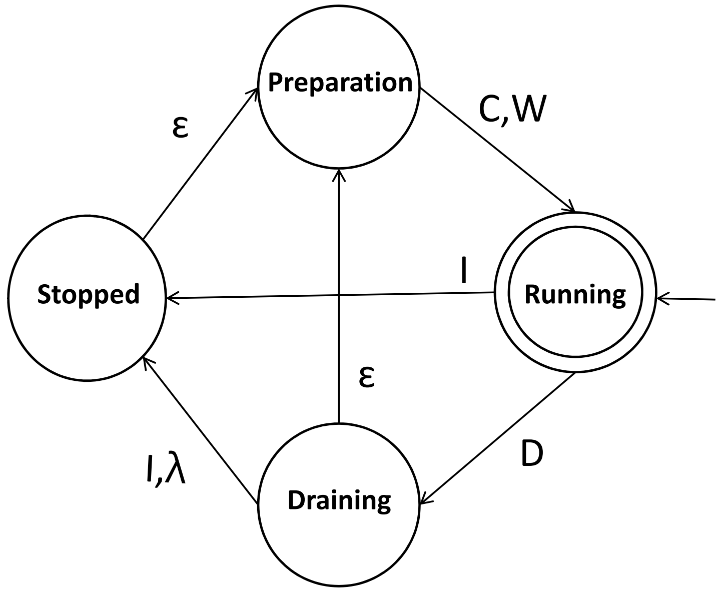


Figure 7.6: *Basic Control Elements FSM for Query Pause/Resume Operation*

7.3.3 Variations of the Pause/Resume Control Element

To explore the possible variations of the Pause/Resume control element, we first need to transform the interaction diagram in Figure 7.5 into a Finite State Machine (FSM). As shown in Figure 7.6, the Start and the Accept states are identical in this FSM. Moreover, in this FSM there is no transition with FStart control element (explained later).

Below, we point out the meaningful variations:

- Pause/Resume of type **I-W**: in which an Immediate Stop is directly followed by a Warm Start. In formal terms

  $$Y^{i-w_{pr}} = Y^{istop}||Y^{wstart}$$

  in which || denoted the append operator. This variation provides all of the desirable guarantee (no-Loss, no-Disorder, no-Duplicate, Termination, and progress); however, since it uses Warm Start, its actual implementation incurs persisting the unfinished output items' states.

- Pause/Resume of type **I-C**: in which an Immediate Stop is directly followed by a Cold Start. In formal terms

  $$Y^{i-c_{pr}} = Y^{istop}||Y^{cstart}$$

  this variation can be lossy.

- Pause/Resume of type **D-W**: in which a Drain Stop is directly followed by a Warm Start. In formal terms
$$Y^{d-w_{pr}} = Y^{dstop}||Y^{wstart}$$
this variation can generate duplicates. Moreover, it does not guarantee termination of the *pause* step.

- Pause/Resume of type **D-C**: in which an Immediate Stop is directly followed by a Cold Start. In formal terms
$$Y^{d-c_{pr}} = Y^{dstop}||Y^{cstart}$$
this variation is similar to the **I-W** variation in terms of guarantees, except for Termination which it does not provide. However, in terms of in terms of persistence overhead, it is less expensive than the **I-W** variation.

As we noted earlier, in this Pause/Resume Model we excluded the Fresh Start control element since it changes the query mapping functions and violates the natural semantics of pause/resume[3]. Using FStart to resume a paused query always causes loss and non-identical output streams. However, using this variant can be considered under circumstances in which loss is tolerated.

7.4 Concretization

So far, we have investigated our model for lifecycle and change using a black-box mapping function that covers a single query with a single input and a single output. To make our model applicable in practice, we need to perform several steps: (1) Refine the mapping functions and control elements to the level of operators, and determine how control elements need to be implemented on operators and their compositions. In turn, this also allows us to work with the compositions of queries. (2) Investigate how existing SPEs can be adapted to support the lifecycle model established in this chapter.

7.4.1 Control Elements on Composition

In Chapter 3, we have shown how query mapping functions can be composed out of operator mapping functions, thus defining how output is computed. To complete this composition, we need to determine the semantics in the presence of control elements.

[3]Using FStart, in fact, models the Query Restart operation!

More specifically, the following needs to be investigated: how is each control element handled in a composition of operators? To answer this question we assume the following query network Q which produces the output stream Y from the input stream X. It is a linear composition of two operators, namely $OP1$ and OP:

$$Y = Q(X)$$

$$Q :: OP1||OP2$$

assuming that Z is the intermediate results stream, we have

$$Z = OP1(X)$$

$$Y = OP2(Z)$$

$$\rightarrow Y = OP2(OP1(X))$$

the following statements hold for this query network (and linear compositions of any length in general):

$$Y_{IS} = Q(X_{IS})$$
$$Y_{IS} = OP2(OP1(X_{IS}))$$
$$Y_{DS} = Q(X_{DS})$$
$$Y_{DS} \neq OP2(OP1(X_{DS}))$$
$$Y_{FS} = Q(X_{FS})$$
$$Y_{FS} = OP2(OP1(X_{FS}))$$
$$Y_{CS} = Q(X_{CS})$$
$$Y_{CS} \neq OP2(OP1(X_{CS}))$$
$$Y_{WS} = Q(X_{WS})$$
$$Y_{WS} \neq OP2(OP1(X_{WS}))$$

In which X_{IS} denotes an input stream which contains an Immediate Stop Control Element. Proofs for two representative cases, namely IStop and DStop, are given later in this section.

In short, what these statements tell is that the straightforward propagation of the *FStart* and *IStop* between operators is sufficient to achieve the desired semantics (generating identical output to that of query-level mapping functions), since (1) the control element is defined on the input stream, (2) the first operator determines the contributions to the following operators and (3) *FStart* and *IStop* affect the output immediately.

However, for *DStop* (and *CStart* and *WStart* as well), this useful property does not generally hold, since the output elements which are drained cannot be determined by considering the first operator alone, unless all operators but the first are stateless. As an example for such a problematic drain consider nested windows, in which draining on the first (inner) window operator will not permit the second (outer) to produce meaningful results any more.

In such cases, we need more complex coordination among operators, since the first and intermediate operators do not have enough knowledge to handle the control elements, and therefore need to delegate this task to their following operators. This delegation ends when a *dominant* operator is reached, which will determine the drain output elements. On linear plans, the last stateful operator dominates the query output, and previous stateful operators (we call them subordinate) must keep producing output until the dominant operator has determined all input for draining.

If there is a dominant function, the precise mapping functions of the subordinate operators are not needed any more. Therefore, we can also support user-defined functions without knowing the detailed mapping function, as long as it is monotonic. For more general query plans, a dominant operator can be constructed using techniques similar to those proposed in the literature for load-shedding on multiple aggregates [81].

Proposition 7.4.1 *Operator Composition Behavior on Immediate Stop*

$$Y_{IS} = Q(X_{IS})$$

$$Y_{IS} = OP2(OP1(X_{IS}))$$

Proof

$$Y_{SS} = \{y_j | \quad \forall x_i \in dep_Q(y_j) \quad i < istop\}$$

$$= \{y_j | \quad \forall x_i \in dep_{OP1}(dep_{OP2}(y_j)) \quad i < istop\}$$

assuming $dep_{OP2}(y_j) = \{z_{k_1}, ..., z_{k_n}\}$

$$= \{y_j | \quad \forall x_i \in dep_{OP1}(\{z_{k_1}, ..., z_{k_n}\}) \quad i < istop\}$$

$$= \{y_j | \quad \forall x_i \in dep_{OP1}(z_{k_1}) \cup ... \cup dep_{OP1}(z_{k_n}) \quad i < istop\}$$

$$= \{y_j | \quad (\forall x_i \in dep_{OP1}(z_{k_1})) \wedge ... \wedge (\forall x_i \in dep_{OP1}(z_{k_n})) \quad i < istop\}$$

$$= \{OP2(\{z_{k_1}, ..., z_{k_n}\}) | \quad (\forall x_i \in dep_{OP1}(z_{k_1})) \wedge ... \wedge (\forall x_i \in dep_{OP1}(z_{k_n}))$$

$$i < istop\}$$

$$= \{OP2(\{z_{k_1}, ..., z_{k_n}\})| \quad z_{k_1} \in Z_{IS} \wedge ... \wedge z_{k_n} \in Z_{IS} \quad i < istop\}$$

$$= OP2(Z_{IS})$$

Notice that in the proof above we take advantage of the following equivalence:

$$dep(\{y_j, y_{j'}\}) = dep(y_j) \cup dep(y_{j'})$$

Proposition 7.4.2 *Operator Composition Behavior on Drain Stop*

$$Y_{DS} = Q(X_{DS})$$

$$Y_{DS} \neq OP2(OP1(X_{DS}))$$

Proof

$$Y_{DS} = \{y_j| \quad \exists x_i \in dep_{Q}(y_j) \quad i < dstop\}$$

$$= \{y_j| \quad \exists x_i \in dep_{OP1}(dep_{OP2}(y_j)) \quad i < dstop\}$$

assuming $dep_{OP2}(y_j) = \{z_{k_1}, ..., z_{k_n}\}$

$$= \{y_j| \quad \exists x_i \in dep_{OP1}(\{z_{k_1}, ..., z_{k_n}\}) \quad i < dstop\}$$

$$= \{y_j| \quad \exists x_i \in dep_{OP1}(z_{k_1}) \cup ... \cup dep_{OP1}(z_{k_n}) \quad i < dstop\}$$

$$= \{y_j| \quad (\exists x_i \in dep_{OP1}(z_{k_1})) \vee ... \vee (\exists x_i \in dep_{OP1}(z_{k_n})) \quad i < dstop\}$$

$$= \{OP2(\{z_{k_1}, ..., z_{k_n}\})| \quad (\exists x_i \in dep_{OP1}(z_{k_1})) \vee ... \vee (\exists x_i \in dep_{OP1}(z_{k_n}))$$

$$i < dstop\}$$

Notice that here there can be z_{k_m} for which, $\forall x_i \in dep_{OP1}(z_{k_m}) \quad i > dstop$

$$= \{OP2(\{z_{k_1}, ..., z_{k_n}\})| \quad z_{k_1} \in (Z_{IS} \cup \Delta) \wedge ... \wedge z_{k_n} \in (Z_{IS} \cup \Delta) \quad i < dstop\}$$

$$= OP2(Z_{IS} \cup \Delta)$$

Special Case If $OP2$ is *stateless* (meaning $|dep_{OP2}(y_j)| = 1$) then

$$Y_{DS} = Q(X_{DS})$$

$$Y_{DS} = OP2(OP1(X_{DS}))$$

This is generalizable to arbitrary *stateless* operators following the last *statefull* operator in the query network.

7.4.2 Our Framework on SPEs

The framework that we have introduced in this chapter has been designed to be general, abstract, and conservative in terms of its assumptions, thus making it applicable in the context of a broad range of SPEs and their query models. In practice, individual SPEs often provide more restricted models, and therefore, our framework can be specialized for the SPE at hand. By doing so, stronger correctness guarantees and more efficient implementations can be achieved.

For example, systems providing only count- and time-based windows (e.g., Borealis [15]) do, by definition, always fulfill Termination and Progress criteria. Similarly, for certain SPEs, the query mapping functions stay fixed over a Stop/Start cycle, since time-based windows are opened based on a predefined time domain, and are not influenced by the position of the Start control element. Thus no issues deriving from initializing a mapping functions come up.

Our approach reaches its limitations when (1) non-monotonic operators such as sort are present, and (2) the output data elements are computed in a non-deterministic way (e.g. affected by system time). It can still be implemented, but the guarantees it can provide are inherently weaker.

As proof of concept, in Section 8.3, we will show how our framework can be specialized for two very different SPEs: MXQuery [10] as a pull-based, language-oriented implementation, and Borealis [15] as a push-based query network.

7.5 A General Architecture to Implement Our Framework

So far, we have discussed query lifecycle, at the level of formal models. Since our goal has been to provide a complete picture of the query lifecycle, we have studied ways to implement the proposed semantics.

Previously, many approaches have been studied for efficient implementation of specific lifecycle operations in databases and SPEs. These approaches typically solve a specific problem and make several assumptions in order to achieve good performance. We instead chose to develop generic architectural extensions that require only minimal changes to the SPE's data and query models.This wrapper-like extension allows the implementation of the necessary control and change logic on top of its already existing architectures.

Our proposed architectural extensions is shown in Figure 7.7.Below, we describe com-

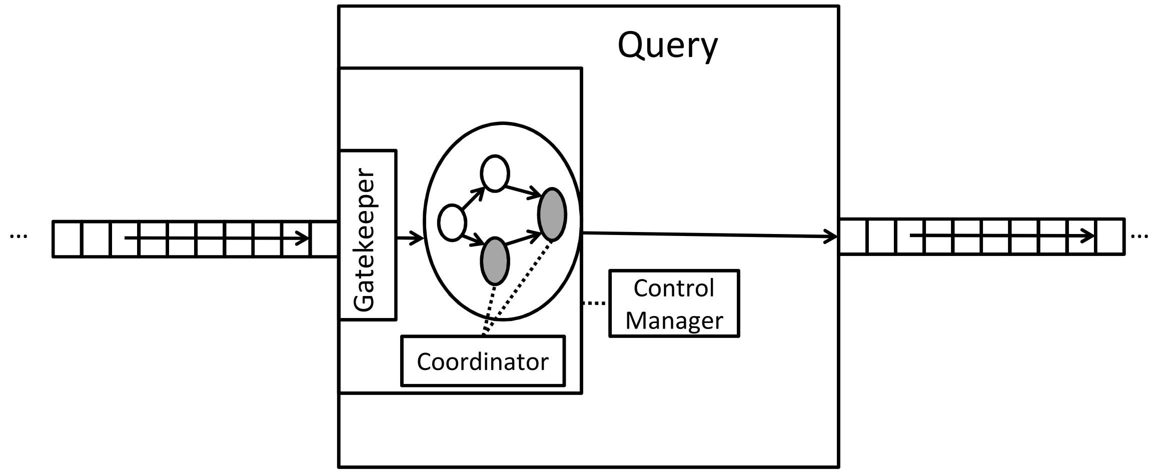


Figure 7.7: *Basic Query Lifecycle Architecture*

ponents in this figure.

Gatekeeper: An important aspect of our framework is the implementation of the basic control elements (i.e., FStart, IStop, DStop), since complex control elements can then be built on top of these. As shown in the previous section, *FStart* and *IStop* can be implemented by just affecting the first operator in the plan. Instead of modifying each operator to support these semantics, we place a special operator with an identity mapping function and the control logic in front of the plan, which we call *Gatekeeper*. Therefore, we do not need to change any operator, greatly simplifying the integration. Since we need to control each *Query Version* independently, there is a *Gatekeeper* for each.

Drainable Operators: *DStop* (and similarly, *CStart* and *WStart*), on the other hand, requires a slightly more invasive approach for stateful operators: Stateful operators in the query plan (e.g., windowing, pattern matching, joins) need to be extended with the ability to perform draining (i.e., completing the processing of the already started windows, but not initiating new ones) and persisting (i.e. persisting the incomplete outputs), yet this facility needs only to be enabled on the dominant operator. For a given windowing operator implementation, only minimal extensions are necessary, since it is already computing the contributing elements when building the windows. For example, for our MXQuery implementation, the draining extension required only about 30 lines of code to be added to the windowing operator.

Coordinator: Instead of extending all operators to propagate the information necessary for a *DStop, CStart,* and *WStart* to the dominant operator, we externalize this logic into a separate component, called *Coordinator*. It interacts with the dominant operator and the gatekeeper, passing on the relevant information and controlling the execution flow.

Control Manager: The Control Manager is responsible for interpreting the control elements. If the queries are known in advance, optimizations can be performed by this component.

As we will explain in the next chapter, we have implemented and integrated this architectural extension into MXQuery[10], an XQuery language SPE.

7.6 Related Work

The use of control elements has been inspired by the punctuation-based stream processing work of Tucker et al. [84], yet with different semantics. In that work, data streams are annotated with punctuations to mark the end of a subset of data in the stream, which are then exploited for optimizations.

Finally, our work also relates to stopping and restarting of long-running in data warehouses [65, 30, 32]. In this case, some queries are intentionally terminated and later restarted to deal with resource contention. The restart should reuse some of the old state for efficiency reasons. Stopping and restarting the same query constitutes a special case in our more general framework. Furthermore, streaming has different semantic requirements than traditional warehouses, e.g., ordered data delivery.

7.7 Conclusions

In this chapter, we presented a punctuation-based framework to model query lifecycle operations. By representing query semantics with dependency functions and by introducing a small set of basic operations, we were able to establish a general framework where complex lifecycle operations can be created out of the basic operations. The set of basic operation currently contains query start and query stop, but it can be extended by adding other basic operations such as load-shedding.

This framework provides a mechanism to evaluate different variations of a control element. For this purpose, it defines both safety and liveness criteria. Safety criteria capture loss, duplication, and disorder, whereas liveness criteria capture termination of a stopped query and progress of a started one.

Moreover, our work builds up a powerful methodology that allows us to easily extend our framework even further to implement other query lifecycle operations. We executed this methodology to model query pause/resume operation.

We also presented a general architecture to implement this framework on typical SPEs, without requiring much effort or fundamental changes on the existing implementation.

As part of ongoing and future work, we plan to extending our abstraction and general framework to support multiple-input stream queries (i.e., join and union). As another direction for future work, we plan to utilize our methodology to model other query lifecycle operations such as recovery and query migration.

Chapter 8

A Model for Continuous Query Modification

8.1 Introduction

8.1.1 Motivation and Use Cases

As we pointed out at the beginning of the previous chapter, continuous queries, in their lifetime, may be subject to lifecycle operations. One of the most challenging and the least explored lifecycle operation is *query modification*. In fact, in contrast to other operations such as *query migration*, *query pause/resume*, or *query re-optimization*, the semantics of the query changes during the transition phase. Moreover, as we will show below, despite the widespread use of query modification in streaming applications, it has not received the necessary attention in previous work.

Continuous queries may need to be modified either due to changes in application semantics, or due to changes in system behavior as illustrated by following examples.

- **Security Monitoring**: Consider a bank that applies the following security policy for its ATM machines (adapted from real-world policies in MASTER Project [9], as explained in Chapter 2): Block a customer card upon 3 failed logins at the same ATM location within a time window of 10 minutes. This policy can be implemented as a continuous count query on a 10-minute window over a stream of failed login events. Now suppose that, due to a change in regulations, the bank would like to change the window in this query to 15 minutes. If this change happens while a user has already tried 2 failed logins within 5 minutes, it is not obvious how the system should behave. A naive approach would be to replace the query with the new one immediately, discarding any existing state. In this case, the user would be able to try up to 3 more logins in the next 15 minutes in addition to the 2 failed ones in the past 5 minutes (leading to a total of 5 tries over 20 minutes, not matching any policy!). A more cautious approach would be to defer the query replacement until a time there is no incomplete query state left, but in more complex use cases such as stock trading, such periods may not exist. (Sections 8.2.3 and 8.2.3 show solutions).

- **Sensor Networking**: Consider a network of temperature and smoke sensors deployed over a forest in order to detect and monitor wildfires. Again, continuous queries can be defined over these sensor readings in order to signal unusual increases in sensor values. Whenever such activity is reported for a certain region of the forest, the firefighters want to replace the currently running query with a more specific query so that the possible fire can be located with a higher degree of certainty. However, in sensor-based applications, there is already an inherent level of uncertainty and loss, and it is also critical for the new query to take effect as fast as possible. Therefore, for this application a lossy, but more responsive approach is better (leading to the solution in Section 8.2.3).

  In the above two use cases, the application semantics necessitates query modification, albeit with different requirements. There are also cases where the modification is triggered by the system itself.

- **Adaptive Load Management**: SPEs need to deal with resource overload; e.g., caused by fluctuating arrival rates. A common technique is load shedding; e.g., by inserting/removing load-reducing drop operators into/from selected parts of a running query plan [80]. This yields a "cheaper" version of the query, while the quality of the results becomes lower. In an alternative strategy, several versions of the same query are defined by the application in advance, each tailored for a different load level (see the military use case in [15]). As the system load changes due to fluctuations in input rates, the system is expected to switch adaptively between different query versions. It is crucial that the switch across different query versions happens seamlessly and efficiently, with as little additional system overhead as possible.

The above examples show the importance and diversity of query modification capability in SPEs. Each requires a different tradeoff between correctness parameters and performance, and provides different information to exploit. What is needed is a general-purpose, reliable, and efficiently implementable model for modifying continuous queries at run-time.

8.1.2 Contributions

In this chapter, we have made the following contributions:

- Defining correctness guarantees for query modification.
- Leveraging our framework and methodology from Chapter 7 to model continuous query modification.
- Exploring the full guarantee space (in terms of safety and liveness criteria) for query modification and based on that, extracting a set of general rules.
- Extending our proposed general architecture from Chapter 7 to support query modification.
- A prototype implementation of this query modification model and benchmarking it to explore the practical aspects and tradeoffs among our query modification variations.

8.1.3 Outline

The rest of this chapter is organized as follows. The description of our query modification model is presented in Section 8.2. In Section 8.3, we describe how our model can be refined for real-life SPEs and provide an architecture and implementation on a state-of-the-art SPE. Section 8.4 reports on results of our performance study and gives guidelines about when to use which method. Finally, after giving a summary of related work in Section 8.5, we conclude this chapter in Section 8.6.

8.2 The Query Modification Model

Having introduced our modeling framework and methodology in previous chapter, we now show how query modification can be modeled. In the query modification model we propose, there are two versions of a query Q; namely Q^{old} and Q^{new}. The goal is to switch from the former to the latter. Following the first step in our methodology (Section 7.2), we will first express the query modification operation by a new complex control element, called *Change* (Section 8.2.1). Second, we will define correctness criteria for query modification (Section 8.2.2).

Change can be translated into a combination of a *Stop* control element, which targets Q^{old}, and a *Start* control element, which targets Q^{new}. However, as we will explain, there can be different variations of this combination, derived by interaction diagram composition, and analyzed in Section 8.2.3. This is complemented by an analysis of interaction of complex and basic control elements in Section 8.2.4. Finally, we will conclude the discussion in Section 8.2.6 by presenting a set of correctness rules for query modification.

8.2.1 Definition of Query Modification

In query modification, there are two versions of a query Q, old (Q^{old}) and new (Q^{new}). Both versions applied to a single input stream. The change control element is formalized by extending the query definition to include query versions (*old* and *new*). Accordingly, we will have two pairs of query mapping functions: $depends_{old}(y_j^{old})$ and $contributes_{old}(x_i)$; as well as $depends_{new}$ (y_j^{new}) and $contributes_{new}(x_i)$.

In case of *Change* there are three output streams: Output stream of Q^{old}, output stream of Q^{new}, and *Change* output stream, denoted by Y^{old}, Y^{new}, Y^{chg} respectively. A change control element defines how the Y^{chg} is built from Y^{old} and Y^{new}.

Our running example throughout the rest of this section is depicted in Figure 8.1. We want to modify $Q1^{old}$ into $Q1^{new}$, which is another continuous aggregation with a tuple-based sliding window of size 2 and slide 2, applying a sum over each window.

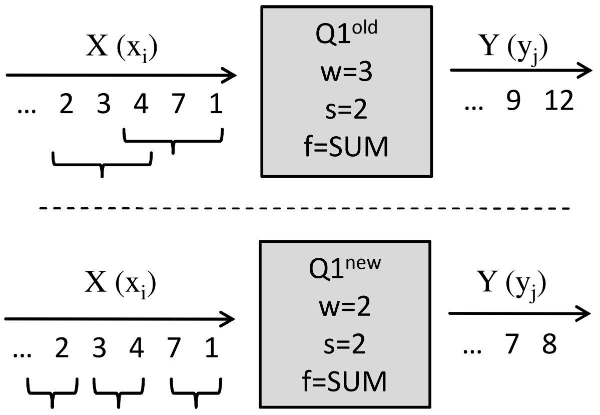


Figure 8.1: *The Old and The New Versions of Q1*

8.2.2 Correctness of Query Modification

In the following, we refine the correctness criteria definitions for query modification.

Safety Guarantees

For safety guarantees we only need to formally specify the Query Modification Reference Output Stream. Intuitively, an ideal or lossless *Change* for a query Q should not lose incomplete contributions from Q^{old}, and at the same time it should include all contributions from Q^{new}[1]. In formal terms

$$Y_{ref} \;=\; Y_{ref}^{old} || \, Y_{ref}^{new}$$

where

$$Y_{ref}^{old} = \{y | y_j \in Y^{old} \wedge \exists \, x_i \;\in\; depends_{old}(y_j^{old}) \;\wedge\; i < chg\}$$
$$Y_{ref}^{new} = \{y | y_j \in Y^{new} \wedge \forall \, x_i \;\in\; depends_{new}(y_j^{new}) \;\wedge\; i > chg\}$$

and *chg* denotes the position of the change element.

1. **Loss**: As an example, assume that we want to switch from $Q1^{old}$ to $Q1^{new}$ by enforcing an IStop on $Q1^{old}$ and an FStart on $Q1^{new}$ (i.e., Change = IStop + FStart). As shown in Figure 8.2, this leads to a lossy *Change* since it produces one fewer output item (11) than then the reference stream.

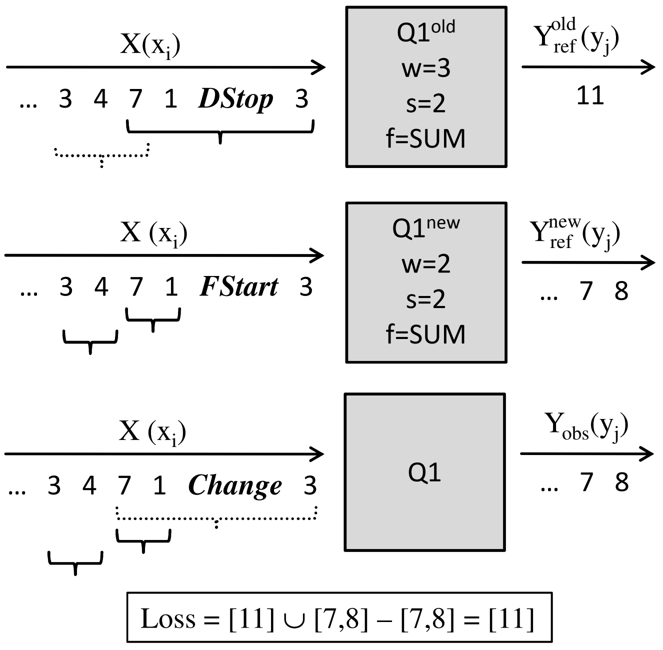


Figure 8.2: *A Lossy Change on Q1*

[1] In other words, the reference stream would be modeled by using DStop + FStart.

2. **Disorder**: We define two levels of order violation for query modification:

   (a) **Query-Level Disorder**: At least one output element from Q^{old} appears after an output element from Q^{new}. Formally, this is defined as follows:

$$\begin{gathered}\exists y_j^{chg}, y_{j'}^{chg} \in \ Y^{chg} : \\ org(y_j^{chg}) \in Y^{new} \wedge org(y_{j'}^{chg}) \in Y^{old} \\ \wedge \ j < j'\end{gathered}$$

   where

$$org(y_j^{chg}) = \begin{cases} y_l^{old} & \text{if } y_j^{chg} \text{ is taken from } Y^{old} \ ; \\ y_k^{new} & \text{else.} \end{cases}$$

   (b) **Stream-Level Disorder**: The order imposed by the query semantics and the structure of the input streams is not preserved. Formally[2],

$$\begin{gathered}\exists y_j^{chg}, y_{j'}^{chg} \in \ Y^{chg} : \\ org(y_j^{chg}) \in Y^{new} \wedge org(y_{j'}^{chg}) \in Y^{old} \\ \wedge \ j > j' \wedge \\ max\left(indexOf\left(depends_{new}(org(y_j^{chg}))\right)\right) < \\ max\left(indexOf\left(depends_{old}(org(y_{j'}^{chg}))\right)\right)\end{gathered}$$

   where

$$indexOf(X) = \{i | x_i \in X\}$$

   Figure 8.3 depicts the two types of disorder. To illustrate the difference we here used the window size 4 for $Q1^{old}$ instead of 3. The figure shows that for SDChange, the output appears in the same order as the windows are closed, while for QDChange the output of Q^{old} wholly precedes the output of the Q^{new}.

   In short, as we will prove later in this section, no loss-free change can preserve both of these orders.

3. **Duplicates**: Two output elements from Q^{old} and Q^{new} are considered duplicates if they share the exact same contributing input elements. Formally,

[2]The *max* function here can be considered as generalization of the ordering windows by their last items. In definition of Stream-Level Disorder, other types of *statistical representatives* can also be applied as well. Another common example can be *min* (order by first item of the windows).

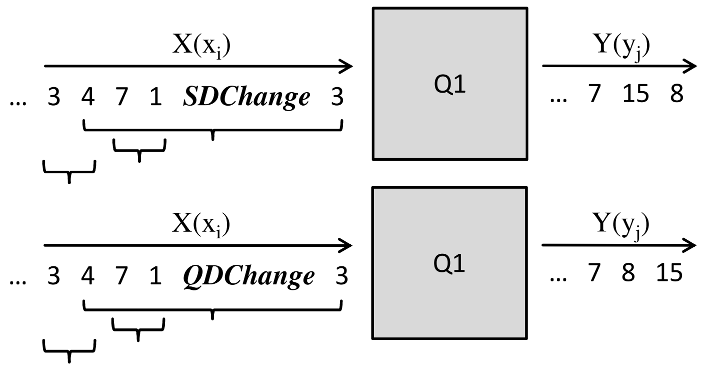


Figure 8.3: *Query- vs. Stream-level Disorder on Q1*

$$y_{j'}^{chg} \text{ is a duplicate of } y_j^{chg} \text{ iff:}$$

$$depends_{new}(org(y_j^{chg})) = depends_{old}(y_{j'}^{chg}))$$

Note that y_j^{chg} and $y_{j'}^{chg}$ can be duplicates even if they carry different values, as long as they depend on the same input stream elements X.

Liveness Criteria

Here are the refined definitions of Liveness Criteria for query modification.

1. **Termination of the Old Query**: Q^{old} eventually goes into the Stopped state in its lifecycle interaction diagram (defined in Section 7.2).
2. **Progress of the New Query**: Q^{new} eventually goes into the Running sate in its lifecycle interaction diagram.

   It is noteworthy that termination of the old query version does not necessarily imply the progress of the new query version and vice versa.

8.2.3 Variations of the Change Control Element

We will now derive the *Change* options by combining our basic control elements (Start and Stop) in different ways (i.e., Step 3 in our methodology). The key idea is to build a common interaction diagram for both query versions from the (simplified) interaction diagram of a single query shown in Figure 8.4. We have decided to exclude Cold Start and Warm Start Control Elements (and consequently the Preparation state) from the original interaction diagram in Figure 7.5; primarily because such start methods are not necessary to describe query modification, since a modification will establish a new mapping function. In addition, it simplifies the interaction diagram.

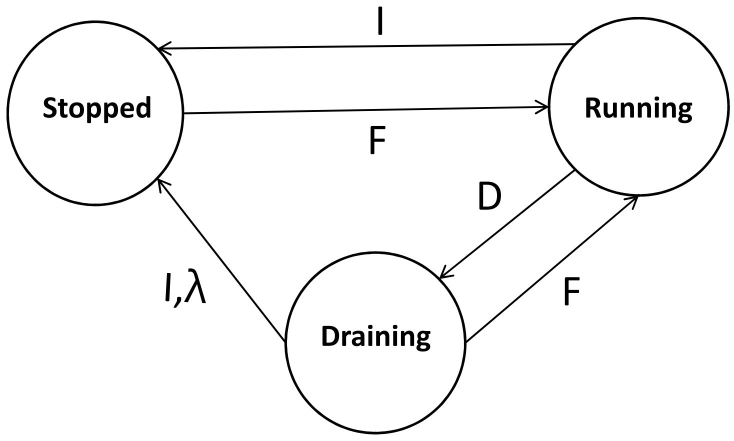


Figure 8.4: *The Projected Interaction Diagram for Single Query*

The resulting interaction diagram is shown in Figure 8.5, on which each of the states is labeled with the state for each of the queries (e.g., RS means Q^{old} is **R**unning and Q^{new} is **S**topped), and transitions are the combination of the individual query version's transitions (e.g., *IStop* for Q^{old} on RS will lead to SS). As outlined before, Change in translated into stopping Q^{old} and starting Q^{new}. In other words, in the corresponding FSM the Initial state is RS and the Accept state is SR.

Once the initial state and the final state have been identified, there are four parameters which influence the number of *paths* between them (notice that each distinct path defines a variation of Change control element):

1. Types of the basic control elements: here we have one option for Start (FStart), and two options for Stop (IStop and DStop).
2. Number of intermediary states: in the case of Change, we restrict this to be at most two.
3. Order of basic control elements: which dictates how basic control elements follow/precede each other.
4. Distance (in terms of data elements) between the basic control elements: depending on the case, we use *direct*, *drain distance* (applicable to DStop), and *first output* (applicable to FStart).

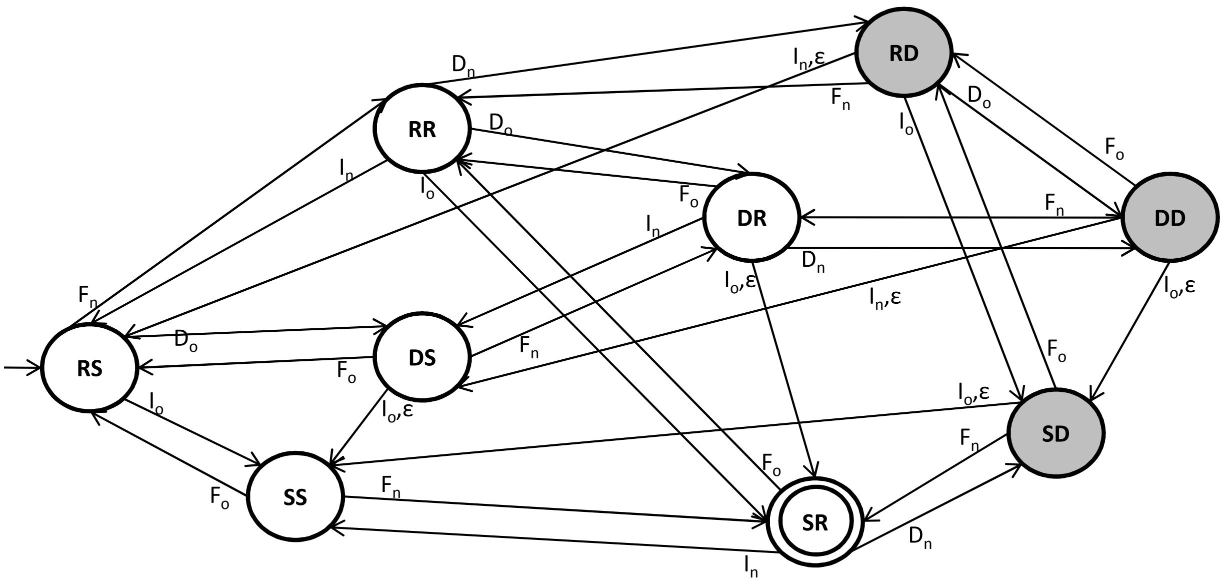


Figure 8.5: *Lifecycle Interaction Diagram - Two Queries*

Stop Type	Order	Distance	Change Option
IStop	I F	direct	IChange
IStop	I F	waiting	n/a
IStop	F I	direct	IChange
IStop	F I	at first result	GIChange
DStop	D F	direct	DChange
DStop	D F	drain completion	DDChange
DStop	F D	direct	DChange
DStop	F D	at first result	GDChange

Table 8.1: *Change Variation Derivation*

As shown in the tree in Figure 8.6, and Table 8.1, there are 8 options in total, of which one can be discarded (IStop-FStart with waiting), since it does not provide any meaningful guarantees. Some of these options are equivalent, since (1) they are executed next to each other with no data operations in between, (2) changing the order of two control elements does lead to the same target state, e.g., for both cases of *IChange*.

All in all, there are 5 variants of Change, which we will now discuss in terms of their correctness, performance, and use cases.

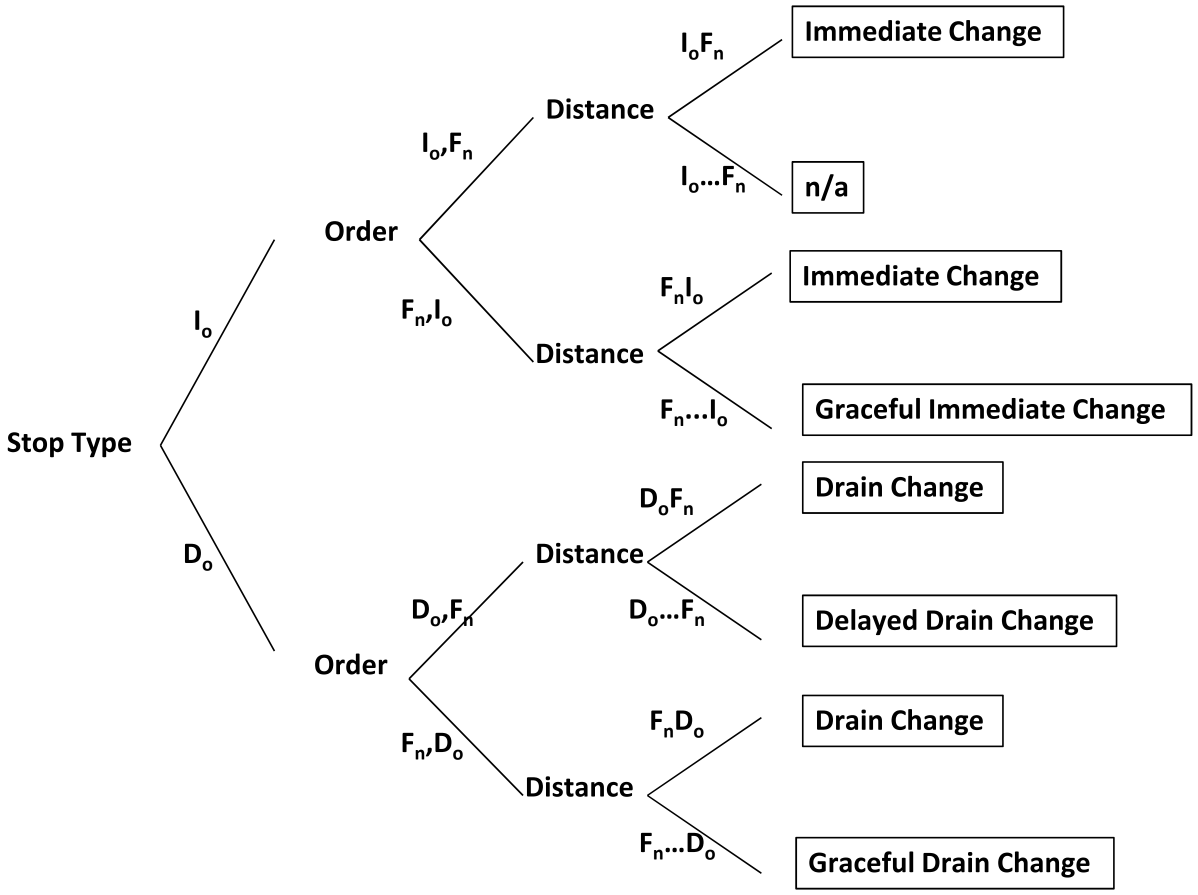


Figure 8.6: *All Possible Variations of Change Control Element Using Two Basic Control Elements*

Immediate Change (IChange)

In some applications, a *Change* should be performed as early as possible, disregarding any partial results of Q^{old}. This is expressed with an IChange control element, depicted as $x_{ichange}$, composed of an *IStop* for Q^{old}, followed directly by an *FStart* for Q^{new}, or vice versa. Thus, the output stream is defined as:

$$Y^{ichg} = Y^{new}_{fstart} \mid\mid Y^{old}_{istop}$$

$$= \left\{ y^{new}_j \in Y^{new} \middle|\ \forall x_i \in depends_{new}(y^{new}_j),\ i > ichange \right\}$$

$$\mid\mid \left\{ y^{old}_j \in Y^{old} \middle|\ \forall x_i \in depends_{old}(y^{old}_j),\ i < ichange \right\}$$

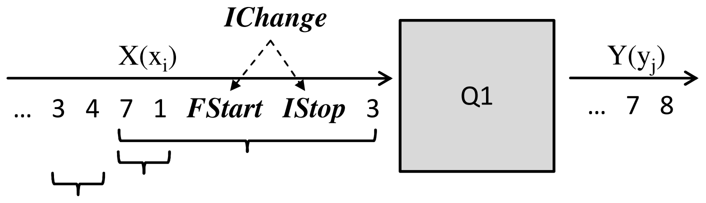


Figure 8.7: *Immediate Change on Q1*

Recall that || is the *append* operator which concatenates two streams.Figure 8.7 shows an example of using IChange control element.

In terms of safety, IChange can cause loss, since the partial results of Q^{old} are discarded. It produces the outputs in the correct order (both stream- and query-level), and does not generate any duplicate output elements (see Figure 8.2). Moreover, in terms of liveness, IChange guarantees both the termination of Q^{old} as well as the progress of Q^{new} (see Figure 8.3). Proofs for these claims can be found in Section 8.2.5.

Safety / Change	**No Loss**	**No Query-Level Disorder**	**No Stream-Level Disorder**	**No Duplicates**
IChange	✗	✓	✓	✓
DDChange	✗	✓	✓	✓
QDChange	✓	✓	✗	✓
SDChange	✓	✗	✓	✓
GIChange	✗	✓	✓	✓
GDChange	✓	✗	✓	✗

Table 8.2: *Safety Guarantees of Change variants*

With respect to the use cases introduced in Section 8.1, *Sensor Networking* is a candidate for IChange, since loss is tolerable while the immediate progress of the new query (in case of an emergency) and preserving the energy-saving requirements of the sensors need to be guaranteed. Given that IChange does not require *DStop* nor any other kind of complex change coordination, its behavior corresponds to what a typical SPE would do.

Delayed Drain Change (DDChange)

An alternative approach for *Change* is to ensure that Q^{old} is drained, and only then Q^{new} is started. We call the corresponding control element DDChange, denoted as $x_{ddchange}$.

Change \ Liveness	**Progress**	**Termination**
IChange	✓	✓
DDChange	✗	✗
QDChange	✓	✗
SDChange	✓	✗
GIChange	✓	✗
GDChange	✓	✗

Table 8.3: *Liveness Guarantees of Change variants*

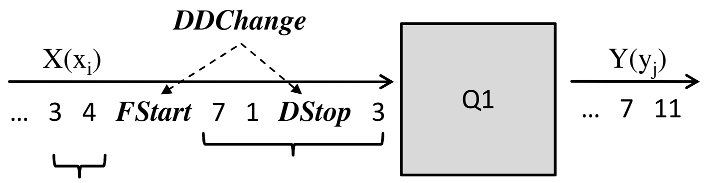


Figure 8.8: *Delayed Drain Change on Q1*

Formally, it is composed of a *DStop* for Q^{old}, followed by an *FStart* for Q^{new} in a statically-incomputable distance . Thus, the output stream is defined as:

$$Y^{ddchg} = Y^{new}_{fstart} || \; Y^{old}_{dstop}$$

$$= \left\{ y^{new}_j \in Y^{new} | \; \forall x_i \; \in depends_{new}(y^{new}_j), \; i \; > ? \; \right\}$$

$$|| \left\{ y^{old}_j \in Y^{old} | \; \exists x_i \; \in \; depends_{old}(y^{old}_j), \; i \; < \; ddchange \; \right\}$$

where '?' denotes the fact that the real position of the FStart element which will be inserted, is not known in advance (*drain distance*) . Therefore the *starting position* of the mapping function for Q^{new} is different to that of all the other change cases (which are all initialized at the position of change), so that output after the change may also be different.Figure 8.8 shows an example of using DDChange control element.

It should be noted that in contrast to all other Change variations, Delayed Drain Change uses a different pair of mapping functions. This is due to the fact that in this case, the effective position of FStart id determined by old query's mapping functions.

In terms of safety, DDChange exhibits a behavior similar to that of IChange. However, DDchange offers neither termination nor progress guarantees.

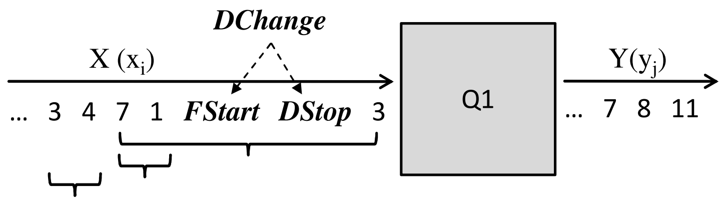


Figure 8.9: *Drain Change on Q1 (QDChange)*

Drain Change (DChange)

Both Change variants presented above can cause loss, which is not desirable in many streaming scenarios. This *Loss* is caused by the input elements that are no longer picked up by Q^{old} and are not yet considered by Q^{new}, since Q^{new} is only started when Q^{old} has been completed. Next, we introduce Drain Change, which is composed of a *DStop* for Q^{old} followed directly by an *FStart* for Q^{new}, or vice versa. Figure 8.9 shows an example of using a DChange control element.

In contrast to previous Change variations, merging the output streams of the old and the new queries is not straightforward in Drain Change, due to the fact that we have overlapping output elements. Hence, we distinguish between two variants of DChange: QDChange, a query-level order preserving Drain Change and SDChange, a stream-level order preserving Drain Change. In QDChange the output stream is defined as:

$$Y^{qdchg} = Y^{new}_{fstart} || \ Y^{old}_{dstop}$$

$$= \left\{ y_j^{new} \in Y^{new} | \ \forall x_i \ \in depends_{new}(y_j^{new}), \ i \ > \ dchange \ \right\}$$

$$|| \left\{ y_j^{old} \in Y^{old} | \ \exists x_i \ \in \ depends_{old}(y_j^{old}), \ i \ < \ dchange \ \right\}$$

while in SDChange is defined as:

$$Y^{ddchg} = Y^{new}_{fstart} \ ||| \ Y^{old}_{dstop}$$

$$= \left\{ y_j^{new} \in Y^{new} | \ \forall x_i \ \in depends_{new}(y_j^{new}), \ i \ > \ dchange \ \right\}$$

$$||| \left\{ y_j^{old} \in Y^{old} | \ \exists x_i \ \in \ depends_{old}(y_j^{old}), \ i \ < \ dchange \ \right\}$$

in which ||| is the *interleave* operator. It interleaves the output elements from the new and the old queries based on their relative dependencies on the input elements. Therefore, it may result in query-level disorder. Going back to Figure 8.2, it demonstrates the difference between SDChange and QDChange control elements.

In terms of safety, both DChange variants are lossless and free of duplicates, but neither of them can provide both order guarantees at the same time. As will be discussed further in Section 8.2.6, this is not caused by the design of our query modification model, but is rather an inherent problem of query modification. In terms of liveness, both DChange variants guarantee the progress of Q^{new}, but not the termination of Q^{old}. The *DChange* variants provide a good match to the requirements of the *Security Monitoring* use case, since they avoid loss and end the execution of Q^{new} by draining.

Graceful Change (GChange)

The approaches discussed so far did not cater for responsiveness, defined as the time elapsed between the last output of Q^{old} to the first output of Q^{new}. By keeping Q^{old} running (instead of draining or stopping it) until the first output of Q^{new} is produced, the responsiveness can be significantly improved. We call this change method *Graceful Change* (GChange).

We further distinguish between two variants of GChange: *Graceful Immediate Change* (GIChange) and *Graceful Drain Change* (GDChange[3]).

In GIChange the output stream is defined as:

$$Y^{gichg} = Y^{new}_{fstart} || \; Y^{old}_{istop}$$

$$= \left\{ y^{new}_j \in Y^{new} | \; \forall x_i \; \in depends_{new}(y^{new}_j), \; i \; > \; gichange \; \right\}$$

$$|| \left\{ y^{old}_j \in Y^{old} | \; \exists x_i \; \in \; depends_{old}(y^{old}_j), \; i \; < ? \; \right\}$$

while in GDChange is defined as:

$$Y^{gdchg} = Y^{new}_{fstart} \; ||| \; Y^{old}_{dstop}$$

$$= \left\{ y^{new}_j \in Y^{new} | \; \forall x_i \; \in depends_{new}(y^{new}_j), \; i \; > \; gdchange \; \right\}$$

[3]We use the Stream-ordered variation of it.

$$||| \left\{ y_j^{old} \in Y^{old} |\ \exists x_i \ \in \ depends_{old}(y_j^{old}),\ i \ < ?\ \right\}$$

where in both cases, $'?'$ denotes the fact that the real position of the Stop element which will be inserted, is not known in advance (*first output distance*).

In GIChange, loss can occur, but no duplicates and disorder; whereas in GDChange, duplicates and query-level disorder can occur, but no loss. Q^{new} will make progress, since it starts immediately, but the termination Q^{old} is not guaranteed, since it depends on the existence of output of Q^{new}, which is not guaranteed.

A typical good use of GChange is when Q^{new} has a low output rate (e.g., very large window size and very small window slide, very low selectivity, etc.). As shown in Section 8.4, GChange can indeed outperform other *Change* approaches in terms of responsiveness.

Graceful Immediate Change is also a candidate for the *Security Monitoring* use case, because it avoids loss of the Q^{old} until Q^{new} produces its first output and provides better responsiveness.

8.2.4 Interaction of Control Elements Revisited

So far we have defined the semantics of a *single* complex control element by mapping it to basic control elements and their interaction. In the next step, we need to cater for the interaction of *multiple* complex control elements or of a complex control element with a basic control element on the whole query. In particular, since changes (apart from IChange) do not complete immediately, such overlapping actions need to be properly defined. An examples of such a case would be an IStop on a query that currently performs a DDChange, as the user decides that results are no longer needed. In this scenario, one would expect no more output after an IStop, but the naive application of the DDChange translation means to perform a FStart after draining, which would start the query again.

Similar to the interaction of basic control elements, we therefore want to ensure that (1) control elements become effective at their specified position, and (2) the order of control elements determines that the latest control element is effective. The interaction diagram for two queries (Figure 8.5) does not provide a direct answer, since it only specifies the behavior of two query versions, no global operations. Yet it already contains the necessary foundations to define our extended semantics:

For a basic control element appearing during a change, we can translate IStop and DStop by applying them on both versions, thus achieving stop semantics. In turn, FStart can be translated into an IStop for Q^{old} and an FStart for Q^{new}. This ensures a start of Q^{new} with correct *starting index*, albeit with possible *Loss* on Q^{old}, since the change is turned into an IChange. A change applied after a basic control element will in any case lead to a running query, since our definition of change requires liveness. If the query is already started or stopped, the implementation is obvious, for a draining query we can again rely on the interaction rules of basic control elements, since the stop of Q^{old} will not extend the drain period of the stop on the whole query.

For complex control elements following other complex control elements, we can build an interaction diagram for three (or more) query versions using the same rules as we built one for two query versions in Section 8.2.3. In the case of 3 query versions, Q^{new} of the first change is Q^{old} of the second. We then translate the actions that are applied to Q^{old} (on the two-version case) onto the first two queries. The correctness analysis for multiple overlapping changes is analogous.

As a result, we are getting a weak transactional model for change: ensuring that we always complete a change is only possible for IChange, while other change models do not provide this guarantee. In our opinion, this is actually a desirable behavior, as it allows more flexibility. In addition, stronger transactional models require giving up strict definitions of position impact and sequential order of Control Elements.

8.2.5 Guarantee Proofs

Proposition 8.2.1 *Immediate Change can be lossy.*

Proof According to the definition of Immediate Change, the output stream is as follows

$$Y^{ichg} = Y^{new}_{fstart} \mid\mid Y^{old}_{istop}$$

Since we have

$$[Y^{old}_{istop}] \subseteq [Y^{old}_{dstop}] \Rightarrow [Y_{ichg}] \subseteq [Y_{ref}]$$

which means Immediate Change can cause loss.

For Disorder, we prove a general proposition and will use it other proofs.

Proposition 8.2.2 *The append operator ($\mid\mid$) always enforces the No Query-Level Disorder Guarantee.*

Proof The property of output of the $\mid\mid$ operator:

$$\begin{gathered}\forall y_j^{chg}, y_{j'}^{chg} \in \ Y^{chg} : \\ org(y_j^{chg}) \in Y^{new} \wedge org(y_{j'}^{chg}) \in Y^{old} \\ \rightarrow \ j > j'\end{gathered}$$

inherently contrasts with possibility of Query-Level Disorder:

$$\begin{gathered}\exists y_j^{chg}, y_{j'}^{chg} \in \ Y^{chg} : \\ org(y_j^{chg}) \in Y^{new} \wedge org(y_{j'}^{chg}) \in Y^{old} \\ \wedge \ j < j'\end{gathered}$$

For Duplicates, we first define a property which will later be used in the proofs.

Disjointness: An output stream of change policy has the Disjointness property, if its elements are exclusively dependent on input elements before x_{change} or elements after it. Thus, following statement holds:

$$\begin{gathered}\nexists \ y_j^{chg} \in Y^{chg} : \\ \exists x_i, x_{i\prime} \in X : \\ (x_i, x_{i\prime} \in depends_{new}(org(y_j^{chg})) \\ \vee \ x_i, x_{i\prime} \in depends_{old}(org(y_j^{chg}))) \\ \wedge \ i < change < i\prime\end{gathered}$$

Proposition 8.2.3 *Immediate Change guarantees No-Duplicate.*

Proof Since it implies the Disjointness property on the output stream and it is guarantees no-duplicate behavior.

Proposition 8.2.4 *Delayed Drain Change cannot avoid Loss.*

Proof definition of output stream of Delayed Drain Change is as follows:

$$Y^{ddchg} = Y_{fstart}^{new} || \ Y_{dstop}^{old}$$

$$= \left\{ y_j^{new} \in Y^{new} | \ \forall x_i \ \in depends_{new}(y_j^{new}), \ i \ > ? \ \right\}$$

$$|| \left\{ y_j^{old} \in Y^{old} | \ \exists x_i \ \in \ depends_{old}(y_j^{old}), \ i \ < \ ddchange \ \right\}$$

where $'?'$ denotes the fact that the real position of the FStart element which will be inserted is not known in advance.

The fact that applying the FStart element is delayed can cause Loss. More concretely, all those output items from the new query which would been produced by input tuples between the initial position of the DDChange and the effective position of FStart are lost.

For Duplicates, we first define a property which will later be used in the proofs.

Exclusive Contribution: An input stream has this property, if it has an particular element -say x_e- for which all of its elements arriving before x_e only contribute to output elements taken from the old query's output and also those arriving after x_e only contribute to output elements taken from the new query's output.

$$\begin{gathered}
\nexists\, x_i \in X : \\
\exists y_j^{chg}, y_{j'}^{chg} \in Y^{chg} : \\
org(y_j^{chg}) \in Y^{old} \wedge\ org(y_{j'}^{chg}) \in Y^{new} \\
\wedge\ org(y_j^{chg}) \in contributes_{old}(x_i) \\
\wedge\ org(y_{j'}^{chg}) \in contributes_{new}(x_i)
\end{gathered}$$

Proposition 8.2.5 *Delayed Drain Change guarantees No-Duplicate.*

Proof Since it assures the Exclusive Contribution usage of input stream, hence it is free of duplicates. Notice that in Delayed Drain change, the values of e (in x_e) is same as the index of x_{fstart}.

Proposition 8.2.6 *Drain Change is lossless.*

Proof It is easily noticeable that reference stream is defined in the same way that we define output stream for Drain Change. Hence Drain Change (both variation, SDChange and QDChange) is always lossless. It should be noted that although the Interleaving operator (|||) differs from the Append operator (||) it terms of ordering the output items, but they are identical with regard to the number of output *tuples*.

Proof of disorder subsumed by **LDD** Rule's proof.

Proposition 8.2.7 *Drain Change guarantees No-Duplicate.*

Proof We show that the difference between dependency set of any arbitrary element of Y^{dchg} taken from the new query's output and any other arbitrary element of Y^{dchg} taken from the old query's output is alway non-empty.

$$\begin{gathered}
\forall\ y_j^{dchg}, y_{j\prime}^{dchg} \in Y^{dchg} \\
org(y_j^{dchg}) \in Y^{old} \wedge\ org(y_{j\prime}^{dchg}) \in Y^{new} : \\
\exists x_i \in dependes_{old}(org(y_j^{dchg})) : i < dchg \\
\wedge \\
\forall x_i \in depends_{new}(org(y_{j\prime}^{dchg})) : i > dchg \\
\Rightarrow dependes_{old}(org(y_j^{dchg})) - depends_{new}(org(y_{j\prime}^{dchg})) \neq \emptyset
\end{gathered}$$

This clearly shows that there are no two items in the output of Drain Change with the exact same dependency set. Hence, there would be no duplicate introduced by the Drain Change in the in the output stream.

Proposition 8.2.8 *Graceful Immediate Change can be lossy.*

Proof GIChange is ideal for cases in which the new query, compared to the old one, has very large dependency sets (i.e. big window sizes). However, if this Change variation is applied in the opposite situation, due to the fact it uses IStop, it discards incomplete outputs from the old query and results in loss.

Proposition 8.2.9 *Graceful Immediate Change preserves the stream-level order.*

Proof This claim holds because in GIChange, the IStop control element discards those unfinished output elements from the old that can potentially violate the stream-level order. In other words, in case of GIChange *Append* operator (||) behaves like a lossy *Interleave* operator (|||).

Proposition 8.2.10 *Graceful Immediate Change guarantees preserves the query-level order.*

Proof This proof is trivial if we recall that in GIChange the old query keeps on generating outputs until the first output of the new query is produced. Once the this happens, all unfinished outputs of the old query are discarded. Thereby, the query-level order is always preserved.

Proposition 8.2.11 *Graceful Immediate Change is duplicate-free.*

Proof According to the definition of the output stream in GIChange, all output elements originated from the new query have at least one contributing input element which is positioned after the IStop control element. Since all contributing input elements of all output elements originated from the old query happen before the IStop control element, there can be possibly no duplicates.

Proposition 8.2.12 *Graceful Drain Change is lossless.*

Proof This is always true because the output stream of GDChange subsumes that of DChange (which is the reference stream). More precisely, both share the exact same number of output elements from the new query but GDChange applies the DStop control element a later position than DChange.

Proposition 8.2.13 *Graceful Drain Change guarantee preserves the stream-level order.*

Proof As pointed out before, using the *Interleave* operator (|||) inherently preserves the stream-level order.

Proposition 8.2.14 *Graceful Drain Change may cause query-level disorder.*

Proof Given the fact that GDChange guarantees both loss and stream-order guarantees, the above claim can directly deduced from the LDD rule.

Proposition 8.2.15 *Graceful Drain Change does not guarantee no-duplicates.*

Proof We prove this by giving an example. Imagine a case that the old query has an output element with the exact same dependency set as the first output element of the new query. Since GDChange uses DStop to stop the old query, both of these elements will be included in the output stream of GDChange and thereby it will have duplicates.

Termination Requirements

Termination of the old query is driven by two parameters: 1) the type of stop control element used to implement change, and 2) whether the start position of the new query is statically determined or not. Immediate Stop terminates a query immediately whereas

Drain Stop lingers the termination while waiting for the “right data”(e.g. closing condition for a window operator) to arrive, and thus it is *data dependent* and statically incomputable. Consequently, DDChange, SDChange, QDChange, GDChange (since they all use DStop), and GIChange (because the position of IStop element is dependent on the behavior of the new query) cannot guarantee termination.

Progress Requirements

Progress of the new query depends on whether the start position of the new query is statically determined or not. Since in all IChange, SDChange, QDChange, GIChange, and GDChange the position of the FStart element is statically determined (the position of the change control element itself), the all guaranteed to make progress. Delayed Drain Change does not provide this guarantee since the position of the FStart control element is dependent on the old query’s termination, which is not guaranteed itself.

8.2.6 Correctness Rules for Change

In addition to covering the design space for change implementation, we investigated the correctness guarantee space. We have observed some common patterns and compiled them into a set of rules. These rules help us determine that we indeed provide the strongest possible guarantees.

LDD Rule

Proposition 8.2.16 *The* **LDD** *rule states that in the general case, a Change policy can ensure at most two out of three of the following guarantees: No* **L***oss, No Stream-level* **D***isorder, No Query-level* **D***isorder.*

Proof In order to prove this rule, we split it into three separate cases; however, without limiting the generality, throughout this set of proofs, we focus on the following example (formal depiction of Figure 8.3; it basically means the last window of the old query fully encompasses one of the new query’s windows):

$$\begin{gathered}\exists y_j^{chg}, y_{j\prime}^{chg} \in Y^{chg} : \\ org(y_j^{chg}) \in Y^{new}, org(y_{j\prime}^{chg}) \in Y^{old} \\ \wedge \\ min(indexOf(depends_{new}(org(y_j^{chg})))) <\end{gathered}$$

$$min(indexOf(depends_{old}(org(y_{j'}^{chg}))))$$
$$\wedge$$
$$max(indexOf(depends_{new}(org(y_{j}^{chg})))) >$$

$$max(indexOf(depends_{old}(org(y_{j'}^{chg}))))$$

- Loss and Query-Level Disorder are guaranteed: by doing so, y_j^{chg} appears prior to $y_{j'}^{chg}$ and that means, Stream Level Disorder is violated
- Loss and Stream-Level Disorder are guaranteed: analogous to the previous one
- Query and Stream-Level Disorder are guaranteed: this means either y_j^{chg} or $y_{j'}^{chg}$ should be excluded from the output stream, resulting in Loss in both ways.

LT Rule

Proposition 8.2.17 *The* **LT** *rule indicates that No* **L***ossless Change policy can guarantee* **T***ermination.*

Proof The proof is straightforward. In order for a Change policy to be Lossless, it has to use the *Drain Stop* control element, and this type of Stop element does not guarantee the termination by its very definition.

These rules show us that we have indeed covered all possible options for change when considering strong guarantees (at least two out of LDD and no duplicates, as well as termination where possible). This can be seen by comparing Table 8.2 and Table 8.3 with the set of all possible combinations of correctness guarantees. Other, new options will only reduce guarantees, and these reductions are typically not useful (e.g., having no order at all, or one kind of disorder with loss). Thus we have shown that our methods to express change cover the relevant problem space and cannot be improved further for the general case.

8.3 Implementation

Here, we first extend the general architecture that we proposed in previous chapter (Section 7.5). Then we show how to incorporate this architecture into existing stream processing engines.

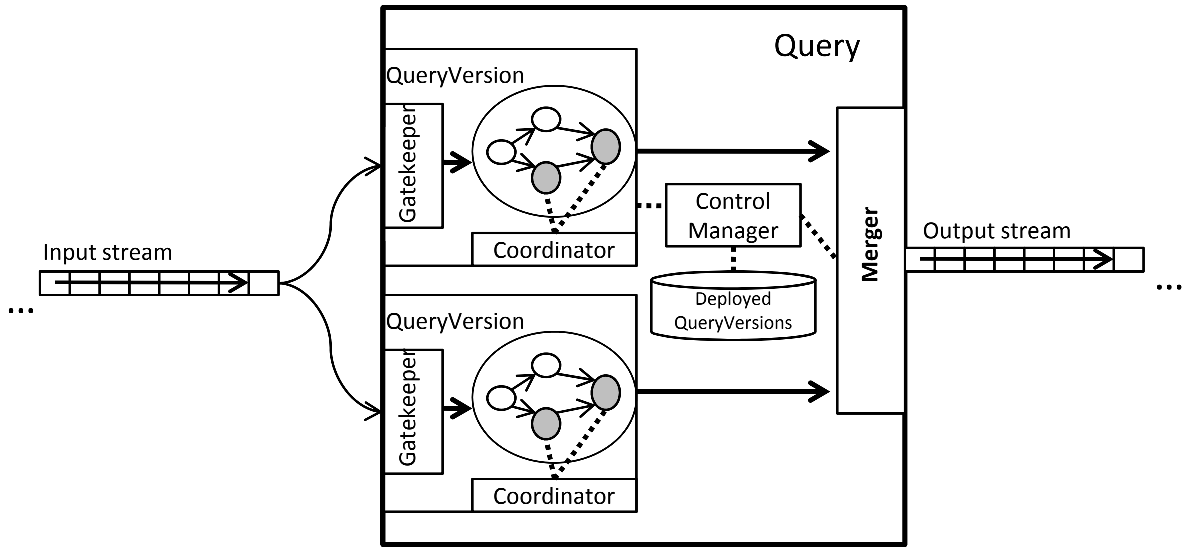


Figure 8.10: *Query Modification architecture*

8.3.1 General Architecture Revisited

For an SPE architecture to support query modification, it must be ensured that the system keeps track of multiple versions of a given query, and executes them in a coordinated way during the change period, taking the chosen change policy and its correctness guarantees into account. To this end, we enrich our proposed architectural extension from Chapter 7. The new extension is shown in Figure 8.10.In the following, we describe the new components (others are previously explained in Section 7.5).

Query Versions: An SPE will keep track of each individual *Query*, which in turn consists of a set of *Query Versions*. *Query Versions* are stored in a *Query Version Repository*, possibly in an already validated/compiled/optimized form, so as to avoid potential errors and minimize overhead during the actual query modification period. These versions can be added or removed from the repository when not required. Each *Query Version* uses its own *Gatekeeper* and *Coordinator*, and the whole repository shares a common *ControlManager* and a *Merger*.

Merger: It combines the output of both query versions into a single stream. The key task of the Merger is to establish the correct delivery order over the two streams. For IChange, DDChange, and QDChange, this is straight-forward, since all output of Q^{old} will be produced before that of Q^{new}. For SDChange and GChange, additional order-related metadata (e.g., starting index) needs to be known for each stream element. This component can be seen as the implementation of the *Append* (||), and *Interleave* (|||)

operators.

8.3.2 SPE-specific Implementation

We implemented our architecture on MXQuery [29] and also studied how an implementation on top of Borealis [15] could be done. Given the differences in the data model, operator semantics and execution strategies, this should provide a good coverage of the SPEs space.

MXQuery is an implementation of XQuery 3.0, which has few implicit assumptions on the data model (sequences of semi-structured items), expressive predicate-based windows and a set of fully composable, Turing-complete expressions. It uses a classical DBMS-style pull model, which requires explicit threading for parallel query execution, yet simplifies output control and merging, since the output is always explicitly requested, and the end of available output is explicitly indicated.

Determining the relative *depends* set for two items out of different versions (as needed by the $|||$ (Interleave) operation, and thus the Merger) is conceptually difficult, given the flexible data model, the data creation operations and the large number of operators. Due to the lazy execution strategy of MXQuery (which ensures that only required data is read from the input), observing the Gatekeeper before requesting the next element gives this information in a very lightweight way, thus not requiring to change the data model and instrument the operators.

Borealis, on the other hand, uses relational tuples in combination with a small number of streaming operators, and count- and time-based windows. Push-based operators are connected by queues and manually composed to form a query network. A scheduler can decide how to prioritize certain operators. This form of coupling simplifies parallel execution, but makes it harder to determine when all output for a given input has been produced, so that a switch can be performed. This limitation can be overcome by instructing the scheduler to prefer operators which are connected to a closed or draining Gatekeeper.

Computing the *depends* set which is required by the Merger to impose the right order between outgoing tuples is simple since aggregates on windows will assign the maximum contributing timestamp to the produced tuple.

8.4 Experiments

We ran two sets of experiments on top of MXQuery [29]: (1) A synthetic data/query set to perform a sensitivity analysis for stateful operators. (2) A Linear Road Benchmark [23] query to study the impact of change on complex queries.

All of our experiments were performed on a system with an Intel Core2 Duo, 2.66 Ghz, 4 GB RAM, running Windows 7, Java 6 (both 32 bit).

8.4.1 Sensitivity Analysis for Stateful Operators

Our sensitivity analysis focuses on the behavior of stateful operators, since stateless operators will have a negligible effect on change performance. We study the impact of *window size*, and *window slide* on *response time*, *correctness criteria*, and *CPU overhead.*

The query in this set of experiments is shown in Listing 8.1. It computes a sum over count-based windows. The value of *WindowSize* is the parameter that we vary in this experiments. The input data consists of a sequence of 2000 XML elements containing an integer payload, which are fed to the system as fast as it could consume it. The change control element is inserted after 1000 data items, ensuring that the system has reached a steady state in terms of open windows and also has enough input to complete the change.

```
declare variable $input external;

for sliding window $w in $input//seq
  start $first at $s when fn:true()
  only end $last at $e when $e − $s = WindowSize
return <sum>{sum($w)}</sum>
```

Listing 8.1: *The Template Query Used in the Sensitivity Analysis Experiments*

All measurements were repeated 100 times. For performance we took the averages, for correctness we checked across all these runs that we always saw the same results. Since standard deviation on all results was small, we do not report it explicitly.

Response Time

In the first experiment, we vary the window size of Q^{new} between 10 and 100 elements, while keeping that of the Q^{old} at 50. Both queries are using a slide of 1, providing a significant overlap amount the windows.

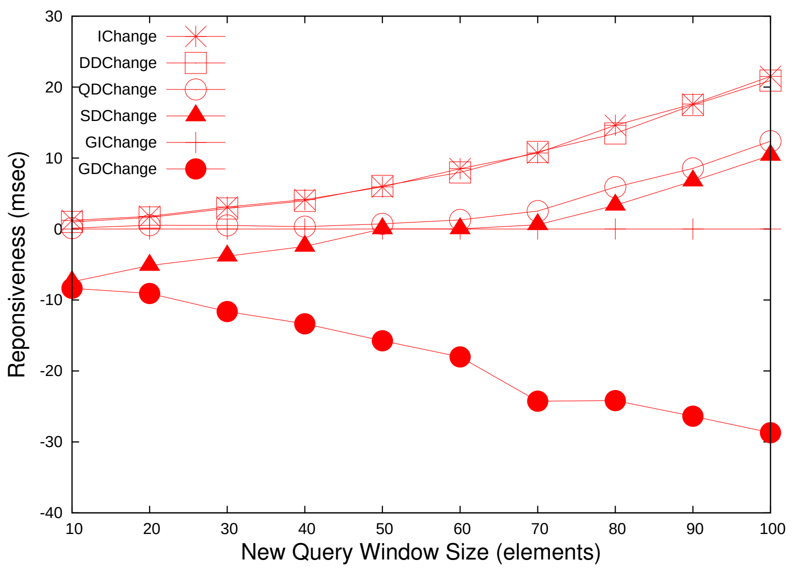


Figure 8.11: *Responsiveness on Window Size, Slide=1*

As Figure 8.11 shows, the different impact of window size on the response times (time between the last element of Q^{old} and the first element of Q^{new}) is apparent for the various methods: For *IChange* and *DChange*, the response time is linear to the size of the new window (from 1.7 msec at WS=10 to 22.6 msec at WS=100), as processing of Q^{new} only starts when Q^{old} has ended, and the processing time is proportional to the number of input items in a window.

For *QDChange*, the response time is 0.2msec for window sizes of Q^{new} that are smaller than or equal to 50, since the output of Q^{new} would have been produced earlier, and needs to be held up until Q^{old} finishes. Once Q^{new} has window sizes bigger than that of Q^{old}, the same trend as for IChange is visible, because now the size of the new window dominates. The additional cost of synchronization between Q^{old} and Q^{new} causes response times to increase at a relatively faster pace. For *SDChange*, smaller windows of Q^{new} mean that the output of Q^{new} needs to be produced before the output of Q^{old}, yielding a negative response time for the smaller values, e.g. -7 msec for WS=10. As the window size of Q^{new} increases, the response time increases, showing values similar to *QDChange* for WS greater than 50.

GIChange shows nearly-ideal response times (3 microseconds), since Q^{old} is kept producing until Q^{new} can produce output, then it is terminated immediately. *GDChange* uses the same approach, but drains Q^{old}, thus showing a “negative” response time of around 10 msec, slightly more than cost of producing windows of Q^{old}, as the two queries run in parallel and need to be coordinated.

Correctness

We measure the violation of correctness properties by creating the reference streams according to the definition in Section 8.2.2 and compare the outputs against it. Figure 8.12 (a) shows that there is constant amount of loss (49 expected elements) for *IChange* and *DDChange*, corresponding to the loss of a complete Q^{old} window until Q^{new} picks up. *QDChange*, *SDChange*, and *GDChange* do not show any loss, since the draining of Q^{old} and the starting of Q^{new} are balanced to avoid this. *GIChange* has loss proportional to the size difference of the new window and old window, since Q^{old} receives an *IStop* as soon as the first output of Q^{new} is available, discarding the last window of Q^{old}.

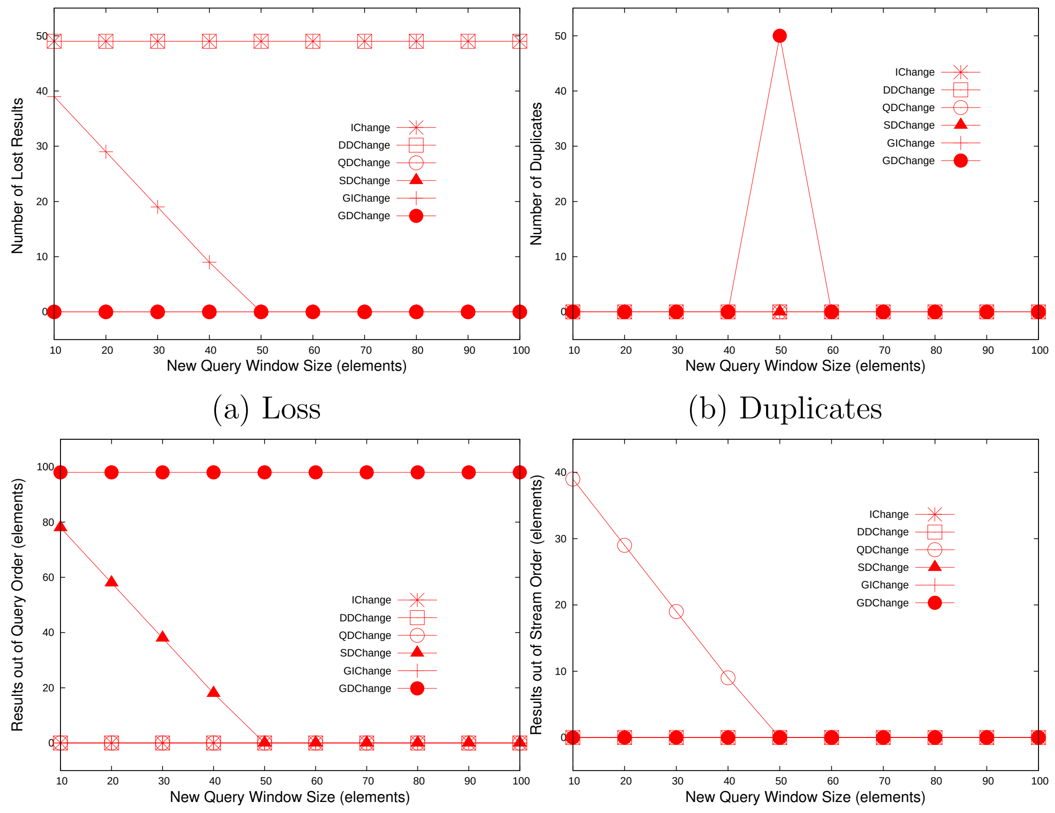
For disorder (Figures 8.12 (c) and (d)) we also see the expected results: *IChange*, *DDChange*, and *GIChange* never cause any disorder, since no overlapping results are produced. *QDChange* produces results out of stream order if the window size of Q^{new} is smaller. The size of disorderedness is proportional to the difference in window size (e.g., 39 at WS=10), since as many "smaller" windows are produced (due to the slide of 1) before the completion of Q^{old} and need to be delayed to maintain query order. In turn, *SDChange* shows the same behavior in respect to items out of query order, while *GDChange* has has a number proportional to the window size of Q^{old}, as it drains it after the start of Q^{new}.

As shown in Figure 8.12 (b), we see duplicates only in the case where the window of size of Q^{new} is the same as that of Q^{old} and the change method is *GDChange*. Nonetheless, it should be noted that both *GIChange* and *GDChange* produce additional results (with different inputs), both by the overlap of mapping functions and the output produced by Q^{old} while "waiting" for output from Q^{new} (which is not part of the reference stream).

CPU Overhead

The different change methods also have a different runtime overhead. In our measurements, we focused on the CPU cost, since the actual memory overhead heavily depends on how an SPE supports the sharing of items, queues, etc.

For all methods, we measured the CPU time of the main thread over the whole experiment execution as well as the use of any helper threads required to perform parallel query execution. The cost of the main thread is almost the same for all methods. Thus, we just show the results for the helper thread in Figure 8.13 (measured in milliseconds of CPU time). Since *IChange* and *DDChange* do not execute both versions in parallel, there is obviously no cost for them. *SDChange* and *QDChange* always produce the reference output, so the relative cost stays the same. For *GIChange* and *GDChange*, additional output of Q^{old} is produced while waiting for output of Q^{new}, therefore we see a higher



(a) Loss (b) Duplicates

(c) Query-Level Disorder (d) Stream-Level Disorder

Figure 8.12: *Correctness Results for Different Changes*

cost in general and an increase with the window size of Q^{new}. In our implementation, *GIChange* is slightly more expensive, since it computes output that might be discarded, while *GDChange* avoids that.

We also performed tests with different window slide sizes and different relative positions of the change elements. The results showed the expected results. The bigger the slide and thus the smaller the overlap, the fewer correctness problems occur. If there are only tumbling windows, placing the change at the window change resulted in error-free results for all methods.

8.4.2 Complex Queries

We also tested the different change methods on the Linear Road Benchmark [23] workload, namely the Accident Segment Query (The prologue of this query is shown in Listing 8.2. The full query is included in Appendix A). There are several differences compared to the

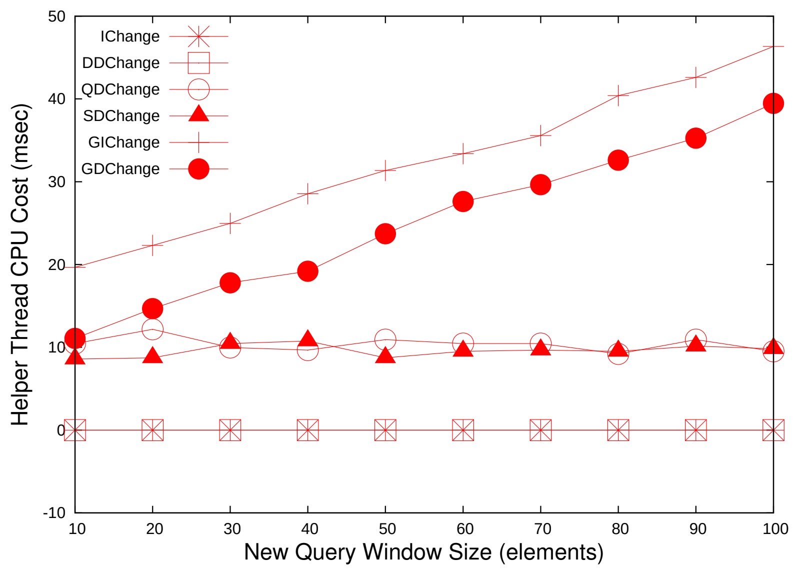


Figure 8.13: *CPU Cost on Window Size, Slide=1*

synthetic workload: 1) The query is significantly more complex, using multiple nested windows with predicates, grouping inside windows and parallel aggregation. 2) Data arrives using a specific timing 3) Results are expected within 5 seconds. Yet the observed results closely mirrored what we had seen on the synthetic data, with a slightly bigger impact on the arrival timing and window slide on the delays.

```
declare  variable  $ReportedCarPositionsSeq external;

forseq  $w in $ReportedCarPositionsSeq early sliding window
start  curItem $s_curr,  prevItem $s_prev when $s_curr/@minute ne
  $s_prev/@minute
end curItem $e_curr,  nextItem $e_next when ($s_curr/@minute + DIFF) eq
  ($e_next/@minute)
    let  $currMin := fn:ceiling($e_curr/@minute)
    let  $stopedCars :=
            for  $rep  in  $w
            group $rep as $r−group by $rep/@VID as $vid_s, $rep/@XWay as
              $xway_s, $rep/@Seg as $seg_s, $rep/@Dir as $dir_s,  $rep/@Lane
              as  $lane_s ,  $rep/@Pos as $pos_s
            where count($r−group) ge DIFF∗2
             ...
```

Listing 8.2: *The Accident Segments Query from the LR Benchmark [23]*

Q^{old} of the accident query defines a sliding window with a size of 120s and a slide of 30s, producing output every 30 seconds, shown as grey bars in Figure 8.14. For Q^{new}, the window size was varied from 60 seconds to 360 seconds (by changing the value of `DIFF` in

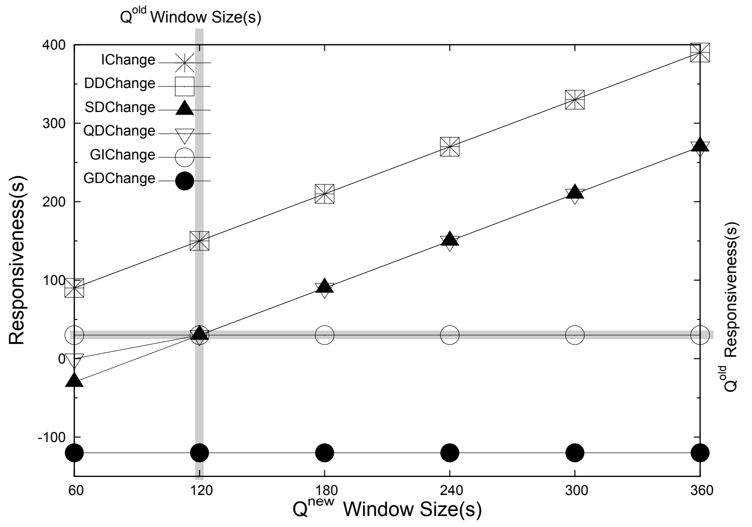


Figure 8.14: *Linear Road Change Responsiveness*

Listing 8.2). There is an increasing response time for all methods not using the graceful approach, since the waiting time for result of Q^{new} increases with increasing window size, while Q^{old} terminates after a fixed time.

The graceful approaches (*GIChange*, *GDChange*) overcome this problem by allowing Q^{old} producing until there is result for Q^{new}, thus yielding a stable response time. In particular, *IChange* and *DDChange* show the highest response time: WS(Q^{new}) + 30 seconds (e.g. Q^{new} 150 seconds at WS 120), since Q^{new} is only started when Q^{old} has terminated. The additional 30 seconds are required to find the next start condition for a window.

QDChange and *SDChange* improve the responsiveness by starting Q^{new} already at the change, thus yielding a response time of WS(Q^{new}) - WS(Q^{old}) + 30 sec (as to open the new window). Once this difference becomes smaller than 0, the two methods differ, since a particular order needs to be established: *QOChange* needs to delay the output of Q^{new} until the last output of Q^{old} has been produced, therefore emitting (almost) at the same time, and out of order to their window specifications. *SDChange* produces the first output of Q^{new} 30s before the last output of Q^{old}, thus creating query-level disorder.

GIChange stops Q^{old} once output for Q^{new} is available (as to avoid duplicates), so there is a constant 30s difference. *GDChange* drains Q^{old} when the first output of Q^{new} is available, as to avoid loss, leading to an additional output with WS(Q^{old}).

Important	Irrelevant	Method
Runtime Resources, Implementation Overhead	Loss, Response Time	IChange
Loss, Query Order	Response Time	QDChange
Loss, Stream Order	Response Time	SDChange
Response Time, Duplicates	Runtime Resources, Implementation Overhead, Loss	GIChange
Response Time, Loss	Runtime Resources, Implementation Overhead, Duplicates, Order	GDChange

Table 8.4: *Change Method Decision Matrix*

8.4.3 Tradeoffs and Guidelines

The results of the conceptual as well as the experimental analysis give a fairly clear answer to when to use which change method, given that there cannot be a single winner which guarantees all desirable properties. We have summarized the tradeoffs in Table 8.4.

Generally speaking, achieving zero loss and low response time incur additional implementation complexity and runtime overhead. So if neither of them is required, using *IChange* is a reasonable choice. Example use cases include sensor networks (due to limited resources) or complex query processors (due to the implementation effort). When loss must be avoided, but response time is less critical, *QDChange* and *SDChange* are the most suitable change variations. The order that is expected during the change then determines which of these two change methods to use.

Finally, Graceful Changes address Response Time, trading it off with higher resource usage. Among them, *GIChange* should be chosen if loss is tolerable, while *GDChange* should be preferred if it is not tolerable. *DDChange* is suitable only in very rare circumstances, since it does not provide stronger guarantees than IChange and requires drain support, In addition, it does not always guarantee the same results as the other change methods on Q^{new}, since the *start position* of the new mapping function is set at the end of the drain area, and not at the change element position as in all other approaches.

8.5 Related Work

The basic vision for dynamic modification of continuous queries (CQ) was first put forth by the Borealis project [15]. The Borealis approach, however, is much more restricted: It focuses on specific operators (such as windows) with specific changes (slide or size) instead of allowing arbitrary changes on queries. No formal semantics of change are given, and the architecture ties its strategies to system time and execution speed. To our knowledge, this approach has never been implemented.

Query modification shares a lot of challenges and solutions with other lifecycle problems in CQ, namely failure handling [59] and plan migration [91, 90]. A fundamental difference is that all of these approaches try to maintain a semantically unchanged query over changes of the infrastructure or execution plan, and change is driven by the system. In our context, change can also be triggered by the application, and therefore, we do not make assumptions about the timing and semantics of the new query.

In a similar spirit, the extensive work on adaptive query processing [41] targets a subset of the problem we are solving: No matter how the actual query execution is modified, the semantics of the query stay the same, and all correctness guarantees must be supported. Our formal framework can describe the behavior of such a system quite well, e.g. using stop and start to mark the boundaries of an adaptation.

Application-driven CQ changes are proposed by Lindeberg et al. [69], who investigate changing window sizes in order to improve results of a health monitoring use case. In contrast to our model, this model is very restricted in terms of the allowed query modifications (size change for tumbling windows) and use cases (heart attack prediction), and provides no formal correctness guarantees.

Finally, our work also relates to stopping and restarting of long-running queries in data warehouses [65, 30, 32]. In this case, some queries are intentionally terminated and later restarted to deal with resource contention. The restart should reuse some of the old state for efficiency reasons. Stopping and restarting the same query constitutes a special case in our more general framework. Furthermore, streaming has different semantic requirements than traditional warehouses, e.g., ordered data delivery.

8.6 Conclusions

In this chapter, we exploited our modeling framework and methodology from the previous chapter and modeled continuous query modification. In the first step, we defined the

Reference Output Stream for query modification. Based on this definition, we formally specified correctness criteria. We then introduced the Change control element which is, in essence, the combination of a Stop control element (for the old version of the query) and a Start control element (for the new version).

As our last step in the modeling phase, we not only identified the guarantee sets provided by variation of Change control element, but also explored the full guarantee space for query modification which resulted in a set of general rules. An important implication of these rules is that in general case there are inherent incompatibilities between different guarantees and there needs to make tradeoffs.

On the practical side, we extended our proposed general architecture from Chapter 7 to support query modification. We also showed that an implementation of this framework is possible on typical SPEs, without requiring much effort or fundamental changes on the existing implementation.

Benchmark results on our prototype implementation on top of MXQuery [10] clearly highlight the practical aspects and performance/correctness tradeoffs among our query modification techniques.

As an avenue for future work, we plan to optimize modification performance by exploiting query knowledge. For example, by identifying common subexpressions across versions in order to minimize change cost. Additionally, since the guarantee rules that we presented in this work make no assumptions, another direction for future work is to take into the consideration the constraints of different stream processing models and refine these rules for each case.

Chapter 9

Conclusion

With the proliferation of dynamically generated data, we have witnessed the emergence of data stream processing as a new data management paradigm. Many Stream Processing Engine (SPE) prototypes have been built and several business applications have started to use data stream management systems.

However, as industry gears toward using data stream processing in complex applications, there are challenges which still need to be addressed. In the following (in Section 9.1), we summarize the main contributions this thesis has made towards making data stream processing a viable and full-blown solution for those complex applications. We then conclude this thesis by discussing ongoing and future work in Section 9.2.

9.1 Summary

In this thesis, motivated by our real-world usecase, MASTER project which aims at solving the compliance monitoring problem in Service Oriented Architecture (SOA), we have made three main contributions to extend the capabilities of data stream processing paradigm. Below, we give a brief summary for each of these contributions and also point out how they have been integrated into the MASTER deliverables.

1. Stream Schema. Metadata specifying structural and semantic constraints, are invaluable in data management. In order to capture and exploit this type of metadata for data stream processing, we have presented Stream Schema, a framework which provides not only precise definitions for individual constraints, but also a well-defined scheme to compose them. This allows recursive and comprehensive description of data streams . Furthermore, we have presented a mechanism for validation of data streams against Stream Schema and analyzed its complexity.

Having established Stream Schema, we have explored its applications, starting with query optimization. Stream Schema can be used to safely pipeline the query execution and partition the data. It can also enable state reduction and query rewrites. Static analysis of stream queries is another area that can benefit from Stream Schema. There it can extend the set of runnable as well as non-executable expressions. Finally, due to the fact that Stream Schema captures data consistency and structure constraints, it can greatly simplify the queries in streaming applications resulting in increased decoupling and reuse.

In the MASTER prototype implementation, queries (for instance in performance study reported in Deliverable D4.2.3 [6]) were manually tuned to take advantage of schema knowledge.

2. Stream Provenance Management. Tracking provenance, exploring which input data led to a given query result, has proven to be an important functionality in many domains. To track provenance on data streams, we have first presented an algebra to define fine-grained stream provenance. Then, we have investigated different possibilities to generate, store, and retrieve the provenance information, highlighting advantages and shortcomings of each of them . We have built Ariadne, a provenance-aware SPE, that instruments operators to generated and propagate provenance. Several optimizations to reduce the computational and storage overhead of provenance management have been identified and implemented. Through our comprehensive experimental study, we have shown the feasibility of fine-grained provenance support and also explored the tradeoffs.

In terms of provenance support in the MASTER prototype, we have provided a lightweight implementation of our proposal [78].

3. Continuous Query Lifecycle Model. Modeling lifecycle operations (such as start, stop, and modification) on continuous query is an essential part of defining precise semantics for data stream processing systems. We have established a framework that can formally express arbitrary lifecycle operations on the basis of input-output mappings and basic control elements (such as query start or query stop) which are inserted into the input stream. Moreover, our framework allows precise specification of correctness criteria, a mechanism to evaluate outcomes of lifecycle operations. We have also devised a bottom-up methodology to model complex lifecycle operations using basic control elements.

We then leveraged this framework and methodology to model query modification, an important complex lifecyle operation which had not been modeled before. We derived all possible variations of query modification, each providing different levels of correctness guarantees and performance. Experiments identify the key performance tradeoffs of the query modification variations.

Deliverable D4.2.2 [5] explains how this model has been integrated into MASTER.

9.2 Ongoing and Future Work

Below, we outline several interesting topics of ongoing work and discuss possibilities for future research.

Stream Provenance Computation using Query Rewrite. As pointed out in Chapter 6, provenance computation through query rewrite has some limitations and it is not applicable to all use cases. However, understanding the cost of such rewrites under the circumstances which they are applicable, is worth investigating. To this end, we plan to adapt the query rewrite techniques from PERM [51] for stream queries and then empirically compare its performance to the operator instrumentation method implemented in Ariadne.

Lifecycle Model for non-Linear Queries. Our continuous query modeling framework focused on linear query networks (compositions of single-input single-out operators). We plan to extend our abstraction and general framework to support multiple-input stream queries (i.e., join and union). Our general approach is to assume total order among stream element on all streams and use only one of the input streams to carry control elements. Furthermore, our mapping functions need to be extended to abstract away the logic of non-linear operators.

As mentioned in Chapter 2, there are other research challenges within the context of MASTER which have not been addressed in this thesis. In following, we point out two of such problems. It is worth noting that these problems exist in other domains and are not MASTER-specific.

Extremely Long-running Queries. Control objectives specify goals that must hold over years, sometimes even decades. For instance to achieve an objective like:

```
Files containing financial data need to be kept for 10 years
```

there needs to be a monitoring query on *file* operations which either ends after ten years or indicates deletion attempts. Any solution to this problem needs to deal with issues like recoverability, efficient state management/reduction, and responsiveness after *sleeping* for years.

Model and Language Transformation. Monitoring queries primarily stem from higher-level, formal description of goals in form of temporal logic or process descriptions. The gap between these descriptions and the monitoring language should be filled by well-defined transformations. Such transformations allow verification of the queries against higher-level models and also automatic/guided compilation of high-level goals into queries. There can be also potential optimizations by matching expressiveness of monitoring query language with required constructs.

Appendix A

Queries in MXQuery Implementation of LR Benchmark

As shown in Figure5.2, nine continuous queries have been used to realize the LR Benchmark queries.

Q1: Car Position Report Filtering

```
declare  variable  $InputSeq external;

for  $w in $InputSeq
where $w/@Type eq 0 return $w
```

Listing A.1: *Q1: Car Positions*

Q2: Accident Segments

```
declare  variable  $ReportedCarPositionsSeq external;

forseq  $w in $ReportedCarPositionsSeq early sliding window
start  curItem $s_curr,  prevItem $s_prev when $s_curr/@minute ne
  $s_prev/@minute
end curItem $e_curr,  nextItem $e_next when ($s_curr/@minute +2) eq
  ($e_next/@minute)
    let  $currMin := fn:ceiling($e_curr/@minute)
    let  $stopedCars :=
            for  $rep in  $w
            group $rep as $r−group by $rep/@VID as $vid_s, $rep/@XWay as
```

```
                  $xway_s, $rep/@Seg as $seg_s, $rep/@Dir as $dir_s, $rep/@Lane
                  as $lane_s, $rep/@Pos as $pos_s
                where count($r−group) ge 4
                return <stopped_car VID="{$vid_s}" XWay="{$xway_s}" Seg="{$seg_s}"
                  Dir="{$dir_s}" Lane="{$lane_s}" Pos="{$pos_s}"></stopped_car>
                let $accidents :=
                  for $car in $stopedCars
                  group $car as $c−group by $car/@XWay as $xway_a, $car/@Seg as
                    $seg_a, $car/@Dir as $dir_a, $car/@Lane as $lane_a, $car/@Pos as $pos_a
                    where count($c−group) ge 2
                  return <accident minute="{$currMin}" XWay="{$xway_a}" Seg="{$seg_a}"
                  Dir="{$dir_a}"></accident> (: Pos="{$pos_a}" :)
        let $accidentsRes := if ( count($accidents) gt 0 ) then <accidents>
                                    {$accidents} </accidents>
                                                else <accidents><accident minute="{$currMin}" XWay="−1"
                                                  Seg="−1" Dir="−1"></accident></accidents>
return $accidentsRes
```

Listing A.2: *Q2: Accident Segments*

Q3: Car Positions to Respond

```
declare variable $ReportedCarPositionsSeq external;
declare variable $CAR_POS_STORAGE external;

for $item in $ReportedCarPositionsSeq

let $prevCarRep := lr:store("CAR_POS_STORAGE", (@VID eq $item/@VID) )

return
(    if ( count($prevCarRep) eq 0 or ($prevCarRep/@Seg ne $item/@Seg and
    $item/@Lane ne 4) ) then $item
  else (),
    lr:store−update("CAR_POS_STORAGE", $item, (@VID eq $item/@VID))
)
```

Listing A.3: *Q3: Car Positions to Respond*

Q4: Segment Statistics for Every Minute

```
declare variable $ReportedCarPositionsSeq external;

```

```
forseq $w in $ReportedCarPositionsSeq early tumbling window
    start curItem $s_curr, prevItem $s_prev when ( fn:ceiling($s_curr/@minute)
      ne fn:ceiling($s_prev/@minute))
    end nextItem $e_next when ( $s_curr/@minute +1) eq $e_next/@minute
    let $currMin := fn:ceiling($s_curr/@minute)
    let $avgCarSpeed :=
            for $rep in $w
            group $rep as $r−group by $rep/@VID as $vid_a, $rep/@XWay as $xway_
              a, $rep/@Seg as $seg_a, $rep/@Dir as $dir_a
            return
              <res XWay="{$xway_a}" Seg="{$seg_a}" Dir="{$dir_a}" VID="{$vid_a}"
              vAvgSpeed="{avg($r−group/@Speed)}" ></res>
     let $segStatistics := (<res endMark="1" minute="{$currMin}"></res>,
            for $car in $avgCarSpeed
            group $car as $c−group by $car/@XWay as $xway, $car/@Seg as $seg,
              $car/@Dir as $dir
            return
              <res endMark="0" minute="{$currMin}" XWay="{$xway}" Seg="{$seg}"
              Dir="{$dir}" avgSpeed="{avg($c−group/@vAvgSpeed)}" carCount=
              "{count($c−group)}"></res>, <res endMark="3" minute="{$currMin}">
              </res>,<res endMark="3" minute="{$currMin}"></res> )
return $segStatistics
```

Listing A.4: *Q4: Segment Statistics for Every Minute*

Q5: Toll Calculation

```
declare variable $SegmentStatSeq external;
declare variable $ACCIDENT_STORAGE external;

forseq $w in $SegmentStatSeq sliding window
    start prevItem $s_prev when $s_prev/@endMark eq 1
    force end nextItem $e_next when ($e_next/@endMark eq 3) and
      ( ($s_prev/@minute + 4) eq $e_next/@minute )
    let $resMin := $e_next/@minute
    let $allAccSeg := lr:store("ACCIDENT_STORAGE", (@minute eq $resMin) )
    let $segData :=
            for $s in $w
            where $s/@endMark eq 0
            group $s as $s−group by $s/@XWay as $xway, $s/@Seg as $seg,
              $s/@Dir as $dir
            return
                    <res XWay="{$xway}" Seg="{$seg}" Dir="{$dir}" avgSpeed=
                    "{avg($s−group/@avgSpeed)}" carCount="{mxq:empty−to−zero(
                    $s−group[@minute eq $resMin]/@carCount)}"></res>
    let $allAffectedSeg :=
    for $segmCurr in $allAccSeg
            let $segm := $segmCurr/@Seg
    return
            if ($segmCurr/@Dir eq 0)
                    (: eastbound direction :)
                            then <res> <XWay>{data($segmCurr/@XWay)}</XWay><Dir>
                              {data($segmCurr/@Dir)}</Dir><startSeg>{data($segm) − 4}
                              </startSeg><endSeg>{data($segm)}</endSeg> </res>
                    (: westbound direction :)
                            else <res> <XWay>{data($segmCurr/@XWay)}</XWay><Dir>
                              {data($segmCurr/@Dir)}</Dir><startSeg>{data($segm)}</startSeg>
                              <endSeg>{data($segm) + 4}</endSeg> </res>
    let $tollResults :=
            for $sData in $segData
                    let $affSeg :=
                            for $sCurr in $allAffectedSeg
                            where $sCurr/XWay eq $sData/@XWay and $sCurr/Dir eq $sData/@Dir
                              and $sCurr/startSeg le $sData/@Seg and $sCurr/endSeg ge
                              $sData/@Seg
                            return $sCurr
                    let $notInAccidentZone := count($affSeg) eq 0

                    let $lastMinCarCount := $sData/@carCount − 50
                    let $t := if ( $notInAccidentZone and $sData/@avgSpeed < 40 and
                                   $lastMinCarCount > 0 )
                                      then $lastMinCarCount ∗ $lastMinCarCount ∗ 2
                                      else 0
              return <res minute="{$resMin + 1}" XWay="{data($sData/@XWay)}"
                Seg="{data($sData/@Seg)}" Dir="{data($sData/@Dir)}" avgSpeed=
                "{data($sData/@avgSpeed)}" ccount="{data($sData/@carCount)}"
                toll="{$t}"> </res>
return <tolls>{$tollResults}</tolls>
```

Listing A.5: *Q5: Toll Calculation*

Q6: Accident Events

```
declare variable $ACCIDENT_STORAGE external;
declare variable $CAR_POSITIONS_TO_RESPOND external;

```

```
for $s_curr in $CAR_POSITIONS_TO_RESPOND
    let $prevMin := $s_curr/@Time idiv 60
    let $allAccSegOnWay := lr:store("ACCIDENT_STORAGE", (@XWay eq
      $s_curr/@XWay and @Dir eq $s_curr/@Dir and @minute eq $prevMin) )
    let $checkAcc :=
            for $s in $allAccSegOnWay/@Seg
                    let $accOnWay := if ($s_curr/@Dir eq 0)
                            (: eastbound direction :)
                            then if ( ($s −5) lt $s_curr/@Seg and $s_curr/@Seg le $s)
                                 then data($s) else ()
                                 (: westbound direction :)
                                 else if ($s +5 gt $s_curr/@Seg and $s_curr/@Seg ge $s)
                                      then data($s) else ()
            return $accOnWay
    let $accidentAlert := if ( count($checkAcc) gt 0 )
                                then <alert Type="1" Time="{$s_curr/@Time}"
                                  Emit="" VID="{$s_curr/@VID}" Seg=
                                  "{$checkAcc[1]}"></alert>
                                else ()
return $accidentAlert
```

Listing A.6: *Q6: Accident Events*

Q7: Toll Events

```
declare variable $TOLL_STORAGE external;
declare variable $BALANCE_STORAGE external;
declare variable $CAR_POSITIONS_TO_RESPOND external;

for $s_curr in $CAR_POSITIONS_TO_RESPOND
    let $prevMin := ($s_curr/@Time idiv 60) + 1
    let $segToll := lr:store("TOLL_STORAGE", (@Seg eq $s_curr/@Seg
      and @XWay eq $s_curr/@XWay and @Dir eq $s_curr/@Dir and @minute
      eq $prevMin) )
    let $newBal := <res VID="{$s_curr/@VID}" Time="{$s_curr/@Time}"
      Bal="0" Toll="{ mxq:empty−to−zero($segToll/@toll) }"></res>
return
  (
  lr:store−update("BALANCE_STORAGE", $newBal, (@VID eq $s_curr/@VID)),
  <res Type="0" VID="{$s_curr/@VID}" Time="{$s_curr/@Time}" Emit=""
  Speed="{ mxq:empty−to−zero($segToll/@avgSpeed) }" Toll="
  { mxq:empty−to−zero($segToll/@toll) }"></res>
  )
```

Listing A.7: *Q7: Toll Events*

Q8: Balance Query

```
declare  variable  $InputSeq external;
declare  variable  $BALANCE_STORAGE external;

for  $w in $InputSeq
where $w/@Type eq 2
return
let  $carBal := lr:store("BALANCE_STORAGE", (@VID eq $w/@VID) )
return
     <res Type="2" Time="{$w/@Time}" Emit="" ResultTime="
     {data($carBal/@Time)}" QID="{$w/@Qid}" Bal="{$carBal/@Bal}">
     </res>
```

Listing A.8: *Q8: Balance Query*

Q9: Daily Expenditure Query

```
declare  variable  $InputSeq external;

for  $w in $InputSeq
where ($w/@Type eq 3) return $w
```

Listing A.9: *Q9: Daily Expenditure Query*

List of Tables

List of Figures

Listings

Bibliography

[1] Apache ServiceMix. At http://servicemix.apache.org/.

[2] Coral8 CCL Reference. At http://www.coral8.com/.

[3] D4.1.2: Signaling Framework Prototype Single Domain. At http://www.master-fp7.eu/.

[4] D4.1.3: Monitoring Framework Prototype Single Domain. At http://www.master-fp7.eu/.

[5] D4.2.2: Monitoring Policy Lifecycle Extension Definition and Implementation. At http://www.master-fp7.eu/.

[6] D4.2.3: MASTER Monitoring Framework Prototype Multi Domain. At http://www.master-fp7.eu/.

[7] Esper Reference Documentation. At http://esper.codehaus.org/esper/documentation/.

[8] IBM InfoSphere Streams. At http://www-01.ibm.com/software/data/infosphere/streams/.

[9] MASTER Project. At http://www.master-fp7.eu/.

[10] MXQuery XQuery Engine. At http://www.mxquery.org/.

[11] StreamSQL Guide of the Implementation in Streambase. At http://streambase.com/.

[12] StreamSQL Homepage. At http://streamsql.org.

[13] Truviso, Inc. At http://www.truviso.com/.

[14] Xerces2 Java Parser Project Homepage. At http://xerces.apache.org/xerces2-j/.

[15] Daniel J. Abadi, Yanif Ahmad, Magdalena Balazinska, Ugur Cetintemel, Mitch Cherniack, Jeong-Hyon Hwang, Wolfgang Lindner, Anurag S. Maskey, Alexander Rasin, Esther Ryvkina, Nesime Tatbul, Ying Xing, and Stan Zdonik. The Design of the Borealis Stream Processing Engine. In *CIDR Conference on Innovative Data Systems Research*, 2005.

[16] Daniel J. Abadi, Donald Carney, Ugur Çetintemel, Mitch Cherniack, Christian Convey, C. Erwin, Eduardo F. Galvez, M. Hatoun, Anurag Maskey, Alex Rasin, A. Singer, Michael Stonebraker, Nesime Tatbul, Ying Xing, R. Yan, and Stanley B. Zdonik. Aurora: A Data Stream Management System. page 666, 2003.

[17] Jagrati Agrawal, Yanlei Diao, Daniel Gyllstrom, and Neil Immerman. Efficient Pattern Matching over Event Streams. In *SIGMOD*, 2008.

[18] Mohamed H. Ali, Ciprian Gerea, Balan Sethu Raman, Beysim Sezgin, Tiho Tarnavski, Tomer Verona, Ping Wang, Peter Zabback, Anton Kirilov, Asvin Ananthanarayan, Ming Lu, Alex Raizman, Ramkumar Krishnan, Roman Schindlauer, Torsten Grabs, Sharon Bjeletich, Badrish Chandramouli, Jonathan Goldstein, Sudin Bhat, Ying Li, Vincenzo Di Nicola, Xianfang Wang, David Maier, Ivo Santos, Olivier Nano, and Stephan Grell. Microsoft CEP Server and Online Behavioral Targeting. volume 2, pages 1558–1561, 2009.

[19] Gustavo Alonso, Fabio Casati, Harumi Kuno, and Vijay Machiraju. *Web Services - Concepts, Architectures and Applications*. Springer, November 2003.

[20] Rajeev Alur and David L. Dill. A Theory of Timed Automata. *Theoretical Computer Science*, 126(2), 1994.

[21] Manish Kumar Anand, Shawn Bowers, Timothy M. McPhillips, and Bertram Ludäscher. Efficient Provenance Storage over Nested Data Collections. In *EDBT*, pages 958–969, 2009.

[22] Anonymous. Pattern Matching in Sequences of Rows. *SQL Standard Change Proposal*, 2007.

[23] A. Arasu, M. Cherniack, E. F. Galvez, D. Maier, A. Maskey, E. Ryvkina, M. Stonebraker, and R. Tibbetts. Linear Road: A Stream Data Management Benchmark. In *VLDB Conference*, Toronto, Canada, September 2004.

[24] Brian Babcock, Shivnath Babu, Mayur Datar, Rajeev Motwani, and Jennifer Widom. Models and Issues in Data Stream Systems. In *PODS*, pages 1–16, 2002.

[25] Shivnath Babu, Utkarsh Srivastava, and Jennifer Widom. Exploiting k-Constraints to Reduce Memory Overhead in Continuous Queries Over Data Streams. *TODS*, 29(3), 2004.

[26] Deepavali Bhagwat, Laura Chiticariu, Wang Chiew Tan, and Gaurav Vijayvargiya. An Annotation Management System for Relational Databases. In *VLDB*, pages 900–911, 2004.

[27] Irina Botan, Gustavo Alonso, Peter M. Fischer, Donald Kossmann, and Nesime Tatbul. Flexible and Scalable Storage Management for Data-intensive Stream Processing. In *EDBT*, pages 934–945, 2009.

[28] Irina Botan, Roozbeh Derakhshan, Nihal Dindar, Laura M. Haas, Renée J. Miller, and Nesime Tatbul. SECRET: A Model for Analysis of the Execution Semantics of Stream Processing Systems. volume 3, pages 232–243, 2010.

[29] Irina Botan, Donald Kossmann, Peter M. Fischer, Tim Kraska, Dana Florescu, and Rokas Tamosevicius. Extending XQuery With Window Functions. In *VLDB '07*, pages 75–86, 2007.

[30] Badrish Chandramouli, Christopher N. Bond, Shivnath Babu, and Jun Yang. Query Suspend and Resume. In *ACM SIGMOD Conference*, Beijing, China, June 2007.

[31] S. Chandrasekaran, A. Deshpande, M. Franklin, J. Hellerstein, W. Hong, S. Krishnamurthy, S. Madden, V. Raman, F. Reiss, and M. Shah. TelegraphCQ: Continuous Dataflow Processing for an Uncertain World. In *CIDR Conference*, Asilomar, CA, January 2003.

[32] Surajit Chaudhuri, Raghav Kaushik, Abhijit Pol, and Ravi Ramamurthy. Stop-and-Restart Style Execution for Long Running Decision Support Queries. In *VLDB '07*, pages 735–745, 2007.

[33] James Cheney, Laura Chiticariu, and Wang Chiew Tan. Provenance in Databases: Why, How, and Where. *Foundations and Trends in Databases*, 1(4):379–474, 2009.

[34] Qi Cheng, Jarek Gryz, Fred Koo, T. Y. Cliff Leung, Linqi Liu, Xiaoyan Qian, and K. Bernhard Schiefer. Implementation of Two Semantic Query Optimization Techniques in DB2 Universal Database. In *VLDB*, pages 687–698, 1999.

[35] Mitch Cherniack, Hari Balakrishnan, Magdalena Balazinska, Donald Carney, Ugur Çetintemel, Ying Xing, and Stanley B. Zdonik. Scalable Distributed Stream Processing, 2003.

[36] Chuck Cranor, Theodore Johnson, Oliver Spataschek, and Vladislav Shkapenyuk. Gigascope: a Stream Database for Network Applications. In *SIGMOD*, 2003.

[37] Yingwei Cui, Jennifer Widom, and Janet L. Wiener. Tracing the Lineage of View Data in a Warehousing Environment. *ACM Transactions on Database Systems (TODS)*, 25(2):179–227, 2000.

[38] Abhinandan Das, Johannes Gehrke, and Mirek Riedewald. Approximate Join Processing over Data Streams. In *SIGMOD*, 2003.

[39] Susan B. Davidson, Sarah Cohen Boulakia, Anat Eyal, Bertram Ludäscher, Timothy M. McPhillips, Shawn Bowers, Manish Kumar Anand, and Juliana Freire. Provenance in Scientific Workflow Systems. *IEEE Data Eng. Bull.*, 30(4):44–50, 2007.

[40] Alan J. Demers, Johannes Gehrke, Mingsheng Hong, Mirek Riedewald, and Walker M. White. Towards Expressive Publish/Subscribe Systems. In *EDBT*, 2006.

[41] Amol Deshpande, Zachary Ives, and Vijayshankar Raman. Adaptive Query Processing. *Found. Trends databases*, 1:1–140, January 2007.

[42] S Devadas and A. R. Newton. Decomposition and Factorization of Sequential Finite State Machines. *IEEE Trans. Computer-Aided Design*, 8(11), 1989.

[43] Nihal Dindar, Baris Güç, Patrick Lau, Asli Ozal, Merve Soner, and Nesime Tatbul. DejaVu: Declarative Pattern Matching over Live and Archived Streams of Events. In *SIGMOD Conference*, pages 1023–1026, 2009.

[44] Luping Ding, Elke A. Rundensteiner, and George T. Heineman. MJoin: A Metadata-Aware Stream Join Operator. In *DEBS*, 2003.

[45] Luping Ding, Karen Works, and Elke A. Rundensteiner. Semantic Stream Query Optimization Exploiting Dynamic Metadata. In *ICDE*, pages 111–122, 2011.

[46] Kyumars Sheykh Esmaili. Data Stream Processing in Complex Applications, 2009.

[47] Kyumars Sheykh Esmaili, Tahmineh Sanamrad, Peter M. Fischer, and Nesime Tatbul. Changing Flights in Mid-air: A Model for Safely Modifying Continuous Queries. In *ACM SIGMOD International Conference on Management of Data (SIGMOD'11)*, Athens, Greece, June 2011.

[48] Peter M. Fischer, Kyumars Sheykh Esmaili, and Renée J. Miller. Stream Schema: Providing and Exploiting Static Metadata for Data Stream Processing. In *EDBT*, pages 207–218, 2010.

[49] Daniela Florescu, Chris Hillery, Donald Kossmann, Paul Lucas, Fabio Riccardi, Till Westmann, Michael J. Carey, and Arvind Sundararajan. The BEA Streaming XQuery Processor. *VLDB Journal*, 13(3):294–315, 2004.

[50] Mohamed Medhat Gaber, Arkady Zaslavsky, and Shonali Krishnaswamy. Mining Data Streams: A Review. *SIGMOD Record*, 34(2), 2005.

[51] Boris Glavic and Gustavo Alonso. Perm: Processing Provenance and Data on the Same Data Model through Query Rewriting. In *ICDE*, pages 174–185, 2009.

[52] Boris Glavic, Kyumars Sheykh Esmaili, Peter M. Fischer, and Nesime Tatbul. The Case for Fine-Grained Stream Provenance. In *BTW Workshop on Data Streams and Event Processing (DSEP'11)*, Kaiserslautern, Germany, February 2011.

[53] Lukasz Golab, Theodore Johnson, Nick Koudas, Divesh Srivastava, and David Toman. Optimizing Away Joins on Data Streams. In *SSPS '08*, 2008.

[54] J. Grant, J. Gryz, J. Minker, and L. Raschid. Semantic Query Optimization for Object Databases. *ICDE*, 1997.

[55] Todd J. Green, Grigoris Karvounarakis, Zachary G. Ives, and Val Tannen. Provenance in ORCHESTRA. *IEEE Data Eng. Bull.*, 33(3):9–16, 2010.

[56] Michael Henderson, Bryce Cutt, and Ramon Lawrence. Exploiting Join Cardinality for Faster Hash Joins. In *SAC*, 2009.

[57] Sun-Yuan Hsieh. The Interval-Merging Problem. *Inf. Sci.*, 177(2):519–524, 2007.

[58] Mohammad Rezwanul Huq, Andreas Wombacher, and Peter M. G. Apers. Facilitating Fine Grained Data Provenance Using Temporal Data Model. In *DMSN*, pages 8–13, 2010.

[59] Jeong-Hyon Hwang, Magdalena Balazinska, Alexander Rasin, Ugur Cetintemel, Michael Stonebraker, and Stan Zdonik. High-Availability Algorithms for Distributed Stream Processing. In *ICDE '05*, pages 779–790, 2005.

[60] Namit Jain, Shailendra Mishra, Anand Srinivasan, Johannes Gehrke, Jennifer Widom, Hari Balakrishnan, Ugur Çetintemel, Mitch Cherniack, Richard Tibbetts, and Stanley B. Zdonik. Towards a Streaming SQL Standard. *PVLDB*, 1(2):1379–1390, 2008.

[61] Theodore Johnson, S. Muthukrishnan, Vladislav Shkapenyuk, and Oliver Spatscheck. Query-Aware Partitioning for Monitoring Massive Network Data Streams. In *ICDE*, pages 1528–1530, 2008.

[62] Grigoris Karvounarakis, Zachary G. Ives, and Val Tannen. Querying Data Provenance. In *SIGMOD Conference*, pages 951–962, 2010.

[63] Silvio Kohler. Complex Event Detection on an Enterprise Service Bus. Master's thesis, ETH Zurich, Switzerland, 2009.

[64] F. Korn, S. Muthukrishnan, and Y. Zhu. Checks and Balances: Monitoring Data Quality Problems in Network Traffic Databases. In *VLDB*, pages 536–547, 2003.

[65] Wilburt Juan Labio, Janet L. Wiener, Hector Garcia-Molina, and Vlad Gorelik. Efficient Resumption of Interrupted Warehouse Loads. In *ACM SIGMOD Conference*, Dallas, TX, USA, 2000.

[66] Y. Law, Haixun Wang, and Carlo Zaniolo. Query Languages and Data Models for Database Sequences and Data Streams. In *VLDB*, 2004.

[67] 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. volume 1, pages 274–288, 2008.

[68] Hyo-Sang Lim, Yang-Sae Moon, and Elisa Bertino. Provenance-based Trustworthiness Assessment in Sensor Networks. In *DMSN*, pages 2–7, 2010.

[69] Morten Lindeberg, Vera Goebel, and Thomas Plagemann. Adaptive-sized Windows to Improve Real-time Health Monitoring: A Case Study on Heart Attack Prediction. In *ACM MIR Conference*, Philadelphia, PA, USA, March 2010.

[70] Mo Liu, Ming Li, Denis Golovnya, Elke A. Rundensteiner, and Kajal T. Claypool. Sequence Pattern Query Processing over Out-of-Order Event Streams. In *ICDE*, pages 784–795, 2009.

[71] Volkmar Lotz, Emmanuel Pigout, Peter M. Fischer, Donald Kossmann, Fabio Massacci, and Alexander Pretschner. Towards Systematic Achievement of Compliance in Service-Oriented Architectures: The MASTER Approach. *Wirtschaftsinformatik*, 50(5):383–391, 2008.

[72] Z. Manna and Amir Pnueli. *The Temporal Logic of Reactive and Concurrent Systems: Specification*. Springer-Verlag, New York, 1991.

[73] Archan Misra, Marion Blount, Anastasios Kementsietsidis, Daby M. Sow, and Min Wang. Advances and Challenges for Scalable Provenance in Stream Processing Systems. In *IPAW*, pages 253–265, 2008.

[74] R. Motwani, J. Widom, A. Arasu, B. Babcock, S. Babu, M. Datar, G. Manku, C. Olston, J. Rosenstein, and R. Varma. Query Processing, Approximation, and Resource Management in a Data Stream Management System. In *CIDR Conference*, Asilomar, CA, January 2003.

[75] Wim De Pauw, Mihai Letia, Bugra Gedik, Henrique Andrade, Andy Frenkiel, Michael Pfeifer, and Daby Sow. Visual Debugging for Stream Processing Applications. In *RV*, pages 18–35, 2010.

[76] Esther Ryvkina, Anurag Maskey, Mitch Cherniack, and Stanley B. Zdonik. Revision Processing in a Stream Processing Engine: A High-Level Design. In *ICDE*, page 141, 2006.

[77] Utkarsh Srivastava and Jennifer Widom. Flexible Time Management in Data Stream Systems. In *PODS*, pages 263–274, 2004.

[78] Beat Steiger. Data Lineage/provenance in XQuery. Master's thesis, ETH Zurich, Switzerland, 2010.

[79] Hong Su, Elke A. Rundensteiner, and Murali Mani. Semantic Query Optimization for XQuery over XML streams. In *VLDB*, 2005.

[80] N. Tatbul, U. Çetintemel, S. Zdonik, M. Cherniack, and M. Stonebraker. Load Shedding in a Data Stream Manager. In *VLDB Conference*, Berlin, Germany, September 2003.

[81] N. Tatbul and S. Zdonik. Window-aware Load Shedding for Aggregation Queries over Data Streams. In *VLDB Conference*, Seoul, Korea, September 2006.

[82] Wolfgang Thomas. Automata on Infinite Objects. In *Handbook of Theoretical Computer Science (vol. B): Formal Models and Semantics*, pages 133–191. MIT Press, 1990.

[83] Kostis Tsoulos. XML schema support in MXQuery. Master's thesis, ETH Zurich, Switzerland, 2008.

[84] Peter A. Tucker, David Maier, Tim Sheard, and Leonidas Fegaras. Exploiting Punctuation Semantics in Continuous Data Streams. *TKDE*, 15(3), 2003.

[85] Stratis D. Viglas and Jeffrey F. Naughton. Rate-Based Query Optimization for Streaming Information Sources. In *SIGMOD*, 2002.

[86] N. N. Vijayakumar and Beth Plale. Tracking Stream Provenance in Complex Event Processing Systems for Workflow-Driven Computing. In *EDA-PS Workshop*, 2007.

[87] Nithya N. Vijayakumar and Beth Plale. Towards Low Overhead Provenance Tracking in Near Real-Time Stream Filtering. In *IPAW*, pages 46–54, 2006.

[88] Min Wang, Marion Blount, John Davis, Archan Misra, and Daby M. Sow. A Time-and-Value Centric Provenance Model and Architecture for Medical Event Streams. In *HealthNet*, pages 95–100, 2007.

[89] Jennifer Widom. Trio: A System for Managing Data, Uncertainty, and Lineage. *Managing and Mining Uncertain Data*, 2008.

[90] Yin Yang, Jürgen Krämer, Dimitris Papadias, and Bernhard Seeger. Hybmig: A Hybrid Approach to Dynamic Plan Migration for Continuous Queries. *IEEE Trans. Knowl. Data Eng.*, 19(3):398–411, 2007.

[91] Yali Zhu, Elke A. Rundensteiner, and George T. Heineman. Dynamic Plan Migration for Continuous Queries over Data Streams. In *ACM SIGMOD Conference*, Paris, France, June 2004.