Storage Management Techniques for Stream Processing

A dissertation submitted to
ETH ZURICH

for the degree of
Doctor of Sciences

presented by
IRINA MAGDALENA BOTAN
M. Sc. "Babes-Bolyai" University Cluj-Napoca, Romania
born 26 September 1981
citizen of Romania

accepted on the recommendation of
Prof. Dr. Donald Kossmann, examiner
Prof. Dr. Mitch Cherniack, co-examiner
  Prof. Dr. Tamer Özsu, co-examiner
Prof. Dr. Nesime Tatbul, co-examiner

2011
Acknowledgments

First and foremost, I would like to thank my advisor, Prof. Donald Kossmann, for the chance he offered me to pursue a PhD degree in his group at ETH, for his guidance in my research work and for the example he set through his own professional and personal life. I learned a lot from Donald during these years.

I would also like to thank my co-advisor, Prof. Nesime Tatbul. Nesime taught me much about how to write papers and offered me the chance to work on very interesting topics.

I am so very grateful to Prof. Irina Athanasiu who guided me during my undergraduate years and encouraged me to pursue a PhD degree.

I am also grateful for the work and availability of my co-advisors Prof. Tamer Özs and Prof. Mitch Cherniack who agreed to review my dissertation.

My thanks go to Prof. Gustavo Alonso, for his help in improving the Storage Manager for Streams work; to Tim Kraska, Peter Fischer and Rokas Tamosevicius for the exciting and fruitful collaboration on the XQuery windowing extensions implementation. Also to Laura Haas, Prof. Renée Miller, Nihal Dindar and Roozbeh Derakhshan for the great experience of jointly working on the SECRET model; to the SAP team we collaborated with in the MaxStream project.

Many thanks to my colleagues and friends: Cristian, Peter, Alex, Ionut, Michael, Martin, Marcos, Eric, Kyumars, Stefan, Lukas. A warm thank you goes to Simonetta, Eva, Jena and Heidy for their help with administrative issues and their friendship.

I would also like to thank my father, Niculae, my family and friends for giving me moral support and for not asking too often about my graduation date.

A special thank you goes to my dear husband, Virgiliu, for always being there for me and for giving me the strength to continue during hard times, and to my sweet daughter, Sara, for allowing me to sleep at night and for making my days brighter with a smile.

Finally, I would like to thank, in loving memory, to my dear mother, Magdalena, for her unconditional love, guidance, support, and for always believing in me.
Contents

1 Introduction 1
  1.1 Background .................................................. 1
  1.2 Problem Statement ........................................... 3
  1.3 Storage Manager for Streams ............................... 4
  1.4 Structure of the Dissertation ............................... 7

2 Modeling Streaming Systems Behavior 9
  2.1 Introduction .................................................. 9
  2.2 Systems Heterogeneity ....................................... 11
  2.3 Basic Definitions and Assumptions ......................... 14
  2.4 The SECRET Model ........................................... 17
  2.5 Modeling Systems Behavior ................................. 24
    2.5.1 State-of-the-art DSMSs in SECRET ...................... 24
    2.5.2 Using SECRET to Analyze a new DSMS ................. 26
  2.6 Summary .................................................. 26

3 Continuous Processing in XQuery 27
  3.1 Introduction ................................................ 27
  3.2 Usage Scenarios ............................................ 29
    3.2.1 Motivating Example .................................... 29
    3.2.2 Other Applications .................................... 30
  3.3 FORSEQ Clause ........................................... 32
<table>
<thead>
<tr>
<th>Section</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>5.4.1</td>
<td>Concepts</td>
<td>116</td>
</tr>
<tr>
<td>5.4.2</td>
<td>Real World Systems</td>
<td>118</td>
</tr>
<tr>
<td>5.5</td>
<td>Implementation Considerations</td>
<td>124</td>
</tr>
<tr>
<td>5.5.1</td>
<td>Transaction Manager Implementation</td>
<td>124</td>
</tr>
<tr>
<td>5.5.2</td>
<td>Transaction Manager Interface</td>
<td>126</td>
</tr>
<tr>
<td>5.5.3</td>
<td>Recoverability and Recovery</td>
<td>127</td>
</tr>
<tr>
<td>5.5.4</td>
<td>Performance Optimization</td>
<td>128</td>
</tr>
<tr>
<td>5.5.5</td>
<td>Transaction Definition</td>
<td>129</td>
</tr>
<tr>
<td>5.6</td>
<td>Experimental Setup</td>
<td>130</td>
</tr>
<tr>
<td>5.6.1</td>
<td>The Linear Road Benchmark</td>
<td>130</td>
</tr>
<tr>
<td>5.6.2</td>
<td>The ADHOC Implementation</td>
<td>131</td>
</tr>
<tr>
<td>5.6.3</td>
<td>The TM implementation</td>
<td>132</td>
</tr>
<tr>
<td>5.6.4</td>
<td>Experimental Tools</td>
<td>133</td>
</tr>
<tr>
<td>5.7</td>
<td>Experimental Results</td>
<td>134</td>
</tr>
<tr>
<td>5.7.1</td>
<td>Performance</td>
<td>134</td>
</tr>
<tr>
<td>5.7.2</td>
<td>Correctness of Results</td>
<td>136</td>
</tr>
<tr>
<td>5.7.3</td>
<td>Failure Handling</td>
<td>137</td>
</tr>
<tr>
<td>5.7.4</td>
<td>Sensitivity to Transaction Size</td>
<td>138</td>
</tr>
<tr>
<td>5.8</td>
<td>Summary</td>
<td>140</td>
</tr>
<tr>
<td>6</td>
<td>Conclusions</td>
<td>143</td>
</tr>
<tr>
<td>6.1</td>
<td>Summary of the Dissertation</td>
<td>143</td>
</tr>
<tr>
<td>6.2</td>
<td>Future Work</td>
<td>145</td>
</tr>
</tbody>
</table>
Abstract

A variety of applications require low-latency processing of data that comes in highly-dynamic streams of items. These applications are implemented using Data Stream Management Systems (DSMSs). More recently, new application domains like real-time business intelligence turned to the “on-the-fly” processing model employed by these systems for a solution to their challenges. As a result, the requirements imposed on the DSMSs have become more complex: e.g., mechanisms for correlating data streams with stored information or near-real time complex analysis of large portions of streaming data.

In order to meet the evolving requirements of modern streaming applications, a clean, flexible and high performance DSMS design is required. Although many system implementations were proposed, none of them offers a clean, systematic approach to data storage management. Rather, the storage manager is usually tightly coupled with the continuous query execution engine. This design decision limits the possibility for further performance improvement and severely restricts the flexibility necessary to accommodate new application requirements. Moreover, today, there is no standard for querying streams and, as a result, each DSMS exposes its own execution semantics, making the implementation of the new requirements even more challenging.

This dissertation investigates the design and implementation of a general-purpose storage management framework for Data Stream Management Systems, that we name SMS (Storage Manager for Streams). The ultimate goal of this framework is to provide a general, clean, flexible and high-performance storage management system which could be virtually “plugged” into any DSMS. In order to achieve this goal, in this work, we combine the experience gained over decades of research on Database Management Systems with the high-performance mechanisms employed by the Data Stream Management Systems.

Following the database systems architecture design, this framework is based on the principle of separating concerns: the query processor is decoupled from the storage
manager. As such, the storage system obtains the flexibility necessary to accommodate new requirements, behind a general interface. Moreover, it can provide specialized store implementations tailored to the particular requirements of the applications, which is key to achieving good performance. In this respect, an important contribution of the framework is the reuse of the access patterns of the continuous query operators to tune the stores’ implementation and as such, to speed up the access on materialized data.

In addition, the unified transactional model proposed in this dissertation makes minimal extensions to the traditional transactional model in order to accommodate streams and continuous queries. As a result, it offers a clean semantics for continuous query execution over arbitrary combinations of data sources (streaming and stored) in the presence of concurrent access and failures. And even more, it can be used to explain the transactional behavior of state-of-the-art DSMSs.

A series of experiments are conducted using the Linear Road streaming benchmark’s implementation in MXQuery (a Java-based open-source XQuery engine, extended with window functions for continuous processing). MXQuery uses SMS for all its data storage related tasks. Our experiments show that the response time of the continuous queries can indeed be lowered if the store implementations are tuned according to the access patterns of the continuous query operators. Moreover, a transaction manager implementing the unified transactional model and designed as an additional component between the access and storage layers of SMS provides correctness and reliability for the Linear Road application with practically no performance penalty. As such, the experimental results indicate that a storage manager built on these ideas is a promising approach.
Zusammenfassung


Um die sich kontinuierlich entwickelnden Anforderungen an moderne Anwendungen von Datenströmen zu erfüllen, ist ein sauberes, flexibles und performantes Design des DSMS erforderlich. Es wurden viele Systemimplementierungen vorgeschlagen, doch keine davon bietet einen sauberen, systematischen Ansatz bezüglich Speicherverwaltung. Stattdessen ist die Speicherverwaltung typischerweise eng an die Query Execution Engine gekoppelt. Diese Designentscheidung begrenzt die Möglichkeiten für weitergehende Performanceverbesserungen und schränkt die zur Integration der Anforderung neuer Anwendungen benötigte Flexibilität empfindlich ein. Darüber hinaus besteht heute kein Standard um Datenströme abzufragen und als Ergebnis davon bietet jedes DSMS seine eigene Ausführungssemantik an, was die Implementierung neuer Anforderungen noch herausfordernder gestaltet.

Diese Dissertation untersucht das Design und die Implementierung eines universellen Speichermanagement-Frameworks für Datenstromverwaltungssysteme, welches wir SMS (Storage Manager for Streams - Speicherverwaltung für Datenströme) nennen. Das endgültige Ziel dieses Frameworks ist es, ein generelles, sauberes, flexibles und performantes System zur Speicherverwaltung zu bieten, das praktisch in jedes beliebige DSMS "eingesteckt" werden kann. Um dieses Ziel zu erreichen kombinieren wir in dieser Arbeit die Erfahrung aus Jahrzehnten von Forschung in Datenverwaltungssystemen mit den Hochleistungsmechanismen, die in Datenstromverwaltungssystemen ver-
wendet werden.


Zudem nimmt das vereinheitlichte Transaktionsmodell, das in dieser Disseration vorgeschlagen wird, nur minimale Erweiterungen am traditionellen, transaktionalen Modell vor, um Datenströme und Continuous Queries unterzubringen. Daraus resultiert eine saubere Semantik für die Ausführung von Continuous Queries über beliebige Kombinationen von Datenquellen (Datenströme und Datenspeicher) in Gegenwart von gleichzeitigen Zugriffen und Fehlern. Darüber hinaus kann es benutzt werden, um das transaktionale Verhalten von modernen DSMS zu erklären.

Eine Reihe von Experimenten wurden mit der Implementierung des Linear Road Streaming Benchmarks auf MXQuery durchgeführt (MXQuery ist eine Java-basierte, quelloffene XQuery Engine, die mit Windowfunktionalität für Continuous Queries erweitert wurde). MXQuery verwendet SMS für alle Speicheraufgaben. Unsere Experimente zeigen, dass die Antwortzeit der Continuous Queries in der Tat gesenkt werden kann, wenn die Speicherimplementierung entsprechend der Zugriffsmuster der Operatoren von Continuous Queries auf Leistung gebracht werden. Darüber hinaus weisen die experimentellen Ergebnisse darauf hin, dass eine Speicherverwaltung, die auf diesen Ideen gebaut ist, ein vielversprechender Ansatz ist.
Chapter 1

Introduction

The Data Stream Management Systems (DSMSs) represent a class of systems built to meet the requirements of data streams applications. These applications come from diverse domains like financial services, sensor monitoring, network traffic, RSS blogs, security, manufacturing, highway traffic monitoring etc. and require timely processing of data that arrives in streams of items.

In this chapter, we first briefly present the data stream processing background. Next, we outline the challenges addressed by this dissertation followed by the contributions it makes. The chapter ends with a road map of the dissertation.

1.1 Background

Data sources like sensors which measure temperature, software reporting car positions on highways or monitoring network data transfer, generate possibly infinite sequences of data items, i.e., streams. These data have to be processed for timely generation of alarms (e.g., if the temperature raises above a threshold), to compute statistics (e.g., how many cars are stopped in a certain segment of a highway), or to detect patterns (e.g., high message rates from a single source in a network).

The traditional Database Management Systems (DBMSs) are simply not suitable for the performance requirements of the streaming applications [24]. More specifically, these systems use a processing model in which mostly static data are first stored, possibly indexed and then analyzed through one-time queries [21].

The highly dynamic, unbounded nature of data streams requires a new processing paradigm: data items from a variety of sources are pushed into the system to be then
Chapter 1. Introduction

Figure 1.1: Typical Data Stream Management System Architecture

processed by persistent, continuous queries which continually produce results to their outputs as new data items arrive in their input sources.

In this context, a new class of systems called Data Stream Management Systems (DSMSs) was proposed. The typical high-level architecture of a DSMS is sketched in Figure 1.1. As depicted in the figure, a DSMS is composed of two main building blocks: (i) a query processing component which is responsible for managing the operators forming the continuous queries as well as the data flow among them, which we name SPE (Stream Processing Engine) and (ii) an internal, usually in-memory, working storage for the various materialization needs during query execution.

There has been a lot of research on numerous aspects of data stream processing, from data models, to continuous query operators implementation, languages and optimization techniques for improving performance etc. Comprehensive reviews of the models and the issues arising from this new processing paradigm are presented in [24] and [58].

Some of the challenges posed by data streams processing had been also previously addressed by works on active databases [89] (trigger-style execution of continuous queries over append-only active tables), time-series databases [45] (the time-based ordering of tuples is used for query execution and optimizations), or sequence databases [90] (a language and query algebra for ordered relations are proposed). The publish/subscribe systems [44] also expose an architecture in which the subscribers (clients to the application) register for information which is pushed into the system by publisher data sources.

Many general-purpose Data Stream Management Systems have been proposed to date. Some of these systems left the research domain (STREAM [20], Aurora [18], Borealis [17], TelegraphCQ [36], CEDR [28]) to become commercial products (Stream-
1.2 Problem Statement

More recently, new application domains like real-time (operational) business intelligence [29] turned to the streaming technology for more efficient mechanisms to process large amounts of data. As a result the streaming applications are more complex: they involve multiple continuous queries that run in parallel, join live data with stored historical information, and are highly data-intensive, requiring temporary materialization of considerable portions of the data streams as well as maintenance of large and highly dynamic operator states [31]. These applications often operate under very strict performance requirements.

In order to meet the requirements of modern data stream applications, clean, flexible and high-performance stream processing system solutions are required. This is true for both the continuous query processing engine (the SPE) as well as the data storage management component of a Data Stream Management System.

Although many DSMS solutions have been offered to date, none of them provides a clean and systematic approach to storage management. In most of the existing systems (e.g., Aurora[18], STREAM[20], TelegraphCQ[36]), the storage manager is tightly coupled with the query processing engine. Such a design severely limits exploiting the optimization opportunities of the applications to their full potential, since a hard-coded implementation does not leave much room for optimization. In addition, this architecture makes the storage managers inflexible, and consequently does not allow the easy integration of new requirements.

Another problem is that currently no clean semantics exists for continuous query execution over combinations of streams and stored data sources under concurrent access and failures. This is critical need considering that modern streaming applications require features like correlations of live data with stored information or enhancing the streaming data items with meta-data. Actually the problem goes even deeper, as in fact,

\[\text{The list of references is not exhaustive and related work more specific to each contribution will be presented in the following chapters}\]
there is no agreed semantics on querying data streams. As a result, each DSMS ended up proposing its own “transactional model”. In most cases, the transactional properties are not even explicitly defined, but they are rather embedded in the execution models or the operator semantics of those systems, making them hard to understand and inflexible to use.

In this context, it becomes hard to implement correct, reliable, and high performance modern streaming applications.

1.3 Storage Manager for Streams

In this dissertation, we investigate the design and implementation of a general-purpose storage management framework for Data Stream Management Systems. The basic insight of this framework is that it combines the techniques and algorithms developed over decades of research on DBMSs with the high-performance mechanisms employed by DSMSs. The result is SMS (Storage Manager for Streams), a general, clean and flexible storage management system which can help increase the performance of streaming applications and may virtually be “plugged” into any DSMS.

This framework is built on the traditional architecture design principle of database systems: the storage manager is separated from the query processor. As a result, the storage manager gains the flexibility to accommodate new requirements and to provide specialized store implementation, behind a general interface.

In practice, being able to customize and tailor the storage management to the requirements of the application is key to achieving good performance. In this respect, one important property of this framework is that, similarly to relational DBMSs, it uses the read patterns of the operators (i.e., continuous query operators) to tune the stores’ implementations. Access pattern analysis and prediction is even more natural in a DSMS where the queries are persistent and are usually known a-priori.

In addition, our storage management framework understands the specific characteristics of data streams processing: given the update-intensive environment, in addition to the read patterns, the update patterns of the operators can also be used to tune the implementation.

In order to cleanly define the semantics of continuous query execution over arbitrary combinations of streaming and stored data sources in the presence of concurrent access and failures, we propose the design and implementation of a unified transactional model. The challenge we face is that streaming systems operate on events, while tradi-
tional stored data sources work with operations (read, update). Moreover, traditionally, queries are one-time, while stream processing uses long-running, continuous queries.

The unified transactional model reuses the traditional transactional theory with minimal extensions to support data streams and continuous queries. Its basic insight is that streaming and stored data sources can be treated uniformly: they are all just data sources to the continuous queries on which updates and queries are executed. Transactions in the execution space over streaming and stored data sources are similar to the traditional ones, but they also accommodate events (stream data items) and continuous queries. To define the correct interleaving of the operations composing the transactions, we propose to use conflict-serializability \[95\]. However, more restricted (e.g., supplementary ordering of transactions) or relaxed (e.g., weaker isolation levels) correctness criteria can be defined depending on the application requirements.

Our proposal of decoupling query processing from storage management makes the integration of a transaction manager natural. Similarly to the DBMS design, we implement the transaction manager as an additional component between SMS’s access and storage layers.

This research work is conducted in the context of MXQuery [6], a light-weight Java-based open-source XQuery [28] engine extended with window functions. Using the SMS as the underlying storage manager for MXQuery, we implemented the Linear Road Benchmark [23]. This streaming benchmark is demanding and tests various aspects of stream processing. We use this benchmark implementation to validate our ideas and demonstrate that our proposal is a promising approach to streaming storage management.

More specifically, the contributions this dissertation makes are grouped around two main topics and are presented in the following:

- **Storage Management for Streams**

  **DSMSs Behavior Analysis.** There is a high heterogeneity among the execution models of the DSMSs today. We present the analysis of a set of state-of-the-art DSMSs (STREAM [22], StreamBase [11], Coral8 [1], StreamInsight [4]) and introduce their behavior formally as instantiations of a general, descriptive model we developed (the SECRET model [32]).

  **XQuery Extensions Implementation.** Two extensions which allow XQuery to process infinite inputs and execute continuous queries are implemented in MXQuery. As a result of the extensions, MXQuery has the capabilities of a Data Stream Management System.
Linear Road Benchmark Implementation. We present a detailed description of the Linear Road Benchmark’s implementation in MXQuery.

Storage Requirements Analysis. A fine-grained analysis is conducted to identify the key storage requirements, which are then translated into storage tuning parameters with corresponding values.

Storage Management Interface. Similar to the traditional DBMSs architecture design, the decoupling of query processing from storage management as a DSMS design principle is proposed. A general-purpose interface is introduced which is used to specify storage requirements as parameter values as well as to allow basic access operations on the stored data.

Implementation Tuning for Performance. A series of algorithms to provide customized store implementations based on read and update patterns information are presented. As a result, the performance of storage access can be improved.

Storage Manager Architecture, Implementation and Performance. An important contribution is the architecture of the Storage Manager for Streams (SMS) which implements the ideas described above. Its performance is analyzed using the Linear Road Benchmark implementation in MXQuery as well as microbenchmarks.

- Transactional Stream Processing.

  Unified Execution Space. We define the space of possible executions of continuous and one-time queries over arbitrary combinations of streaming and stored data sources.

  Unified Transactional Model. We show how streaming events and operations on stored data can be flexibly grouped in transactions. Conflict-serializability and Conflict-serializability with Arrival Ordering are proposed as correctness criteria to specify the correct interleavings of the operations belonging to these transactions. As a result, we design a unified transactional model over stored and streaming data sources.

  DSMS Transactional Behavior Description. Using the unified model, the transactional behavior of some state-of-the-art DSMSs can be described.

  Unified Transaction Manager Implementation and Performance. The implementation of a transaction manager which uses the unified model is presented. Moreover, the experiments with the Linear Road Benchmark show that the performance penalty is low compared to the benefits (correctness and reliability) brought by the presence of a transaction manager.
1.4 Structure of the Dissertation

The chapters composing this dissertation can be read also individually, as each convey a complete idea. As graphically depicted in Figure 1.2, this dissertation is structured like in the following:

- In Chapter 2, we present SECRET, a model for the behavior of Data Stream Management Systems. The goal is to have a clear understanding of today’s state-of-art DSMSs. Moreover, this chapter lists the definitions for the basic stream processing notions which will be extensively used throughout this dissertation. The content presented in this chapter is based on a joint work published at VLDB 2010 [32].

- Chapter 3 introduces the implementation in MXQuery of two extensions for XQuery which allow it to do continuous query processing. We also present the details of the Linear Road Benchmark [23] as well as its implementation in MXQuery. As such, this chapter describes the context in which our research work on storage management and transactional stream processing was conducted. The content is based on a joint work published in VLDB 2007 [33].

- In Chapter 4, we present the analysis of streaming applications with respect to storage requirements as well as the algorithms to provide high performance customized storage implementations. The architecture of the Storage Manager for Streams (SMS) is introduced as well as a detailed analysis of its performance. The content was published in an article in EDBT 2009 [30].

- Completing the frame, the unified transactional model and its implementation are

![Figure 1.2: Schematic Road Map of the Dissertation Outlining Contributions]
described in Chapter 5.

- Finally, Chapter 6 summarizes the content of this dissertation and presents the conclusions and lessons learned. In addition, it indicates some possible avenues for future work.
Chapter 2

Modeling Streaming Systems Behavior

There are many academic and commercial Data Stream Management Systems today. Yet, there are no standards for querying data streams: each system proposes its own syntax and even particular execution semantics.

In this chapter, we present the execution behavior of a set of state-of-the-art heterogeneous DSMSs using SECRET, a descriptive model that allows us to analyze and understand the results of window-based queries along four dimensions. Each of these dimensions represents a certain aspect of window-based execution from window construction to actual execution and result generation. As such, the model gives an end-to-end view of what impacts execution semantics from inputs to outputs. Moreover, this model provides the basic formalism for definitions and assumptions that will set the background for the main components of the dissertation to follow.

In the chapter, we first outline through examples the types of heterogeneity found in the current systems and give formal definitions of the key stream processing components (e.g., data stream, window). Next, we formally present the model and subsequently use it to describe the behavior of five DSMSs.

2.1 Introduction

Stream computing is passing from the domain of pure research into the real world of commercial systems. Research projects like Aurora[18], TelegraphCQ[36], STREAM[82] have shown how data can be processed as it pours into a system from a diversity of
Chapter 2. Modeling Streaming Systems Behavior

sources such as sensors, online transactions, and other feeds. Each system proposed its own set of operators, windowing constructs, and, in some cases, whole new query languages [22, 52]. As these systems have been commercialized [1, 11, 13], they have added features to meet the needs of their own customers. For the purchaser or user of a DSMS, the choices are confusing. Without a clear understanding of features and semantics, applications are not portable, and can be hard to build, even on a given DSMS.

The emerging processing engines have different capabilities, and even common capabilities may be expressed differently in different systems. For example, both StreamBase [11] and Coral8 [1] allow time-based windows where a window is defined by an interval size (in units of time) and where different windows are separated by a slide value that specifies how many units of time separate the start of different consecutive windows. To specify such a window in StreamBase, the user has to write “[SIZE x ADVANCE y TIME]”. In Coral8, the same function is requested with the “KEEP x SECONDS” clause. StreamBase allows an arbitrary slide value for a window (specified by the ADVANCE clause); Coral8 only permits two values: 1 msec or a slide that is equal to the window size. Worse yet, the underlying semantics of such common features as windows is often radically different. Even if the window size and the slide are set to the same respective values in Coral8 and StreamBase, different query results can be obtained due to hidden differences in their query execution models.

It is hard to get information about underlying formal models used by current commercial systems [1, 2, 4, 9, 11, 13]. Each system seems to use a different model, and the query results that they generate are not easy to compare. Jain et al. [65] tried to reconcile the differences across two of these commercial systems, Oracle CEP and StreamBase. They only consider the way that window execution is triggered. Though an important first step, this work focuses on only one aspect of execution behavior, just one of the aspects the model we present next captures and explains.

As a first generation research system, STREAM’s CQL provides a formal model based on the relational model [22]. CQL is an extension to SQL:1999 that introduces a new “stream” data type which can be used together with relations. In addition to the “relation-to-relation” operations of the relational algebra, CQL introduces “stream-to-relation” operations for constructing windows on streams as well as “relation-to-stream” operations to convert results of relational operations back into the stream data type. Through these mappings, one can essentially reuse most of the relational algebra semantics in a rather straight-forward way. Finally, CQL has also introduced the notion of time into the query execution semantics: time advances from \( t - 1 \) to \( t \), when all data items up to \( t - 1 \) have
2.2 Systems Heterogeneity

The heterogeneity among DSMSs exposes itself at three levels:

- **Syntax heterogeneity**: This type of heterogeneity refers to the differences in the language clauses (keywords) used for the definition of common constructs (e.g., windows). The syntax differences are understandable given the lack of a standard language for stream processing.

- **Capability heterogeneity**: This type of heterogeneity refers to the differences in support for certain types of queries across different DSMSs, and also exposes itself at the language syntax level. For example, Coral8 offers a clause that controls how often a query result should be emitted, a feature we have not encountered in any other system. For example, an `OUTPUT EVERY 2 ROWS` clause at the end of...
a query suppresses the output to only reflect the query results generated by each second row arrival in the input.

- **Execution model heterogeneity**: This type of heterogeneity refers to the differences in the underlying query execution models across different DSMSs. It is hidden from and cannot be influenced by the application developer, and is subtler than the other types of heterogeneity, hence potentially more confusing. As a result, we focus on analyzing the execution semantics of the systems.

To better understand the need for this type of descriptive model, consider three examples, defined on a simple input stream $\text{InStream}(\text{Time}, \text{Val})$ of tuples. Time represents the application timestamp (generated and assigned by the source application) of the tuple in seconds, and Val, an integer value, represents the content of the tuple. Our queries compute an average over Val, and $\text{OutStream(Avg)}$ is the output stream containing the results of the query.

**Example 1: Window construction in a DSMS**

Consider a query which computes the average value of the tuples in the input stream using a time-based tumbling window of size 3 seconds. We ran this query on StreamBase three different times, each time feeding it exactly the same input data file (we only shifted the application timestamps by one second for each run and asked the system to feed the input according to those timestamps). Surprisingly, StreamBase produced different results each time:

\[
\begin{align*}
\text{InStream}(\text{Time}, \text{Val}) &= \{(10, 10), (11, 20), (12, 30), (13, 40), (14, 50), (15, 60), (16, 70), \ldots\} \\
\text{StreamBase Output1} &= \{(15), (40), \ldots\} \\
\text{InStream}(\text{Time}, \text{Val}) &= \{(11, 10), (12, 20), (13, 30), (14, 40), (15, 50), (16, 60), (17, 70), \ldots\} \\
\text{StreamBase Output2} &= \{(10), (30), \ldots\} \\
\text{InStream}(\text{Time}, \text{Val}) &= \{(12, 10), (13, 20), (14, 30), (15, 40), (16, 50), (17, 60), (18, 70), \ldots\} \\
\text{StreamBase Output3} &= \{(20), (50), \ldots\}
\end{align*}
\]

\(^1\)A time-based window contains the tuples arriving in a specified time interval; a window is tumbling if it slides by an interval that is greater or equal to the size of the window. Please see Section 2.3 for a more formal definition of time-based windows.
Intuitively, as the query mentions a time-based tumbling window of size three seconds we expected the result of Output3 in all runs (i.e., the first three tuples belong to the first window, the next three to the second window, etc.). However, the slightly different time values seem to have led to a difference in window construction, hence different results. More specifically, it seems that in the first run’s case, a result is reported for a window containing only the first two data items ((10, 10) and (11, 20)) and for the second run, the first item alone ((10, 10)) generates a result.

Example 2: Evaluation differences across DSMSs

Consider a query which continuously computes the average value of the tuples over a time-based window of size 5 seconds that slides by 1 second. We ran this query in three different DSMSs: STREAM [10], Coral8 [1], and StreamBase [11], with the following results:

\[
\begin{align*}
\text{InStream}(\text{Time}, \text{Val}) & = \{(30, 10),(31, 20),(36, 30), \ldots\} \\
\text{STREAM Output} & = \{(10),(15),(20),(30), \ldots\} \\
\text{Coral8 Output} & = \{(10),(15),(20),(30), \ldots\} \\
\text{StreamBase Output} & = \{(10),(15),(15),(15),(15),(20),(30), \ldots\}
\end{align*}
\]

This is a simple query that is supported by all three systems. Thus, one would naturally expect that all systems would return the same result. However, as shown above, this was not the case: StreamBase produced a different result than STREAM and Coral8. To understand this difference, consider Figure 2.1. It shows a window moving over the stream, each bar representing the position of the window at a certain moment in time. At first, the window contains only tuple (30, 10); then it additionally contains tuple (31, 20); and so forth. At time 35, the window moves so that it no longer contains (30,
Chapter 2. Modeling Streaming Systems Behavior

10; likewise, at time 36, tuple (31, 20) “expires” from the window and tuple (36, 30) is inserted. So where does the difference in results come from? In STREAM and Coral8, the Average operator is invoked on a window whenever the window’s content changes (i.e., when a tuple is added to or expires from the window), whereas in StreamBase, the invocation happens for each window, even if the tuple content of the window stays the same. As a result, in contrast to Coral8 and STREAM, StreamBase will generate an output result for the third, fourth and fifth windows (or window positions) as well. Thus the evaluation strategy used by an system is another important factor affecting the query results.

Example 3: State Change

As has been the focus of a publication by Jain et al [65], DSMSs may react each to different events (e.g., arrival of single tuple in an input, time passing etc.), leading to different query execution semantics and results. This paper provides an extensive set of examples showing the problem between the tuple-driven execution model of StreamBase [11] and the time-driven execution model of Oracle [9]. As depicted in Section 2.5, we determined that Coral8 exposes a batch-based execution model, similar to the proposed unifying solution in [65].

These three motivating examples show that we need a way to understand, express, and predict the query execution behaviors of different DSMSs. For this purpose, we will use the SECRET model, presented next in the chapter.

2.3 Basic Definitions and Assumptions

With so many models around, we need to clearly position our work in the stream processing context. In this section we do so by presenting the definitions for a set of basic stream processing concepts and the constructs that are used in the SECRET model, together with any relevant assumptions we make.

Definition 1 (Time Domain) The time domain $\mathbb{T}$ is a discrete, linearly ordered, countably infinite set of time instants $t \in \mathbb{T}$. We assume that $\mathbb{T}$ is bounded in the past, but not necessarily in the future.

Definition 2 (Stream) A stream $S$ is a countably infinite set of elements $s \in S$. Each stream element $s : \langle v, t^{app}, t^{sys}, bid \rangle$ consists of a relational tuple $v$ conforming to a schema $S$, with an application time value $t^{app} \in \mathbb{T}$, a system time value $t^{sys} \in \mathbb{T}$, and a batch-id value $bid \in \mathbb{N}$. We use the notation $s.t^{app}$, $s.t^{sys}$, and $s.bid$ to denote the
2.3. Basic Definitions and Assumptions

application time value, system time value, and batch-id value of stream element $s$, respectively.

In the above definition (as in related work [91]), two different notions of time are used: “application time” ($t^{app}$) and “system time” ($t^{sys}$). These both take values from the time domain $\mathbb{T}$, but carry two different meanings, and therefore, are used for two different purposes in our model. The value $t^{app}$ captures the time information that is associated with the occurrence of the application event that a stream element represents (usually provided by the data source), and therefore is used as the basis for query execution over the stream; whereas $t^{sys}$ captures the time information that is associated with the occurrence of the related system event (arrival of the corresponding stream element at the system) and is therefore used as the basis for reasoning about tuple arrival events in the system and how the system should react to them. Elements in a stream are assigned unique $t^{sys}$ values, but multiple elements can share the same $t^{app}$ value. Therefore, streams are totally ordered by the $t^{sys}$ values of their elements, whereas they are partially ordered by their $t^{app}$ values.

This order assumption might seem like a limitation, but since SECRET aims to explain the execution model differences among SPEs, it has the finest granularity for tuple order information. Please note that some engines impose a total order on the input data based on arrival before processing. The total order assumption might not be met in practice because of network latencies and distributed data sources. In case of out of order tuples, the system can buffer input tuples for a maximum amount of time and then reorder them [91].

**Definition 3 (Batch)** A batch $\mathbb{B}$ of stream elements for a given stream $S$ is a finite subset of $S$, where all $b \in \mathbb{B}$ have an identical $t^{app}$. Each such batch is given a unique batch-id $bid \in \mathbb{N}$ such that, for all $b \in \mathbb{B}$, $b.bid = bid$, indicating that $b$ belongs to the batch that is uniquely identified by $bid$. For tuples $t_1$ and $t_2$ where $t_1.t^{sys} < t_2.t^{sys}$, then $t_1.bid \leq t_2.bid$.

Batches are used to define a further ordering among simultaneous tuples [65]. By definition, all tuples in a given batch have the same $t^{app}$ value, but that does not mean that all tuples with the same $t^{app}$ value are in the same batch. For example, we can have four tuples with $t^{app} = 5$ in two consecutive batches of two tuples each. Therefore, a new batch can arrive without $t^{app}$ advancing. This implies that streams are also partially ordered by their $bid$ values.

**Definition 4 (Window)** A window $W$ over a stream $S$ is a finite subset of $S$. 

Chapter 2. Modeling Streaming Systems Behavior

Windows can be defined in many ways. In this chapter, we will mainly focus on “time-based windows”. In time-based windows, stream elements whose $t_{app}$ values fall into a certain $t_{app}$ interval constitute a window. More formally:

**Definition 5 (Time-based Window)** A time-based window $W = (o, c]$ over a stream $S$ is a finite subset of $S$ containing all data elements $s \in S$ where $o < s.t_{app} \leq c$.

Most systems do not process arbitrary sets of time-based windows, but rather require the windows to have a specific relationship to each other defined by two parameters, size ($\omega$) and slide ($\beta$). More formally:

**Definition 6 (Window Size and Slide)** The set $\mathbb{W}$ of all time-based windows defined over a stream $S$ must satisfy the following two constraints:

1. **Size ($\omega$):** All windows must be the same size, that is, $\forall W = (o, c] \in \mathbb{W}$, $c - o = \omega$.

2. **Slide ($\beta$):** The distance between consecutive windows must be the same. For two windows $W_1 = (o_1, c_1]$ and $W_2 = (o_2, c_2]$, we require that $o_1 \neq o_2$. Furthermore, we say $W_1$ and $W_2$ are consecutive if $o_1 < o_2$ and there is no window $W' = (o', c')$ such that $o_1 < o' < o_2$. For all consecutive windows $W_1$ and $W_2$ in $\mathbb{W}$, we require that $o_2 - o_1 = \beta$.

At $t_{app} = t$, we say a window $W = (o, c]$ is open, if $o < t \leq c$. A window is closed, if $c < t$.

**Definition 7 (Tuple-based Window)** A tuple-based window $W$ over a stream $S$ is a finite subset of $S$ containing data elements $s \in S$ such that $|W| = \omega$ (the number of tuples contained in a tuple-based window is equal to the size value; for an open window, $|W| < \omega$).

As opposed to time-based windows which span over time intervals, the tuple-based windows are defined based on number of tuples.

**Definition 8 (Sliding Window)** A sliding window $W$ over a stream $S$ is a type of window for which the value of slide is smaller than the value of size $\beta < \omega$ (successive windows overlap).

**Definition 9 (Tumbling Window)** A tumbling window $W$ over a stream $S$ is a type of window for which the slide is greater or equal to the size $\omega \leq \beta$ (successive windows do not overlap).

Another assumption we make is that (stream) queries are known a-priori and much of the work presented in Chapter 4 will be based on this assumption.
2.4 The SECRET Model

In this section, we present a model for analysis of the execution semantics of continuous queries in DSMSs. The model is named SECRET, as it captures window-based query execution semantics along four complementary dimensions: ScopE (Section 2.4.0.1), Content (Section 2.4.0.2), REport (Section 2.4.0.3), and Tick (Section 2.4.0.4). Each of these dimensions refers to a certain aspect of window-based execution. As a result, these dimensions give an end-to-end view of what impacts execution semantics from inputs to outputs.

The SECRET model separates the operational aspects of how the DSMS processes streams from the non-procedural effects of that processing. For example, we can talk about how time-based windows are formed independently of their content; this does not depend on any procedural aspect of a system. By contrast, when the system chooses to evaluate results depends heavily on its processing model.

This model also makes a clear separation between data-level issues (e.g., values in a stream), query-level issues (e.g., window size) and system-level issues (e.g., when the system takes an action).

Figure 2.2 illustrates how SECRET can be used to explain the semantics of a given query plan. SECRET is compositional in the same way a query plan is composed of a sequence of operators.

Given a query’s window parameters, ScopE provides information about potential window intervals. Content then helps us map those intervals into actual window contents, for a given input stream. REport states under what conditions those window contents become visible to the query processor for evaluation. Finally, Tick models what drives an DSMS to take action on a given input stream. Tick is the actual entry point to the control loop of our model, creating a chain reaction by invoking Report, which in turn

![SECRET of a Query Plan](image-url)
invokes Content, which builds on Scope (Tick → REport → Content → ScopE).

In this section, we will present the SECRET model for time-based windows: each parameter will be explained in detail, in reverse order, from Scope to Tick.

2.4.0.1 Scope

For a query \( q \), the function \( \text{Scope} \) maps an application time value \( t \) to an interval over which \( q \) should be evaluated. To define \( \text{Scope} \), the active window is defined as the open window with the earliest start time at \( t \).

A value \( t_0 \in \mathbb{T} \) that denotes the application time instant of the start of the very first window in a given system is assumed. Its value is system-specific, and possibly, invocation-specific, since different systems use a different starting point for their application time line depending on environmental factors (recall Example 1 of Section 2.2). Hence, the initial window \( (W_0) \) starts at time \( t_0 \), the next one \( (W_1) \) starts at time \( t_0 + \beta \), and window \( i \) \( (W_i) \) starts at time \( t_0 + i\beta \). Let \( W_i = (o_i, c_i] \) be the \( i^{th} \) window in \( \mathbb{W} \).

Next, the formula that computes \( n \), the index of the earliest open window (i.e., the active window) at time \( t \) is presented.

\[
 n = \max(0, \left\lceil \frac{t-t_0-\omega}{\beta} \right\rceil)
\]

Hence, the start time of \( W_n = (o_n, c_n] \) is \( o_n = t_0 + n\beta \), and \( \text{Scope} \) at time \( t \) is defined as follows:

\[
 \text{Scope}(t) = \begin{cases} 
 \emptyset & \text{if } t < t_0 \\
 (o_n, t] & \text{otherwise}
\end{cases}
\]

Figure 2.3: Scope of a Window
2.4. The SECRET Model

Figure 2.3 illustrates our Scope formulation. As a simple example, assume we have a query \( q \) with a window of size 5 seconds and of slide 2 seconds, to be run on a system with \( t_0 \) of 30 seconds. Then the window scope at \( t = 34 \) seconds is \( \text{Scope}(34) = (30, 34] \), since \( n = 0 \) and \( \alpha_0 = 30 \).

2.4.0.2 Content

Scope defines the interval for query evaluation at application time \( t \). A complementary function Content specifies the set of elements of stream \( S \) that are in this scope. As such, Content makes the mapping from the application time interval representation of a window to a set of data elements. The content of a window at application time instant \( t \) and system time instant \( \tau \) is formally defined as follows:

\[
\text{Content}(t, \tau) = \{ s \in S : s.t^{app} \in \text{Scope}(t) \land s.t^{sys} < \tau \}
\]

Note that unlike Scope, the result of Content depends on actual contents of the input stream, which only become available at run time. Therefore, \( \text{Content}(t, \tau) \) might potentially return different results even if it is called with the same \( t \) value, depending on how much of the input stream is already available (determined by \( \tau \)) when it is invoked.

2.4.0.3 Report

The Report dimension in the SECRET model defines the conditions under which the window contents become visible for further query evaluation and result reporting. DSMSs use different reporting strategies as illustrated in Example 2 of Section 2.2. We have identified four basic reporting strategies:

1. **Content change** (\( R_{cc} \)): reporting is done for \( t \) only if the content has changed since \( t - 1 \).
2. **Window close** (\( R_{wc} \)): reporting is done for \( t \) only when the active window closes (i.e., \( |\text{Scope}(t)| = \omega \)).
3. **Non-empty content** (\( R_{ne} \)): reporting is done for \( t \) only if the content at \( t \) is not empty.
4. **Periodic** (\( R_{pr} \)): reporting is done for \( t \) only if it is a multiple of \( \lambda \), where \( \lambda \) denotes the reporting frequency.

Furthermore, some systems use multiple strategies (e.g., the content must have changed and be non-empty). Hence, four boolean variables are used \( (R_{cc}, R_{wc}, R_{ne}, \ldots) \). This allows for more flexible reporting strategies.
and each of them can be set to true or false by a system. Note that if all of these variables are set to false, then the report will still return $\text{Content}(t, \tau)$. This is the default behavior in the SECRET model. When all four variables are false, the reporting takes place every time it is triggered by the previous step of the model (i.e., $\text{Tick}$ to be defined below).

$$\text{Report}(t, \tau) = \begin{cases} 
\text{Content}(t, \tau) & \text{if } (\neg R_{cc} \lor \text{Content}(t, \tau) \neq \\
\text{Content}(t - 1, \tau)) \\
\land (\neg R_{wc} \lor (|\text{Scope}(t)| = \omega \land \\
t < \max \{s.t^{app}|s \in S \land s.t^{sys} \leq \tau\}) \land \\
\neg R_{ne} \lor \text{Content}(t, \tau) \neq \emptyset) \\
\land (\neg R_{pr} \lor \text{mod}(t, \lambda) = 0) \\
\emptyset & \text{otherwise}
\end{cases}$$

2.4.0.4 Tick

The Tick dimension defines the condition which drives an DSMS to take action on its input (also referred to as “window state change” or “window re-evaluation” [65]). Like $\text{Report}$, $\text{Tick}$ is also part of a system’s internal execution model. While some systems react to individual tuples as they arrive, others collectively react to all or subsets of tuples with the same $t^{app}$ value. During our analysis, we have identified three main ways the different systems “tick”: (a) tuple-driven, where each tuple arrival causes a system to react; (b) time-driven, where the progress of $t^{app}$ causes a system to react (remember Example 3 from 2.2); (c) batch-driven, where either a new batch arrival or the progress of $t^{app}$ causes a system to react. These different Tick behaviors are illustrated in Figure 2.4. We show two time lines for $t^{sys}$ and $t^{app}$. Tuple arrivals are shown on the time line for $t^{sys}$, and window scopes are shown underneath, on the time line for $t^{app}$. Circles around the tuples show the units of tuples that the system will react to at one time, whereas the arrows show to which application time instant those units belong. Note that the tuples are the same in all three figures, and that the four tuples in the middle have the same $t^{app}$ value.

First we list some helper notions which are required by the Tick formulas.

$S(\tau)$ denotes the set of tuples in stream $S$ that has arrived through time instant $\tau$.

\footnote{Remember from the definition of batch that a new batch can arrive without $t^{app}$ advancing.}
2.4. The SECRET Model

Figure 2.4: Tick Models

\[ S(\tau) = \{ s \in S | s.t^{sys} \leq \tau \} \]

\( S_I(\tau) \) denotes the set of tuples in stream \( S \) that has arrived at time instant \( \tau \). There can be at most one such tuple.

\[ S_I(\tau) = \{ s \in S | s.t^{sys} = \tau \} \]

The following mapping functions are used to define \( \text{Tick} \):

\( \text{app}(\tau) \): Given a system time instant \( \tau \), returns the application time value of the tuple that has arrived at \( \tau \).

\[ \text{app}(\tau) = \{ s.t^{app} | s \in S_I(\tau) \land S_I(\tau) \neq \emptyset \} \]
Table 2.1: Tick Example

<table>
<thead>
<tr>
<th>$\tau$</th>
<th>$S(v, t^{app}, t^{sys}, bid)$</th>
<th>tuple-driven</th>
<th>time-driven</th>
<th>batch-driven</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\tau = 30$</td>
<td>$(a, 1, 30, b1)$</td>
<td>Report($x$, 30), $\forall x \in [t_0, 1]$</td>
<td>Report($x$, 30), $\forall x \in [t_0, 1]$</td>
<td>Report($x$, 30), $\forall x \in [t_0, 1]$</td>
</tr>
<tr>
<td>$\tau = 40$</td>
<td>$(b, 2, 40, b2)$</td>
<td>Report($x$, 40), $\forall x \in [1, 2]$</td>
<td>Report($x$, 40), $\forall x \in [1, 2]$</td>
<td>Report($x$, 40), $\forall x \in [1, 2]$</td>
</tr>
<tr>
<td>$\tau = 45$</td>
<td>$(c, 2, 45, b2)$</td>
<td>Report(2, 45)</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>$\tau = 60$</td>
<td>$(d, 2, 60, b3)$</td>
<td>Report(2, 60)</td>
<td>-</td>
<td>Report(2, 60)</td>
</tr>
<tr>
<td>$\tau = 80$</td>
<td>$(e, 2, 80, b3)$</td>
<td>Report(2, 80)</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>$\tau = 90$</td>
<td>$(f, 4, 90, b4)$</td>
<td>Report($x$, 90), $\forall x \in [2, 4]$</td>
<td>Report($x$, 90), $\forall x \in [2, 4]$</td>
<td>Report($x$, 90), $\forall x \in [2, 4]$</td>
</tr>
</tbody>
</table>

$prev_{app}(\tau)$: Given a system time instant $\tau$, returns the application time value of the most recent tuple that has arrived before $\tau$. If no such tuple exists, it returns $t_0$.

$$prev_{app}(\tau) = \max(\max(t_0, s^{app}|s \in S(\tau - 1)))$$

$batch(\tau)$: Given a system time instant $\tau$, returns the batch-id value of the tuple that has arrived at $\tau$.

$$batch(\tau) = \max\{s.bid|s \in S_I(\tau)\}$$

$prev_{batch}(\tau)$: Given a system time instant $\tau$, returns the batch-id value of the most recent tuple that has arrived before $\tau$. If no such tuple exists, it returns 0.

$$prev_{batch}(\tau) = \max(0, \max\{s.bid|s \in S_I(\tau - 1)\})$$

$prev_{tick}(\tau)$: Given a system time instant $\tau$, returns the application time value of the most recent tuple that has arrived before $\tau$ for which the result of the tick was non-empty. If no such tuple exists, it returns $t_0$.

$$prev_{tick}(\tau) = \{\max(t_0, app(max(x|x < \tau \land Tick(x) \neq \emptyset)))\}$$

Based on the above, $Tick$ for each tick model is presented. All formulas follow a similar structure.
In a **tuple-driven** system, Tick is triggered under two conditions: (i) if a tuple arrives whose $t_{app}$ is the same as the previous tick time, or (ii) if a tuple arrives whose $t_{app}$ is greater than the previous tick time.

$$
\text{Tick}(\tau) = \begin{cases} 
\{\text{Report}(app(\tau), \tau)\} & \text{if } S_I(\tau) \neq \emptyset \land \text{prev_tick}(\tau) = app(\tau) \\
\bigcup_{x<app(\tau)} Report(x, \tau) & \text{if } S_I(\tau) \neq \emptyset \land app(\tau) > \text{prev_tick}(\tau) \\
\emptyset & \text{otherwise}
\end{cases}
$$

In a **time-driven** system, there is no need to react to each tuple in a simultaneous sequence separately, therefore, the first condition in the tuple-driven case is skipped. On the other hand, the second condition needs to be triggered if a tuple with a new $t_{app}$ arrives (which means that $t_{app}$ has advanced, to which the system must react).

$$
\text{Tick}(\tau) = \begin{cases} 
\bigcup_{x<app(\tau)} Report(x, \tau) & \text{if } S_I(\tau) \neq \emptyset \land app(\tau) > \text{prev_app}(\tau) \\
\emptyset & \text{otherwise}
\end{cases}
$$

Finally, a **batch-driven** system acts like a modified tuple-driven system. Both the condition for simultaneous tuples as well as for a tuple with a new $t_{app}$ need to be checked. The only difference is that an additional check is needed to verify if the new tuple arrival initiates a new batch (compare the new batch-id with the previous one).

$$
\text{Tick}(\tau) = \begin{cases} 
\{\text{Report}(app(\tau), \tau)\} & \text{if } S_I(\tau) \neq \emptyset \land \text{prev_tick}(\tau) = app(\tau) \land batch(\tau) > \text{prev_batch}(\tau) \\
\bigcup_{x<app(\tau)} Report(x, \tau) & \text{if } S_I(\tau) \neq \emptyset \land batch(\tau) > \text{prev_batch}(\tau) \\
\emptyset & \text{otherwise}
\end{cases}
$$

In order to illustrate how the $\text{Tick}$ formulation works, in Table 2.1 we present a sample trace of the model for the scenario shown in Figure 2.4. The table shows when exactly
each of the three models triggers \textit{Report} and with which time values. As expected, the time-driven model invokes \textit{Report} only when time advances, one for each time point including the gap time ($t_{app} = 3$). The tuple-driven model, on the other hand, invokes \textit{Report} at every new tuple arrival. Finally, the batch-driven model invokes \textit{Report} at every new batch arrival as well as time advance ($t_{app} = 3$).

In this section we only focused on describing SECRET for analyzing the execution semantics of queries with \textit{time-based} windows. Possible extensions to the model with respect to input aspects (system time-based windows, synchronized timestamps, out-of-order streams), query aspects (tuple-based windows, binary operators), or system aspects are presented in [32].

### 2.5 Modeling Systems Behavior

#### 2.5.1 State-of-the-art DSMSs in SECRET

The SECRET model was tested using five different DSMSs (StreamBase, STREAM, Coral8, StreamInsight and MXQuery), which are representative in the area and are widely used. The parameter values obtained are presented in Table 2.2.

StreamBase commercialized the Aurora/Borealis academic prototypes, and therefore, its basic execution model descends from those of these two systems. Similarly, the Oracle CEP system directly follows STREAM’s query execution model [65]. Coral8 is a

<table>
<thead>
<tr>
<th>DSMS</th>
<th>$t_0$ (Scope)</th>
<th>Report</th>
<th>Tick</th>
</tr>
</thead>
<tbody>
<tr>
<td>STREAM [10]</td>
<td>$t_{i+1} - \omega$</td>
<td>window close &amp; content change &amp; non-empty &amp; $\lambda=1$</td>
<td>time-driven</td>
</tr>
<tr>
<td>Coral8 [1]</td>
<td>$\left\lceil \frac{t_{i+1} - \omega}{\beta} \right\rceil - 1$</td>
<td>content change &amp; non-empty &amp; $\lambda=1$</td>
<td>batch-driven</td>
</tr>
<tr>
<td>StreamBase [11]</td>
<td>$\left\lceil \frac{t_{i+1} - \omega}{\beta} \right\rceil - 1$</td>
<td>window close &amp; non-empty &amp; $\lambda=1$</td>
<td>tuple-driven</td>
</tr>
<tr>
<td>StreamInsight [4]</td>
<td>Based on application time</td>
<td>window close &amp; non-empty</td>
<td>time-driven</td>
</tr>
<tr>
<td>MXQuery [6]</td>
<td>Dependent on the input stream</td>
<td>window close &amp; non-empty</td>
<td>tuple-driven</td>
</tr>
</tbody>
</table>

**Table 2.2:** SECRET Parameters Values for Different DSMSs
widely used commercial system and its model is also substantially different from the two families of DSMSs mentioned above. More recently, Microsoft proposed StreamInsight, a system which exposes a temporal model and whose execution is driven by special events in the streams which mark the advance of application time (SECRET can explain the behavior of StreamInsight in the presence of point-based events).

Next, we present in short how SECRET explains the behavior observed in the examples from Section 2.2. A detailed description of the experimentation with the StreamBase, STREAM and Coral8 systems and which also explains the examples in Section 2.2 in more depth can be found in [32].

**Example 1: Window construction in a DSMS.** The first example in Section 2.2 shows that the same query (compute the average value of tuples in a time-based tumbling window with size and slide of 3 seconds) may produce different results when it is run multiple times on the same input stream, in a given SPE (StreamBase). It turns out that, before each run, StreamBase re-assigns application time values to tuples according to the current system time. While doing that, it still takes the tuples’ own source-assigned time values as basis, using exactly the same time differences between consecutive tuples (i.e., only the absolute time values change; their relative values stay the same). The value of \( t_0 \) in StreamBase depends on the absolute value of the application time of the first tuple. As such, \( t_0 \) affects window scopes, and therefore contents, and eventually the query results.

**Example 2: Evaluation differences across DSMSs.** The query continuously computes the average value for the tuples arriving in a sliding window of size 5 seconds. Our experimentation showed that StreamBase evaluates a query when a window closes and the window is non-empty; Coral8 evaluates when the content of the window changes and the window is non-empty; and STREAM evaluates when all of these conditions are true. Accordingly, each system evaluates the AVERAGE operation at different instants, with different window scope and content values, and this leads to different query results. Therefore, we can conclude that SECRET’s explanation for the difference observed in this example is primarily due to the differences between the Evaluation parameter values of those systems.

MXQuery is an XQuery based streaming engine that we developed and used for the research depicted in this dissertation. This system is presented in great detail in Chapter 3. MXQuery uses a special class of windows, called predicate-based windows. A detailed description of MXQuery’s behavior description in SECRET and the required model extensions can be found in Section 3.6.
2.5.2 Using SECRET to Analyze a new DSMS

The first step in analyzing a DSMS with our SECRET model is to find out what value each SECRET parameter should take for the given system. If the required knowledge about the system is not readily available, these values can be obtained by executing a set of queries against a range of inputs. The input stream should have irregularities due to gaps in application time and due to simultaneous tuples with common application times. Furthermore, queries should include some with windows that slide each time unit, windows with slide parameters (i.e., having a slide value greater than minimum window unit, but less than the window size), or tumbling windows. By executing different configurations of these input and query properties, the SECRET parameter values can be revealed.

2.6 Summary

The exposition in this chapter can be seen as a survey of the state-of-the-art concepts and techniques in data stream processing.

We show that there are high differences in the semantics of the different streaming systems and that a way to formally describe their behavior is necessary. We start by giving the definitions of the basic concepts in stream processing: data stream, time domain, window, batch etc. Using these definitions as basis, we formally present the execution semantics of the time-based windowed execution.

The formal framework presented, the SECRET model, is used to describe the behavior of streaming systems along four dimensions: Scope, Content, Report and Tick, which model the execution from window construction to actual execution and result generation. The SECRET model has been successfully applied [32] to a series of representative Data Stream Management Systems like StreamBase [10], STREAM [22], Coral8 [1], StreamInsight [4], MXQuery [6]. We present the values these systems take for each of the parameters and as such, we describe the systems’ behavior as instantiations of this model.

The formal description in this chapter clearly positions our work in today’s stream processing context. Moreover, understanding how the streaming systems work helps us capture those systems’ transactional behaviors, an issue covered in great detail in Chapter 5.
Chapter 3

Continuous Processing in XQuery

This chapter establishes the context of our work on storage management for streaming systems: MXQuery, a Java-based open source XQuery engine. Two extensions are presented which make it possible to use XQuery for continuous processing of infinite inputs. The first extension allows the definition and processing of different kinds of windows over an input sequence; i.e., tumbling, sliding, and landmark windows. The second extends the XQuery data model (XDM) to support infinite sequences. The efficient implementation of these extensions requires, among others, proper main memory management. This chapter gives the details of the extensions implementation in MXQuery and presents the results of running the Linear Road Benchmark on top of the extended XQuery engine.

3.1 Introduction

An important class of data streams is represented by communication data, which is often depicted by streams of XML items (e.g., RSS and Atom feeds). For many applications it is important to detect certain patterns in that stream: a credit card company, for instance, might be interested to learn if a credit card is used particularly often during one week as compared to other weeks. Implementing this audit involves a continuous window query.

Arguably, XQuery [28] is the most promising programming language for processing XML data (XQuery 1.0 is a recommendation of the W3C [15]). Even though XQuery 1.0 is extremely powerful (it is turing-complete), it lacks support for window queries and continuous queries.
Both window operators and continuous queries have been studied extensively in the SQL world; proposals for SQL extensions are described in, e.g., [74] [22] [36] [12]. That work, however, is not applicable in the XML world. The first and obvious reason is that SQL is not appropriate to process XML data. It can neither directly read XML data, nor can SQL generate XML data if the output is required to be XML (e.g., an RSS feed). Another reason is that the SQL extensions are not expressive enough to solve certain use cases, even if all data were relational (e.g., CSV). The SQL extensions have been specifically tuned for particular streaming applications and support only simple window definitions based on time or window size as presented in Chapter 2 (some systems, e.g., StreamBase [11], allow value-based definition of windows, but with restrictions: the fields on which the windows are defined have to contain monotonically increasing values).

In this chapter, we present two extensions to XQuery which allow it to support window queries and continuous queries. The result is a powerful extension that is appropriate for all use cases, including the classic streaming applications for which the SQL extensions were designed and the more progressive use cases of the XML world (i.e., RSS feeds and document management). The goal is to obtain an implementation performance for these extensions which should be comparable to the performance of continuous SQL queries: there should not be a performance penalty for using XQuery. Obviously, our work is based on several ideas and the experience gained in the SQL world from processing data streams. Nevertheless, there are important differences to the SQL data stream approach. In fact, it is easier to extend XQuery because the XQuery data model (XDM), which is based on sequences of items [46], is already a good match to represent data streams and windows. As a result, the extensions compose nicely with all existing XQuery constructs and all existing XQuery optimization techniques remain relevant. In contrast, extending SQL for window queries involves a more drastic extension of the relational data model and a great deal of effort in the SQL world has been spent on defining the right mappings for these extensions. As an example, CQL [22] defines specific operators that map between streams and relations.

In summary, this chapter makes the following contributions:

- **Window and Continuous Queries**: The syntax and semantics of the `FORSEQ` clause for defining and processing complex windows using XQuery are detailed (these extensions were accepted for the XQuery 1.1 standard [34]). In addition, a simple extension to the XQuery data model (XDM) in order to process infinite data streams and use XQuery for continuous queries is presented.
3.2. Usage Scenarios

- **Implementation Design:** The implementation details of these extensions in an open source XQuery engine (MXQuery) and the optimization techniques applicable in order to obtain good performance (in particular, main memory management).

- **Linear Road Benchmark:** The Linear Road Benchmark implementation details in MXQuery as well as the results of running it. The benchmark results confirm that the proposed XQuery extensions can be implemented efficiently.

The remainder of this chapter is organized like in the following: Section 3.2 presents an use case scenario motivating this work. The FORSEQ clause is presented in detail in Section 3.3, followed by the extension to the XQuery data model in Section 3.4. The implementation of the extensions is presented in Section 3.5 followed by a MXQuery's execution described using the SECRET model (Chapter 2) in Section 3.6. A detailed description of the Linear Road Benchmark can be found in Section 3.7, the benchmark's implementation in MXQuery in Section 3.8 and the experimental results in Section 3.9. The related work is presented in Section 3.10. The chapter ends with a summary in Section 3.11.

3.2 Usage Scenarios

3.2.1 Motivating Example

The following simple example illustrates the need for an XQuery extension. It involves a blog with RSS items of the following form:

```xml
<rss:item>
... <rss:author>...</rss:author> ...
</rss:item>
```

Given such a blog, the goal is to find all *annoying authors* who have posted three consecutive items in the RSS feed. Using XQuery 1.0, this query can be formulated as shown in Figure 3.1. This query involves a three-way self join which is not only tedious to specify but also difficult to optimize. In contrast, Figure 3.2 shows this query using the FORSEQ clause. This clause partitions the blog into sequences of postings of the same author (i.e., windows) and iterates over these windows (Lines 1-4 of Figure 3.2).
for $first at $i in $rssfeed
let $second := $rssfeed[$i+1],
let $third := $rssfeed[$i+2]
where ($first/author eq $second/author) and
    ($first/author eq $third/author)
return $first/author

Figure 3.1: Annoying Authors: XQuery 1.0

forseq $w in $rssfeed tumbling window
    start curItem $first when fn:true()
    end nextItem $lookAhead when
        $first/author ne $lookAhead/author
where count($w) ge 3
return $w[1]/author

Figure 3.2: Annoying Authors: Extended XQuery

If a window contains three or more postings, then the author of this window of postings is annoying and the author is returned (Lines 5 and 6). The syntax and semantics of the FORSEQ clause are presented in detail in Section 3.3 and need not be understood at this point. For the moment, it is only important to observe that this query is straightforward to implement and can be executed in linear time or better, if the right indexes are available. Furthermore, the definition of this query can easily be modified if the definition of annoying author is changed from, say, three to five consecutive postings. In comparison, additional self-joins must be implemented in XQuery 1.0 in order to implement this change.\footnote{In fact, the two queries of Figures 3.1 and 3.2 are not equivalent. If an author posts four consecutive postings, this author is returned twice in the expression of Figure 3.1 whereas that author is returned only once in Figure 3.2}

3.2.2 Other Applications

The management of RSS feeds is one application that drove the design of the XQuery extensions. There are several other areas; the following is a non-exhaustive list of further application scenarios:
3.3. FORSEQ Clause

- **Web Log Auditing:** In this scenario, a window contains all the actions of a user in a session (from login to logout). The analysis of a Web log involves, for example, the computation of the average number of clicks until a certain popular function is found. Security audits and market-basket analyses can also be carried out on user sessions.

- **Financial Data Streams:** Window queries can be used in order to carry out fraud detection, algorithmic trading and finding opportunities for arbitrage deals by computing call-put parities [47].

- **Social Games / Gates:** An RFID reader at a gate keeps track of the people that enter and exit a building. People are informed if their friends are already in the building when they themselves access the building.

- **Sensor Networks:** Window queries are used in order to carry out data cleaning. For instance, the average of the last five measurements (possibly, disregarding the minimum and maximum) is reported, rather than reporting each individual measurement [67].

- **Document Management:** Different text elements (e.g., paragraphs, tables, figures) are grouped into pages. In the index, page sequences such as 1, 2, 3, 4, 7 are reformatted into 1-4, 7 [69].

Around sixty different use cases were compiled in these application areas in [47]. All these examples have in common that they cannot be implemented well using Version 1.0 of XQuery without support for windows. Furthermore, many examples of [47] cannot be processed using SQL, even considering the latest extensions proposed in [74] [22] [36] [12] because these examples require powerful constructs in order to define window boundaries. Most of these use cases involve other operators such as negation, existential and universal quantification, aggregation, correlation, joins, and transformation in addition to window functions. XQuery already supports all these operators which makes XQuery a natural candidate to extend, rather than inventing a new language from scratch in order to address these applications.
Chapter 3. Continuous Processing in XQuery

\[ FLWORExpr ::= (ForseqClause|ForClause|LetClause) + WhereClause? OrderByClause? "return" ExprSingle \]

\[ ForseqClause ::= "forseq" "$" VarName TypeDeclaration? "$" in ExprSingle WindowType? (",", "$" VarName TypeDeclaration? "$" in ExprSingle WindowType?)* \]

\[ WindowType ::= ("tumbling window"|"sliding window"|"landmark window") \]

\[ StartExpr EndExpr \]

\[ StartExpr ::= "start" WindowVars? "when" ExprSingle \]

\[ EndExpr ::= "force"? "end" WindowVars? "when" ("newstart"|ExprSingle) \]

\[ WindowVars ::= ("position"|"curItem"|"prevItem"|"nextItem") "$" VarName TypeDeclaration? (",", ("position"|"curItem"|"prevItem"|"nextItem") "$" VarName TypeDeclaration?)* \]

Figure 3.3: Grammar of Extended FLWOR Expression

3.3 FORSEQ Clause

3.3.1 Basic Idea

Figure 3.2 gives an example of the FORSEQ clause. The FORSEQ clause is an extension of the famous FLWOR expressions of XQuery. It is freely composable with other FOR, LET, and FORSEQ clauses. Furthermore, FLWOR expressions that involve a FORSEQ clause can have an optional WHERE and/or ORDER BY clause and must have a RETURN clause, just as any other FLWOR expression. A complete grammar of the extended FLWOR expression is given in Figure 3.3.

Like the FOR clause, the FORSEQ clause iterates over an input sequence and binds a variable with every iteration. The difference is that the FORSEQ clause binds the variable to a sub-sequence (aka window) of the input sequence in each iteration, whereas the FOR clause binds the variable to an item of the input sequence. To which sub-sequences the variable is bound is determined by additional clauses. The additional TUMBLING WINDOW, START, and END clauses of Figure 3.2 for instance, specify that
$w$ is bound to each consecutive sub-sequence of postings by the same author. In that example, the window boundaries are defined by the change of author in postings in the \texttt{WHEN} clause of the \texttt{END} clause (details of the semantics are given in the next subsection).

The running variable of the \texttt{FORSEQ} clause ($w$ in the example) can be used in any expression of the \texttt{WHERE}, \texttt{ORDER BY}, \texttt{RETURN} clauses or in expressions of nested \texttt{FOR}, \texttt{LET}, and \texttt{FORSEQ} clauses. The only requirement is that those expressions must operate on sequences (rather than individual items or atomic values) as input. In Figure 3.2 for example, the \texttt{count} function is applied to $w$ in the \texttt{WHERE} clause in order to determine whether $w$ is bound to a series of postings of an annoying author (three or more postings).

As shown in Figure 3.3 FLWOR expressions with a \texttt{FORSEQ} clause can involve an \texttt{ORDER BY} clause, just like any other FLWOR expression. Such an \texttt{ORDER BY} clause specifies in which order the sub-sequences (aka windows) are bound to the running variable. By default, and in the absence of an \texttt{ORDER BY} clause, the windows are bound in ascending order of the position of the last item of a window. If two (overlapping) windows end in the same item, then their order is implementation-defined. For instance, annoying authors in the example of Figure 3.2 are returned in the order in which they made annoying postings. This policy naturally extends the order in which the \texttt{FOR} clause orders the bindings of its input variable in the absence of an \texttt{ORDER BY} clause.

The \texttt{FORSEQ} clause does not involve an extension or modification of the XQuery data model (XDM) [46]. Binding variables to sequences is naturally supported by XDM. As a result, the \texttt{FORSEQ} clause is fully composable with all other XQuery expressions and no other language adjustments need to be made. In contrast, extending SQL with windows involves an extension to the relational data model and, as mentioned in the introduction, a great deal of effort has been invested into defining the exact semantics of such window operations in such an extended relational data model.

Furthermore, the XQuery type system does not need to be extended, and static typing for the \texttt{FORSEQ} clause is straightforward. To give a simple example, if the static type of the input is \texttt{string*}, then the static type of the running variable is \texttt{string+}. The “+” quantifier is used because the running variable is never bound to the empty sequence. To give a more complex example, if the static type of the input sequence is \texttt{string*}, \texttt{integer*} (i.e., a sequence of strings followed by a sequence of integers), then the static type of the running variable is: \texttt{(string+,integer* | string*,integer+)}; i.e., a sequence of strings, a sequence of integers, or a sequence of strings followed by integers (similarly, simple rules apply to the other kinds of variables that can be bound by the \texttt{FORSEQ}
3.3.2 Types of Windows

Previous work on extending SQL to support windows has identified different kinds of windows; i.e., tumbling windows, sliding windows (whose formal definitions can be found in Section 2.3) and landmark windows [58]. Figure 3.4 shows examples of these three types of windows; as a reference, Figure 3.4 also shows how the traditional FOR and LET clauses of XQuery work. The three types of windows differ in the way they overlap: tumbling windows do not overlap; sliding windows overlap, but have disjoint first items (because the window boundaries are defined through predicates, it is possible for two windows to share the last item); landmark windows can overlap in any way. This section describes how the FORSEQ clause can be used to support these kinds of windows.

Furthermore, previous work on windows for SQL proposed alternative ways to define the window boundaries (start and end of a window). Here, all published SQL extensions [74, 22, 36, 12] propose to define windows based on size (i.e., number of items), duration (time span between the arrival of the first and last item) or value (the value difference of certain attributes corresponding to the data items defining the start and the end of a window). Our approach for XQuery is more general and is based on using predicates in order to define window boundaries. Size and time constraints can easily be expressed in such a predicate-based approach (examples are given in the remainder of this chapter). Furthermore, more complex conditions which involve any property of an item (e.g., the author of a posting in a blog) can be expressed using FORSEQ. One of the consequences of having predicate-based window boundaries is that the union of all windows does not necessarily cover the whole input sequence; that is, it is possible
that an input item is not part of any window.

### 3.3.2.1 Tumbling Windows

The first kind of window supported by the `FORSEQ` clause is a so-called *tumbling window* \[84\]. Tumbling windows partition the input sequence into disjoint sub-sequences, as shown in Figure 3.4. An example of a query that involves a tumbling window is given in Figure 3.2 in which each window is a consecutive sequence of blog postings of a particular author. Tumbling windows are indicated by the `TUMBLING WINDOW` keyword as part of the `WindowType` declaration in the `FORSEQ` clause (Figure 3.3).

The boundaries of a tumbling window are defined by (mandatory) `START` and `END` clauses. These clauses involve a `WHEN` clause which specifies a predicate. Intuitively, the `WHEN` condition of a `START` clause specifies when a window should start. For each item in the sequence this clause is checked for a match. Technically, a match exists if the effective Boolean value (EBV) \[43\] of the `WHEN` condition evaluates to true. As long as no item matches, no window is started and the input items are ignored. Thus, it is possible that certain items are not part of any window. Once an item matches the `WHEN` condition of the `START` clause, a new window is opened and the matching item is the first item of that window. At this point, the `WHEN` condition of the `END` clause is evaluated for each item, including the first item. Again, technically speaking, the EBV is computed. If an item matches the `END` condition, that item is the last item of the window.

Any XQuery expression can be used in a `WHEN` clause (Figure 3.3), including expressions that involve existential quantification (on multiple sub-elements) or nested FLWOR expressions (possibly with `FORSEQ`). The semantics of the `START` and `END` clauses for tumbling windows can best be shown using the automaton depicted in Figure 3.5. The condition of the `START` clause is not checked for an open window. A window is only closed when its `END` condition is fulfilled or at the end of the input sequence.

To give two simple examples, the `FOR` clause of XQuery can be simulated with a `FORSEQ` clause as follows:

```
forseq $w in $seq tumbling window
  start when fn:true()  
  end when fn:true() ... 
```

That is, each item opens and immediately closes a new window (both `START` and `END` conditions are set to true) so that each item represents a separate window. The `LET` clause can be simulated with a `FORSEQ` clause as follows:
Figure 3.5: Window Automaton

forseq $w$ in $\text{seq}$ tumbling window
   start when fn:true()
   end when fn:false() ...

The first item of the input sequence opens a new window (START condition is true) and this window is closed at the end of the input sequence. In other words, the whole input sequence is bound to the running variable as a single window.

In order to specify more complex predicates, both the START and the END clause allow the binding of new variables. The first kind of variable identifies the position of a potential first item (in the START clause) or last item (in the END clause), respectively. For instance, the following FORSEQ clause partitions the input sequence ($\text{seq}$) into windows of size three; the last window might be smaller:

forseq $x$ in $\text{seq}$ tumbling window
   start position $i$ when fn:true()
   end position $j$ when $j-i$ eq 2 ...

For each window, $i$ is bound to the position of the first item of the window in the input sequence; i.e., $i$ is 1 for the first window, 4 for the second window, and so on. Correspondingly, $j$ is bound to the position of the last item of a window as soon as that item has been identified; i.e., $j$ is 3 for the first window, 6 for the second window, and so on. In this example, $j$ might be bound to an integer that is not a multiple of three for the last window at the end of the input sequence.
Both $i$ and $j$ can be used in the WHEN expression of the END clause. Naturally, only variables bound by the START clause can be used in the WHEN condition of the START clause. Furthermore, in-scope variables (e.g., $\text{seq}$ in the examples above) can be used in the conditions of the START and END clauses. The scope of the variables bound by the START and END clauses is the whole remainder of the FLWOR expression. For instance, $i$ and $j$ could be used in the WHERE, RETURN and ORDER BY clauses or in any nested FOR, LET, or FORSEQ clauses in the previous example.

In addition to positional variables, variables that refer to the previous (prevItem), current (currItem), and next items (nextItem) of the input sequence can be bound in the START and END clause. In the expression of Figure 3.2, for instance, the END clause binds variable $\text{lookAhead}$ to the item that comes after the last item of the current window (i.e., the first item of the next window). These extensions are syntactic sugar because these three kinds of variables can be simulated using positional variables; e.g., \text{end nextItem $\text{lookAhead}$ and when $\text{lookAhead}$ ...} is equivalent to \text{end position $j$ when $\text{seq}[$j+1$]$ ...}. In both cases, an out-of-scope binding (at the end of the input sequence) is bound to the empty sequence.

### 3.3.2.2 Sliding and Landmark Windows

In the SQL world, several different kinds of windows were identified and found useful in practice. In addition to tumbling windows, so-called sliding (Section 2.3) and landmark windows are needed in many applications. In contrast to tumbling windows, both sliding and landmark windows can overlap. The difference between sliding and landmark windows is that two sliding windows never have the first item in common, whereas landmark windows do not have such a constraint (Figure 3.4). A more formal definition of landmark windows is given in [84].

Based on this experience, the FORSEQ clause also supports sliding and landmark windows. As shown in Figure 3.3, only the TUMBLING WINDOW keyword needs to be replaced in the syntax. Again, (mandatory) START and END clauses specify the window boundaries. The semantics are analogous to the semantics of the START and END clauses of a tumbling window (Figure 3.5). The important difference is that each item potentially opens one (for sliding windows) or several new windows (for landmark windows) so that conceptually, several automata need to be maintained at the same time.
3.3.3 General Sub-sequences

In its most general form, the \texttt{FORSEQ} clause takes no additional clauses; i.e., no specification of the window type and no \texttt{START} and \texttt{END} clauses. In this case, the syntax is as follows (Figure 3.3):

\begin{verbatim}
forseq $w$ in $\$seq \ldots
\end{verbatim}

This general version of the \texttt{FORSEQ} clause iterates over all possible sub-sequences of the input sequence. These sub-sequences are not necessarily consecutive. For example, if the input sequence contains the items \texttt{(a, b, c)}, then the general \texttt{FORSEQ} carries out seven iterations ($2^n - 1$, with $n$ the size of the input sequence), thereby binding the running variable to the following sub-sequences: \texttt{(a), (a,b), (b), (a,b,c), (a,c), (b,c), and (c)}. Again, the sequences are ordered by the position of their last item (Section 3.3.1); i.e., the \texttt{(a)} sequence comes before the sequences that end with a \texttt{“b”} which in turn come before the sequences that end with a \texttt{“c”}. Again, the running variable is never bound to the empty sequence.

This general \texttt{FORSEQ} clause is the most powerful variant. Landmark, sliding, and tumbling windows can be seen as special cases of this general \texttt{FORSEQ}. We use a special syntax for these three kinds of windows because use cases that need these three types of windows are frequent in practice [47]. Furthermore, the general \texttt{FORSEQ} clause is difficult to optimize. Use cases for which landmark, sliding, and tumbling windows are not sufficient, are given in [96] for RFID data management. In those use cases, regular expressions are needed in order to find patterns in the input stream. Such queries can be implemented using the general \texttt{FORSEQ} clause by specifying the relevant patterns (i.e., regular expressions) in the \texttt{WHERE} clause of the general \texttt{FORSEQ} expression.

3.3.4 Syntactic Sugar

There are use cases which benefit from additional syntactic sugar. The following paragraph presents such syntactic sugar.

3.3.4.1 End of Sequence

As mentioned in Section 3.3.2.1 by default, the condition of the \texttt{END} clause is always met at the end of a sequence. That is, the last window will be considered even if its last item does not match the \texttt{END} condition. In order to specify that the last window should
only be considered if its last item indeed matches the END condition, the END clause can be annotated with a special keyword FORCE (Figure 3.3). The FORCE keyword is syntactic sugar because the last window could also be filtered out by repeating the END condition in the WHERE clause.

3.3.4.2 NEWSTART

There are several use cases in which the START condition should implicitly define the end of a window. For example, the day of a person starts every morning when the person’s alarm clock rings. Implicitly, this event ends the previous day, even though it is not possible to concretely identify a condition that ends the day. In order to implement such use cases, the WHEN condition of the END clause can be defined as NEWSTART. As a result, the START condition (rather than the END condition) is checked for each open window in order to determine when a window should be closed. Again, the NEWSTART option is syntactic sugar and avoids that the condition of the START clause is replicated in the END clause.

3.4 Continuous XQuery

The second extension makes XQuery a candidate language to specify continuous queries on potentially infinite data streams. In fact, this extension is orthogonal to the first extension, the FORSEQ clause: both extensions are useful independently, although we believe that they will often be used together in practice.

The XQuery data model (XDM) is extended [46] to support infinite sequences as legal instances of XDM. As a result, XQuery expressions can take an infinite sequence as input. Likewise, XQuery expressions can produce infinite sequences as output. A simple example illustrates this extension. A temperature sensor in an ice cream warehouse produces measurements of the following form every minute: <temp>-8</temp>. Whenever a temperature of ten degrees or higher is measured, an alarm should be raised. If the stream of temperature measurements is bound to variable $s$, this alarm can be implemented using the following (continuous) XQuery expression:

```xml
declare variable $s as (temp)** external;
for $m in $s where $m ge 10
return <alarm> { $m } </alarm>
```
In this example, variable $s$ is declared to be an external variable that contains a potentially infinite sequence of temperature measurements (indicated by the two asterisks). Since $s$ is bound to a (potentially) infinite sequence, this expression is illegal in XQuery 1.0 because the input is not a legal instance of the XQuery 1.0 data model. Intuitively, however, it should be clear what this continuous query does: whenever a temperature above 10 is encountered, an alarm is raised. The input sequence of the query is infinite and so is the output sequence.

Extending the data model of a query language is a critical step because it involves refining the semantics of all constructs of the query language for the new kind of input data. Fortunately, this particular extension of XDM for infinite sequences is straightforward to implement in XQuery. The idea is to extend the semantics of non-blocking functions (e.g., for, forseq, let, distinct-values, all path expressions) for infinite input sequences and to specify that these non-blocking functions (potentially) produce infinite output. Other non-blocking functions such as retrieving the $i$th element (for some integer $i$) are also defined on infinite input sequences, but generate finite output sequences. Blocking functions (e.g., order by, last, count, some) are not defined on infinite sequences; if they are invoked on (potentially) infinite sequences, then an error is raised. Such violations can always be detected statically (i.e., at compile-time). For instance, the following XQuery expression would not compile because the \texttt{fn:max()} function is a blocking function that cannot be applied to an infinite sequence:

\begin{verbatim}
declare variable $s$ as (temp)** external;
fn:max($s)
\end{verbatim}

Extending XDM does not involve an extension of the XQuery type system. \texttt{(temp)**} is the same type as \texttt{(temp)*}. The two asterisks are just an annotation to indicate that the input is potentially infinite. These annotations (and corresponding annotations of functions in the XQuery function library) are used in the data flow analysis of the compiler in order to statically detect the application of a blocking function on an infinite sequence (Section 3.5).

A frequent example in which the \texttt{FORSEQ} clause and this extension for continuous query are combined, is the computation of moving averages. Moving averages are useful in, e.g., sensor networks as described in Section 3.2.2; rather than reporting the current measurement, an average of the current and the last four measurements is reported for every new measurement. Moving averages can be expressed as follows:
3.5 Implementation

This section describes how we extend the MXQuery [6] engine, an existing Java-based open-source XQuery engine, in order to implement the `FORSEQ` clause and continuous XQuery processing. We further use the extended MXQuery engine in order to run the Linear Road Benchmark (Section 3.8).

3.5.1 MXQuery

The MXQuery engine was developed as part of a collaboration between Siemens and ETH Zurich. The main purpose of the MXQuery engine is to provide an XQuery implementation for small and embedded devices; in particular, in mobile phones and small gateway computers. Within Siemens, for instance, the MXQuery engine has been used as part of the Siemens Smart Home project in order to control lamps, blinds, and other devices according to personal preferences and weather information via Web services. MXQuery has also been used as a reference implementation for the XQuery Update language and XQueryP, the XQuery scripting extensions.

Since MXQuery was designed for embedded systems, it has a simple and flexible design. The parser is a straightforward XQuery parser and creates an expression tree. In a functional programming language like XQuery, the expression tree plays the same role as the relational algebra expression (or operator tree) in SQL. The expression tree is normalized using the rules of the XQuery formal semantics [43]. After that, the expression tree is optimized using heuristics (MXQuery does not have a cost-based query optimizer). For optimization, MXQuery only implements a dozen of essential query rewrite rules such as the elimination of redundant sorts and duplicate elimination. The final step of compilation is code generation during which each expression of the expression tree is translated into an iterator that can be interpreted at run-time.

\[\text{declare variable } \$\text{seq as (xs:int)** external;}\]
\[\text{forseq } \$\text{w in } \$\text{seq sliding window}\]
\[\text{start position } \$\text{s when } \text{fn:}\text{true}()\]
\[\text{end position } \$\text{e when } \$\text{e} - \$\text{s eq 4}\]
\[\text{return } \text{fn:avg}(\$\text{w})\]
As in SQL, iterators have an `open()`, `next()`, `close()` interface \(^{[59]}\); that is, each iterator processes its input one *item* at a time and only processes as much of its input as necessary. The iterator tree is often also called *plan*, thereby adopting the SQL query processing terminology. Figure 3.6 gives an example plan for the `FOR` query that raises an alarm when the temperature raises above ten degrees (first query of Section 3.4). Like Saxon \(^{[68]}\), BEA’s XQuery engine \(^{[49]}\), and FluXQuery \(^{[70]}\), MXQuery was designed as a *streaming* XQuery engine.

As shown in Figure 3.7, MXQuery can take input data from multiple sources; e.g., databases, the file system, or streaming data sources such as RSS streams or message queues. In order to take input from different sources, the data sources must implement the iterator API. That way, data from data sources can be processed as part of a query plan at runtime. MXQuery already has predefined iterator implementations in order to integrate SAX, DOM, StaX, and plain XML from the file system. Furthermore, MXQuery has predefined iterator implementations for CSV and relational databases. In order to access the data as part of an XQuery expression, an external XQuery variable is declared for each data source. A special Java API is used in order to bind the iterator...
that feeds the data from the data source to the external variable.

As depicted in Figure 3.7, MXQuery also has an internal, built-in store in order to materialize intermediate results (e.g., for sorting of windows). In the current release of MXQuery, this store is implemented fully in main memory.

Although MXQuery was particularly designed for embedded systems, its architecture is representative. The following subsections describe how we extended the MXQuery engine in order to implement the \texttt{FORSEQ} clause and work on infinite data streams.

### 3.5.2 Plan of Attack

In order to implement the \texttt{FORSEQ} clause, the following adaptations were made to the MXQuery engine:

- The parser was extended according to the production rules of Figure 3.3. This extension was straightforward and needs no further explanation.

- The optimizer was extended using heuristics to rewrite \texttt{FORSEQ} expressions. The heuristics are described in Section 3.5.4.

- The runtime system was extended with four new iterators that implement the three different kinds of windows and the general \texttt{FORSEQ}. Furthermore, the MXQuery internal main-memory store (Figure 3.7) was extended in order to implement windows. These extensions are described in Section 3.5.3.

To support continuous queries and infinite streams, the following extensions were made:

- The parser was extended in order to deal with the new \texttt{**} annotation, which declares infinite sequences.

- The data flow analyses of the compiler were extended in order to identify errors such as the application of a blocking operator (e.g., count) to a potentially infinite stream.

- The runtime system was extended in order to synchronize access to data streams and merge/split streams.

The first two extensions (parser and type system) are straightforward and can be implemented using standard techniques of compiler construction [19] and database query optimization [85]. The third extension is significantly more complex, but not specific
to XQuery. For our prototype implementation, we followed as much as possible the approach taken in the Aurora project [18]. We will refer again to this extension in the Chapter 4, where a more detailed description of our synchronization solutions is presented.

### 3.5.3 Runtime System

#### 3.5.3.1 Window Iterators

The implementation of the FORSEQ iterators for tumbling, sliding, and landmark windows is similar to the implementation of a FOR iterator: all these iterators implement second-order functions which bind variables and then execute a function on those variable bindings. All XQuery engines have some sort of mechanics to implement such second-order functions and these mechanics can be leveraged for the implementation of the FORSEQ iterators. The MXQuery engine has similar mechanics as those described in [49] to implement second-order functions: in each iteration, a FORSEQ iterator binds a variable to a sequence (i.e., window) and then it executes a function on that variable binding. The function to execute is implemented as an iterator tree as shown in Figure 3.6 for a FOR iterator. This iterator tree encodes FOR, LET, and FORSEQ clauses (if any) as well as WHERE (using an IfThenElse iterator), ORDER BY, and RETURN clauses. In general, the implementation of second-order functions in XQuery is comparable to the implementation of nested queries in SQL.

The only difference between a FOR iterator and a FORSEQ iterator is the logic that computes the variable bindings. Obviously, the FOR iterator is extremely simple in this respect because it binds its running variable to every item of an input sequence individually. The FORSEQ iterator for tumbling windows is fairly simple, too. It scans its input sequence from the beginning to the end (or infinitely for unbounded inputs), thereby detecting windows as shown in Figure 3.5. Specifically, the effective Boolean value of the conditions of the START or END clauses are computed for every new item in order to implement the state transitions of the automaton of Figure 3.5. These conditions are also implemented by iterator trees. The automata for sliding and landmark windows are more complicated, but the basic mechanism is the same and straightforward to implement.
3.5. Implementation

3.5.3.2 Window Management

The most interesting aspect of the implementation of the `FORSEQ` iterators is memory management and garbage collection. The items of the input sequence are materialized in main memory. Figure 3.8 shows a (potentially infinite) input stream. Items of the input stream that have been read and materialized in main memory are represented as squares; items of the input stream which have not been read yet are represented as ovals. The materialization of items from the input stream is carried out lazily, using the iterator model. Items are processed as they come in, thereby identifying new windows, closing existing windows, and processing the windows (i.e., evaluating the `WHERE` and `RETURN` clauses). This way, infinite streams can be processed. Full materialization is only needed if the query involves blocking operations such as `ORDER BY`, but such queries are illegal on infinite streams (Section 3.4).

According to the semantics of the different types of windows, an item can be marked as active or consumed in the stream buffer. An active item is an item that is involved in at least one open window. Correspondingly, consumed items are items that are not part of any active window. An item is immediately marked as consumed if no window is open and it does not match the `START` condition of the `FORSEQ` clause (Figure 3.5). Otherwise, an item is marked as consumed if all windows that involve that item have been fully processed; this condition can be checked easily by keeping a position pointer that keeps track of the minimum first position of all open windows. In Figure 3.8, consumed items are indicated as white squares; active items are indicated as colored squares. In Figure 3.8, Window 1 is closed whereas Window 2 is still open; as a result only the items of Window 2 are marked as active in the stream buffer. The position pointer is not shown in Figure 3.8; it marks the start of Window 2.

Consumed items can be garbage collected. To implement memory allocation and

![Figure 3.8: Stream Buffer](image)

Consumed, available for GC  |  Active  |  Not Read
garbage collection efficiently in Java, the stream buffer is organized as a linked list of chunks (chunking and chaining are not shown in Figure 3.8 for readability). That is, memory is allocated and de-allocated in chunks which can store several items. If all the items of a chunk are marked consumed, that chunk is released by removing it from the linked list of chunks. Furthermore, all references to closed windows are removed. At this point, there are no more live references that refer to that chunk, and the space is reclaimed by the Java garbage collector.

Windows are represented by a pair of pointers that refer to the first and last item of the window in the stream buffer. Open windows only have a pointer to the first item; the last pointer is set to NULL (i.e., unknown). Obviously, there are no open windows that refer to chunks in which all items have been marked as consumed. As a result of this organization, items need to be materialized only once, even though they can be involved in many windows. Furthermore, other expressions that potentially require materialization can reuse the stream buffer, thereby avoiding copying the data.

3.5.3.3 General FORSEQ

The implementation of the general FORSEQ varies significantly from that of the three kinds of windows. In particular, representing a sub-sequence by its first and last item is not sufficient because the general FORSEQ involves the processing of non-contiguous sub-sequences. In order to enumerate all sub-sequences, our implementation uses the algorithm of Vance/Maier [94], including the bitmap representation to encode sub-sequences. This algorithm produces the sub-sequences in the right order so that no sorting of the windows is needed in the absence of an ORDER BY clause. Furthermore, this algorithm is applicable to infinite input streams. Additional optimizations are needed in order to avoid memory overflow for a general FORSEQ on infinite streams; e.g., the hopeless window detection, described in the next section.

3.5.4 Optimizations

This section lists several simple optimizations that we found useful in our implementation. In particular, these optimizations were important in order to meet the requirements of the Linear Road Benchmark (Section 3.7). Each of these optimizations serves one or a combination of the following three purposes: a.) reducing the memory footprint (e.g., avoid materialization); b.) reducing the CPU utilization (e.g., indexing); c.) improving streaming (e.g., producing results early). Although we are not aware of any streaming
SQL engine which implements all these optimizations, we believe that most optimizations are also applicable for streaming SQL. A condition for most optimizations is that a predicate-based approach to define window boundaries is adopted. So far, no such streaming SQL proposals have been published.

The purpose of this list is to give an impression of the kinds of optimizations that are possible. All these optimizations are applied in addition to the regular XQuery optimizations on standard XQuery expressions (e.g., [37]). For example, rewriting reverse axes can be applied and is just as useful for FORSEQ queries as for any other query.

### 3.5.4.1 Predicate Movearound

The first optimization is applied at compile-time and moves a predicate from the WHERE clause into the START and/or END clauses of a FORSEQ query. This optimization can be illustrated by the following example:

```xquery
forseq $w in $seq landmark window
    start when fn:true()
    end when fn:true()
where $w[1] eq ''S'' and $w[last] eq ''E'' return $w
```

This query can be rewritten into the following equivalent query, which computes significantly fewer windows and can therefore be executed much faster and with lower memory footprint:

```xquery
forseq $w in $seq landmark window
    start curItem $s when $s eq ''S''
    force end curItem $e when $e eq ''E''
return $w
```

### 3.5.4.2 Cheaper Window

In some situations, it is possible to rewrite a landmark window query into a sliding window query or a sliding window query into a tumbling window query. This rewrite is useful because tumbling windows are cheaper to compute than sliding windows, and sliding windows are cheaper than landmark windows. This rewrite is frequently applicable if schema information is available. If it is known (given the schema), for instance, that the input sequence has the following structure “a, b, c, a, b, c, ...”, then the following expression:
forseq $w$ in $\$seq$ sliding window
  start curItem $s$ when $s$ eq 'a'
  end curItem $e$ when $e$ eq 'c'
return $w$

can be rewritten into the following equivalent expression:

forseq $w$ in $\$seq$ tumbling window
  start curItem $s$ when $s$ eq 'a'
  end curItem $e$ when $e$ eq 'c'
return $w$

### 3.5.4.3 Indexing Windows

Using sliding and landmark windows, it is possible that several thousand windows are open at the same time. In the Linear Road Benchmark, for example, this situation is the norm. As a result, the END condition must be checked several thousand times (for each window separately) with every new item (e.g., car position reading). Obviously implementing such a check naively will increase the response time of the queries dramatically. Therefore, it is advisable to use an index on the predicate of the END clause. Again, this indexing is best illustrated with the help of an example:

forseq $w$ in $\$seq$ landmark window
  start curItem $s$ when fn:true()
  end curItem $e$ when $s/@id$ eq $e/@id$
return $w$

In this example, windows consist of all sequences in which the first and last items have the same id (this query pattern is frequent in the Linear Road Benchmark which tracks cars identified by their id on a highway). The indexing idea is straightforward. An "@id" index (e.g., a hash table) is built on all windows. When a new item (e.g., a car position measurement with the id of a car) is processed, then that index is probed in order to find all matching windows that must be closed. In other words, the set of open windows can be indexed just like any other collection.
3.5.4.4 Improved Pipelining

In some situations, it is not necessary to store items in the stream buffer (Figure 3.8). Instead, the items can directly be processed by the WHERE clause, RETURN clause, and/or nested FOR, LET, and FORSEQ clauses. That is, results can be produced even though a window has not been closed. This optimization can always be applied if there is no ORDER BY and no FORCE in the END clause. It is straightforward to implement for tumbling windows. For sliding and landmark windows additional attention is required in order to coordinate the processing of several windows concurrently.

3.5.4.5 Hopeless Windows

Sometimes it is possible to detect at runtime that the END clause or the predicate of the WHERE clause of an open window cannot be fulfilled. We call such windows hopeless windows. Such windows can be closed immediately, thereby saving CPU cost and main memory.

3.5.4.6 Aggressive Garbage Collection

In some cases, only one or a few items of a window are needed in order to process the window (e.g., the first or the last item). Such cases can be detected at compile-time by analyzing the nested expressions of the FLWOR expression (e.g., the predicates of the WHERE clause). In such situations, items in the stream buffer can be marked as consumed even though they are part of an open window, resulting in a more aggressive chunk-based garbage collection.

3.6 MXQuery in SECRET

In order to position MXQuery’s execution model with respect to other Data Stream Management Systems, in this section we describe its behavior expressed using the SECRET model presented in Section 2.4. Moreover, we briefly outline the extensions the model has to undergo in order to accommodate predicate-based windows.

Scope and Content. As previously explained, MXQuery uses a window specification based on predicates. That is, instead of defining size and slide values (like for time-based or tuple-based windows), the window boundaries are marked through predicates
(start and end), which could be expressed on any of the stream items attributes. When these predicates evaluate to true, a new window is created.

A window is open as long as its end predicate is false. The special characteristic of predicate-based windows is that they can overlap in many ways (as previously explained): they can be fully contained in one another, and even more, multiple windows can be closed at the same time. This is a clear difference to time- and tuple-based windows (Section 2.3) for which at most one window is closed at each Tick. In this light, returning a single active window, like the earliest open window, will miss valid Scope intervals.

As a result, the Scope parameter has to return multiple intervals representing the open windows. Moreover, the Scope parameter becomes operational as it cannot be defined in the absence of the stream data (the window boundaries may be defined through predicates on the stream items attributes). The start of the first window is dependent on the input stream’s data and is represented by the first encountered item for which the START predicate matches.

The Content parameter, on the other hand, can be used without any change.

Report. In MXQuery, by default, reporting is executed when a window closes (the condition for window closing in the Report parameter has to match the predicate defining the end boundary of the window) and its content is not empty (the running variable is never bound to an empty sequence). In addition, an optimization option exists (Section 3.5.4.4) which allows early release of results (i.e., before the window closes).

The window closing condition in the Report parameter has to be extended, in order to reflect the end predicate. Moreover, as reporting could be executed for multiple windows at the same time, the Report parameter has to be enhanced with a property specifying the order in which those window contents are made visible. In MXQuery’s case, it is defined by the order of the first items in those windows.

Tick. MXQuery reacts to each new data item arrival on a stream and therefore exposes a tuple-driven execution model, similar to e.g., StreamBase [11].

3.7 The Linear Road Benchmark

The Linear Road Benchmark [23] is a streaming benchmark designed for Data Stream Management Systems. Other proposals exist, like Nexmark [8] which is still work-in-progress, but Linear Road is the most complex and it exercises various aspects of a
DSMS, requiring window-based aggregations, stream correlations and joins, efficient storage and access to intermediate results and querying a large (millions of records) database of historical data. Furthermore, Linear Road poses real-time requirements: all events must be processed within five seconds.

The benchmark describes a traffic management scenario in which the toll for a road system (a set of highways divided into segments) is computed based on the utilization of those roads and the presence of accidents. Both toll and accident information are reported to drivers.

The input for the Linear Road Benchmark is represented by a stream of car position reports (each car reports its position in the road system once every 30 seconds) and query execution requests.

An accident occurs when at least two cars are stopped. A car is considered to be stopped if it reports four consecutive position reports with speed 0. The accidents are only reported to cars which are potentially affected by the accident, i.e., they report positions which are five segments away from the segment where the accident took place, in both directions. Each accident is cleared after a period of time.

The utilization statistics of a highway include the number of cars and their average speed computed over periods of five minutes and are updated every minute. When a car crosses from a segment of a highway to the next, the driver is informed by the toll of the segment s/he is entering and the toll of the previous segment s/he traveled on is charged to the car’s balance account.

Furthermore, the benchmark involves a stream of historic queries on account balances and total expenditures per day. The request on account balance should return the current balance of the driver. The total expenditure query refers to historical information of tolls assessed to cars in the previous ten days and it reports the amount of tolls spent by a driver on a certain day.

As a result, the benchmark specifies four output streams: toll notification, accident notification, account balances, and daily expenditures (the benchmark also specifies a fifth output stream as part of a travel time planning query, but no published implementation includes this query and neither do we).

The benchmark specification contains a data generation program which produces a stream of events composed of car positions and queries. The data format is CSV which is natively supported by MXQuery. Three hours worth of data are generated. An implementation of the benchmark is compliant if it produces the correct results and fulfills the five seconds real-time requirement. The correctness of the results are validated using
a validation tool so that load shedding or other load reduction techniques are not allowed. Fulfilling the real-time requirements becomes more and more challenging over time: with a scale factor of 1.0, the data generator produces 1'200 events per minute at the beginning and 100'000 events per minute at the end.

The benchmark 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 ($L=0.5, 1.0, 1.5, 2.0, ...$). A certain load level is considered to be achieved if all the input events are processed within five seconds from their arrival in the processing system.

### 3.8 Linear Road Benchmark Implementation

To validate our implementation of `FORSEQ` and continuous XQuery processing, we implemented the Linear Road Benchmark using the extended MXQuery engine.

Only few results of compliant implementations have been published so far: Aurora [23], an (unknown) relational database system [23], IBM Stream Processing Core [66] and a more recent implementation on SCSQ [14]. Both the Aurora and IBM Stream Core implementations are low-level, based on a native (C) implementation of operators or processing elements, respectively. The implementation of the benchmark on an RDBMS uses standard SQL and stored procedures, but no details of the implementation have been published. There is also an implementation of the benchmark using CQL [22]; however, no results of running the benchmark with that implementation have been published. To the best of our knowledge, our implementation is the first compliant XQuery implementation of the benchmark.

Our first attempt was to implement the whole benchmark in a single XQuery expression; indeed, this is possible! However, MXQuery was not able to optimize this huge expression in order to achieve acceptable (i.e., compliant) performance. As a consequence, we decided to (manually) partition the implementation into ten continuous XQuery expressions (two expressions are merged into `Toll Computation` and two more into `Accident Detection`) and eight (temporary) stores; i.e., a total of 18 boxes. Figure 3.9 shows the corresponding workflow.

As can be observed in the figure, we tried to avoid the repeated materialization of data as much as possible by sharing intermediate results. This was a very important

---

3In our experiments, we encountered the same bugs as reported in [66] with the validation tool and data generator. Otherwise, all our results validated correctly.
optimization step as Linear Road is a very data-intensive application. In fact, the storage manager’s efficiency in handling all these data was a determining factor for obtaining good performance (this idea is discussed in detail in Chapter 4).

The input stream produced by the Linear Road data generator is fed into three continuous XQuery expressions: Car Position Reports which filters the position reports from the input, Balance Request, which extracts balance request queries and reports the account balance to the drivers and Daily Expenses, which filters the daily expenditure queries, looks up the information in the historical data and informs the driver of the value.

The Car Position Reports query generates a stream of results which is processed by three queries: detect the accidents (Accidents Detection), compute the toll values corresponding to each highway segment (Toll Computation) and detect which cars cross from a segment to the next (Segment Cross). In order to detect whether a car crosses a segment or not, we use a store (CARS in Figure 3.9) which maintains position information for all the cars in the system.

As an example, next, we present the query which detects the accidents given an input stream of position reports from cars with speed 0:
The query uses `FORSEQ` to define a sliding window having a size of two minutes. For each window, the first `FOR` expression groups the reports based on the position information (highway, segment, lane, car id etc.) to extract the cars which reported the same position four times in a row (the condition in the group-by statement) and as such, to detect stopped cars. The second `FOR` expression groups stopped cars which have
the same position. If a group has more than two elements (two cars stopped) then an accident is detected and returned.

The ACCIDENTS data store holds the accidents which occurred on the road system and is used by the Toll Computation query to calculate the toll values and the Accident Notification query to alarm drivers crossing segments of accidents in their way.

The segment toll values are maintained in the TOLLS store which is used by the Toll Notification query to inform the drivers of the toll values for the segments they are entering. Additionally, the Toll Notification query is responsible for assessing (to a driver's current balance) the toll corresponding to the segment the driver exits.

The output results of all the four queries are written in corresponding files: Accident Alerts, Toll Alerts, Balance Reports and Daily Expenses.

Binding an input stream to an XQuery expression is done by external variable declarations as specified in the XQuery recommendation [28]. This approach is in line with the approaches taken in [23, 66], the other published and compliant benchmark implementations. Aurora, however, uses 60 boxes (!).

Nine threads were used in order to run the continuous XQuery expressions and move data into and out of data stores. Except for the HISTORICAL TOLLS which resided in a relational database, all other data stores were all main-memory based (not persistent and not recoverable) using a synchronized version of the stream buffer described in Section 3.5. More details about the stores' implementation will be presented in Chapter 4.

### 3.9 Experimental Results

The implementation of the benchmark was evaluated on a Linux machine with a 2.2 GHz AMD Opteron processor and 4GB of main memory. Our hardware is comparable to the machines used in [23] and [66]. A Sun JVM Version 1.5.0.09 was used, the maximum heap size was set to 2 GB which corresponds to the available RAM used in the experiments reported in [23, 66]. The results can be summarized as follows for the different scale factors L:

- \( L=0.5, 1.0, 1.5, 2.0 \): MXQuery is fully compliant.
- \( L=2.5 \): MXQuery is not compliant. The response time exceeds the five seconds limit.
The best published results on comparable hardware are compliant with an $L$ of 2.5 \[23, 66\]. These implementations are low-level C implementations that do not use a declarative language (such as SQL or XQuery). Even if our implementation did not achieve load 2.5, the differences are surprisingly small (less than a factor of 2) given that our focus was to extend a general-purpose XQuery engine whereas those implementations directly target the Linear Road Benchmark. Also, MXQuery is written in Java which comes with a performance penalty.

The only compliant SQL implementation of the benchmark \[23\] is at an $L$ of 0.5 (contrasting an $L$ of 2.0 of our XQuery implementation). The maximum response times of the SQL implementation at $L$ 1.0 and 1.5 were several orders of magnitude worse than the benchmark allows (2'031 and 16'346 seconds, respectively). Details of that SQL implementation of the benchmark are not given; however, it seems that the overhead of materializing all incoming events in a relational database is prohibitive. As part of the STREAM project, a series of CQL queries were published in order to implement the benchmark. However, no performance numbers were ever published using the CQL implementation. In summary, there does not seem to be an SQL implementation of the benchmark that outperforms our XQuery implementation. Fundamentally, there is no reason why either SQL or XQuery implementations would perform better on this benchmark because essentially the same optimizations are applicable to both languages. Due to the impedance mismatch between streams and relations, however, it might be more difficult to optimize streaming SQL because certain optimizations must be implemented twice (once for operators on streams and once for operators on tables).

### 3.10 Related Work

As mentioned in the introduction, there have been numerous proposals to extend SQL; the most prominent examples are AQuery \[74\], CQL \[22\], and StreaQuel \[36\]. StreamSQL \[12\] is an activity (started in November 2006) that tries to standardize streaming extensions for SQL. As part of all that work, different kinds of windows were proposed. In \[97\], the authors propose a special construct named MATCH RECOGNIZE for finding patterns in sequences of rows.

In our design, we were careful that all queries that can be expressed in these SQL extensions can also be expressed in a straightforward way using the proposed XQuery extensions. In addition, if desired, special kinds of streams such as the $i$-streams and $d$-streams devised in \[22\] can be implemented using the the proposed XQuery exten-
3.10. Related Work

sions. Furthermore, we adopted several important concepts of those SQL extensions such as the window types. Nevertheless, the work on extending SQL to support windows is not directly applicable to XQuery because XQuery has a different data model and supports different usage scenarios. Our use cases, for instance, involved certain patterns, e.g., the definition of window boundaries using general constraints (e.g., on authors of RSS postings) that cannot be expressed in any of the existing SQL extensions. All SQL extensions published so far are only able to specify windows based on size or time constraints; those SQL extensions are thus not expressive enough to handle these use cases, even if the data is relational. Apparently, StreamSQL will adopt the predicate-based approach, but nothing has been published so far (the StreamSQL documentation in [12] still uses size and time constraints only).

There have also been proposals for new query languages in order to process specific kinds of queries on data streams. One example is SASE [96] which was proposed to detect patterns in RFID streams; these patterns can be expressed using regular expressions. Another proposal is Wavescope [79], a (Turing-complete) functional programming language in order to process signals in a highly scalable way. While such languages and systems are useful for particular applications, this work provides general-purpose extensions to an existing main-stream programming language. Again, we made sure in our design that all the SASE and Wavescope use cases can be expressed using the XQuery extensions; however, our implementation does not scale as well for those particular use cases as the SASE and Wavescope implementations.

There have been several prototype implementations of stream data management systems; e.g., Aurora [18], Borealis [17], Cayuga [42], STREAM [22], and Telegraph [36]. All that work is orthogonal to the main contribution of this work. In fact, our implementation of the Linear Road Benchmark makes extensive use of the techniques proposed in those projects.

The closest related work is the work on positional grouping in XQuery described in [69]. This work proposes extensions to XQuery in order to layout XML documents, one of the usage scenarios that also drove our design. The work in [69] was inspired by functionality provided by XSLT in order to carry out certain XML transformations. However, many of our use cases on data streams cannot be expressed using the proposed extensions in [69]; our proposal is strictly more expressive. Furthermore, the work of [69] does not discuss any implementation issues. Another piece of related XML work discusses the semantics of infinite XML (and other) streams [80]. That work is orthogonal to our work.
3.11 Summary

This chapter presents two extensions for XQuery: windows and continuous queries. Due to their importance, similar extensions have been proposed for SQL, and several ideas of those SQL extensions (in particular, the types of windows) have been adopted in our design. The proposed extensions are implemented in MXQuery, an open source XQuery engine.

The Linear Road Benchmark's implementation in MXQuery is described in detail. This benchmark implementation will also be extensively used in the experiments presented in Chapters 4 and 5.

The experimental results seem to indicate that XQuery stream processing can be implemented as efficiently as SQL stream processing and that there is no performance penalty for using XQuery. Nevertheless, the Linear Road Benchmark implementation pointed out some possible areas of performance improvement, not specific to XQuery, but rather related to the efficiency of intermediate state and window management for such data intensive applications like the above-mentioned benchmark. We will return to this observation in Chapter 4.
Chapter 4

Storage Manager for Streams

Our experience with data-intensive streaming applications implementation in MXQuery (Chapter 3) showed that, very often, Data Stream Management Systems operate under strict performance requirements. Key to meeting such requirements is to efficiently handle data storage management time-critical tasks, like: managing internal states of continuous query operators or the traffic on the queues between operators, as well as providing storage support for shared computation and archived data.

In this chapter, we introduce a general purpose storage management framework for DSMSs that performs these tasks based on a clean, loosely-coupled, and flexible system design that also facilitates performance optimization. An important contribution of the framework is that, in analogy to buffer management techniques in relational database systems, it uses information about the access patterns of streaming applications to tune and customize the performance of the storage manager.

We first analyze typical application requirements at different granularities in order to identify important tunable parameters and their corresponding values. Based on these parameters, we define a general-purpose storage management interface. With the help of this interface, a developer can use our SMS (Storage Manager for Streams) to generate a customized storage manager for streaming applications. We explore the performance and potential of SMS through a set of experiments using the Linear Road Benchmark.
4.1 Introduction

Modern data stream processing applications are complex and impose diverse requirements: they involve multiple continuous queries that run in parallel, join live and stored data sources, and are highly data-intensive. Implementing these applications often requires temporary materialization of large windows as well as maintenance of large and highly dynamic operator state. These applications (e.g., highway traffic monitoring [23], Internet traffic analysis [41], log monitoring/mining [53], scientific data processing [73]) also operate under very strict performance requirements.

In this context, efficient data management in a DSMS is crucial for meeting the requirements of these applications. Although many DSMS solutions have been offered to date (e.g., [18, 20, 36]), none of them provides a clean and systematic approach to storage management. In these systems, the storage manager is tightly coupled with the query processing engine, like MXQuery’s initial architecture presented in Section 3.5.1. Such a design not only makes the storage manager adhoc and inflexible, but it also restricts the possibility for further optimizations. In practice, being able to customize and tailor the storage manager to the requirements of the application is key to achieving good performance.

Interestingly, using a tunable storage manager separated from the query engine has been a fundamental design principle of traditional DBMS architectures and has been the basis for many useful performance optimizations (e.g., [39]). A first contribution of this work is to argue that a similar design is needed for DSMSs. Hence, in this chapter we propose such a separate storage manager and prove its advantages by showing how to make a storage manager tunable. By analyzing the patterns observable on data streams and the queries over these data streams, we have identified a set of important parameters that can be used to tune the performance of the storage manager. Based on this analysis, we have developed an advanced interface that can be used to generate highly efficient, customized storage managers.

Our approach of a tunable, customizable storage manager clearly borrows ideas successfully exploited for buffer management in traditional databases [39]. In this area, we make additional contributions as the problem is quite different in the case of data streams. As in relational databases, the read patterns of streaming queries can be predicted in advance and can be used to select data structures, access paths, and indexes. This approach makes even more sense in a streaming system, where continuous queries are known a priori and their more accurate static analysis is possible. Unlike relational databases though (where data is relatively more static and updates
are less frequent), in a streaming system, the update patterns also play a key role, as they affect decisions on data layout to better support highly dynamic data movement. Considering update patterns in the optimization strategies is nontrivial and this chapter discusses in great detail which parameters are relevant and how to use them to tune the storage manager.

The final two contributions of the work are the system itself and its performance evaluation. The system we describe is called SMS (Storage Manager for Streams). SMS is a general-purpose storage manager for DSMSs that uses the storage parameters defined through our analysis and delivers a method for tuning them for performance. SMS is built on a well-defined and powerful interface so that it can easily and effectively be tailored to different application needs and can potentially serve as the underlying storage manager for any DSMS. To prove the advantages of SMS and the feasibility of the ideas explored in this work, we used SMS as the underlying storage manager for MXQuery and ran the Linear Road implementation presented in Chapter 3. Our experiments show that SMS can achieve a 2-6 factor of improvement over a “one-size-fits-all” baseline store implementation (i.e., not tuned well to application needs), measured using three different performance metrics.

This chapter is structured as follows: in Section 4.2, some motivating application scenarios are described together with their storage requirements. In Section 4.3, we present our fine-grained analysis to identify the key storage requirements and their respective parameters. Section 4.4 focuses on the access pattern parameters and discusses a framework for the possible values these parameters can take. Section 4.5 presents our SMS architecture. In Section 4.6 we describe the implementation of the SMS store instances for different storage parameters, based on state-of-the-art techniques. Section 4.7 proves, through a benchmark study, that SMS can meet the storage requirements very well, and can significantly improve the query processing performance. Section 4.8 presents an overview of the related work. Finally, we conclude the chapter with a summary in Section 4.9.

### 4.2 Application Scenarios

We are targeting data-intensive streaming applications, including but not restricted to applications that need to manage a large state during stream processing. Examples of such applications are:
Chapter 4. Storage Manager for Streams

Internet Traffic Analysis: Correlation of packet streams over wide-area networks requires keeping a significant amount of data available for joining over time/location/protocol [41].

Log Monitoring/Mining: Discovering new patterns or checking the presence of predefined patterns in data log streams involve managing large processing state [53].

Scientific Data Processing: Very high data rates of scientific experiments need to be processed under fairly tight time constraints in order not to overload the storage and network capabilities [73].

One challenging application scenario that we studied in detail in the previous chapter (Sections 3.7 and 3.8) is the Linear Road Benchmark, which was developed to evaluate the performance of a DSMS.

In short, Linear Road simulates the traffic on a set of highways with segments, and provides variable tolling depending on traffic statistics and accident occurrences. The input stream consists of car position reports and queries. The system has to react to different patterns in the reports and answer the queries accordingly.

Linear Road’s design goal was to stress all possible components of a DSMS, not just some operator or scheduling implementation. In particular, it provides a fairly comprehensive set of requirements to stream storage, some of which we briefly outline here:

- For computing the toll for a specific segment on a highway, the traffic statistics of that segment are used. Statistics are obtained from analyzing a window with traffic information (number of cars, speed etc.) created for the past five minutes with a slide of one minute. Therefore, the traffic statistics information expire in the order that they were generated.

- Once an accident is detected, every vehicle that enters into a segment in the vicinity of that accident must be notified. This requires storing the accidents until they are cleared. Cleared accidents are determined by new values in the streaming data (i.e., cars that were previously stopped, started to move).

- Tolls assessed to a specific car need to be stored, for the drivers to be able to later request their current balance of toll spendings. One solution for storing this information is to create (key, value) pairs, while using the new tolls to update the balance that has the appropriate key (the car id).

- The car position reports are used by multiple queries: accident detection, segment crossing detection and toll computation. These three queries maintain windows of different sizes on the same data, which for performance reasons can be shared.
• For the daily expenditure query, ten weeks worth of data need to be stored (the tolls spent by all drivers in this time interval). Given the static nature of this large data, a solution is to use a relational database for storing them.

• As the information about the accidents in the past minute may not be available when a certain car crosses a segment, a synchronization condition on the store contents is required to keep the request waiting until the most recent accident data is available.

As discussed above, Linear Road presents a clear evidence that data-intensive stream processing applications may come with a wide variety of storage requirements. Providing a flexible and configurable storage management solution is key to meeting these requirements in the most effective and scalable way.

In the next sections, we will describe the three major building blocks of a decoupled storage manager for streams: (i) the language for specifying storage requirements (represented with a set of parameters) in Sections 4.3 and 4.4, (ii) the interface to communicate them in Section 4.5, and (iii) their internal implementations in Section 4.6.

4.3 Modeling Storage Management for Stream Processing

We first conduct a requirement analysis in order to identify the set of tunable parameters that a Stream Processing Engine (SPE\(^1\)) can use to communicate its needs to a storage manager.

These parameters have threefold importance: (i) they provide the required data access functionality (i.e., basic per-item operations), (ii) they support the SPE in query processing and optimization (e.g., predicate push-down, push/pull models), and (iii) they offer support for reducing the storage management costs (e.g., lowering response times and memory consumption).

\(^1\)We use the term SPE to refer to the query processing part of a DSMS.
4.3.1 Architectural Parameters

A DSMS can be architected in two ways in terms of its processing model: push-based and pull-based. In the former way, input stream items are pushed through the query network (e.g., Aurora [18]), whereas the latter follows the traditional database systems approach in which stream items are pulled from the source input stream, processed and finally stored in a result container to be further pulled by other operators (e.g., MXQuery, Section 3.5). A general-purpose storage system has to be able to support and combine both processing models under the same framework. Therefore, the first part of our analysis refers to how the data items get into the storage structures as well as from there back into the SPE, behavior captured by two parameters:

**Role**: This property shows how the data gets materialized from a stream into a store (Figure 4.1). A store can be either active or passive. An active store is directly connected to the source stream and pulls the items from it. In the case of a passive store, on the other hand, the SPE receives the items from the input stream and pushes them to be materialized in the store.

**Access Model**: This property shows how the data gets from a store to the SPE. There are two ways: the pull-model and the push-model. Pull-based stores wait for requests from operators, while push-based stores notify the engine about newly available data.

4.3.2 Functional Parameters

In addition to architectural parameters, there are also certain properties that arise from a special functional requirement of an SPE.
4.3. Modeling Storage Management for Stream Processing

**Synchronization and Schema:** SPEs usually have a scheduler whose job is to run operators in a certain order. One task that they cannot achieve is to delay the access of an operator on materialized input until a condition is met (e.g., until the contained values satisfy a predicate).

An example extracted from the Linear Road Benchmark is accident notification. A driver entering a new segment of a certain highway should be notified by accidents in the vicinity, an information which should be up to date (from the last minute); if not, the request for accident details has to be delayed until the condition is satisfied.

For implementing this requirement in Aurora, the authors had to add a new operator box, named WaitFor [25]. A materialized store with some synchronization capability would have made the creation of such an operator box unnecessary. That is, the SPE may specify a condition on the store which needs to be met before the consumer operator is allowed to retrieve data. Usually this condition is on some attribute values, which means that it is important to know the schema of the streaming items (if available). In the given example (waiting for the most recent information on accidents), the condition will eventually be met, because of the monotonicity of time. Of course, thresholds on the waiting period have to be defined for the conditions which are not guaranteed to be finally satisfied.

From this example we extract two parameters: *Synchronization* whose value is represented by the synchronization condition and *Schema* representing the stored data’s schema knowledge: if all the stored items comply to the same schema or not, the number of attributes, the attributes names and the types. The schema information is also important for predicate push-down optimizations. Besides functionality, as it will be shown later in the chapter, knowing that all items comply to the same schema helps the storage manager in making some implementation decisions.

One issue that needs to be pointed out at this stage is that this synchronization mechanism solves a particular aspect of a more general problem: how to order the operations on a store in order to make sure that the query results satisfy the correctness imposed by the streaming application. We will come back to this observation in Chapter 5 where we will formalize the problem and present a general solution.

**Persistence:** Queries involving archived data sometimes need items which are already stored in a relational database. Additionally, an SPE, based on the characteristics of the data (volume and/or how often it is updated, etc), could also specify that it should be stored on disk rather than in main memory. To account for these scenarios, we introduced a parameter named *Persistence*, with two possible values: *Persistent* and *Transient*. 
<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
<th>Possible Values</th>
<th>Default Values</th>
</tr>
</thead>
<tbody>
<tr>
<td>Role</td>
<td>How the data is materialized in a store</td>
<td>Active, Passive</td>
<td>Passive</td>
</tr>
<tr>
<td>Access Model</td>
<td>How the data gets from a store to the SPE</td>
<td>Push, Pull</td>
<td>Pull</td>
</tr>
<tr>
<td>Synchronization</td>
<td>If the store acts as a synchronization point</td>
<td>False, Synchronization condition</td>
<td>False (no wait)</td>
</tr>
<tr>
<td>Schema</td>
<td>Schema knowledge: number, names, types of attributes</td>
<td>No schema, Schema information</td>
<td>No schema</td>
</tr>
<tr>
<td>Persistence</td>
<td>Where the data is materialized (disk or memory)</td>
<td>Persistent, Transient</td>
<td>Transient</td>
</tr>
<tr>
<td>Read Pattern</td>
<td>Requests for reads from a store</td>
<td>Sequential, Random, Clustered</td>
<td>Random</td>
</tr>
<tr>
<td>Update Pattern</td>
<td>Requests for updates to a store</td>
<td>FIFO, RANDOM, IN-PLACE</td>
<td>RANDOM</td>
</tr>
<tr>
<td>Sharing</td>
<td>If the store is shared by multiple operators executing reads</td>
<td>Shared, Not shared</td>
<td>Not Shared</td>
</tr>
</tbody>
</table>

**Table 4.1: SMS Tunable Parameters**

### 4.3.3 Performance-related Parameters

Apart from providing the necessary architectural support and functionality, the storage manager should have high performance. As stated before, we target data-intensive applications which have low response time and low memory consumption requirements. As we will show later in the chapter, certain pieces of information help the storage manager to make decisions about how to help the SPE in meeting these requirements. The access patterns of the operators and sharing of materialized data are two of the most important parameters because, given their values, the storage manager will implement specific mechanisms for speeding up store access operations.

**Access Patterns (Read and Update):** An operator’s storage requirements can be characterized by its *data access patterns*. By observing the read and update behavior of an operator on its internal state (or on intermediate results), we can depict some general patterns, which can be then used in implementations for performance. We will present a more detailed, fine-grained analysis of the read and update patterns in
Section 4.4

Sharing: Continuous queries show high similarities and this observation has been already used by the DSMS developers for multi-query optimization. The performance of running multiple continuous queries in parallel could be improved by sharing computation, and therefore, intermediate results and operator states [22]. As an example, we consider computing the accidents in Linear Road. As outlined in Section 4.2, there are two queries which share the accidents information: one which notifies other cars about the accidents, and another which computes the tolls assigned to a segment based on accident vicinity. In order to avoid recomputing and duplicating the accident information, this data is materialized in a single, shared store. For this purpose, the Sharing parameter can be used to support an SPE in performing its sharing-based query optimizations.

Table 4.1 provides a summary of the SMS parameters that we identified in this section. In the next section, we will further elaborate on access patterns and sharing, and show how they can be set to capture performance-related behaviors. We only concentrate on the performance-related parameters as they are tunable and, as their name implies, have a high impact on performance. The architectural and functional parameters have generally binary values and the decisions for their implementations are straightforward.

### 4.4 Access Patterns

In a DSMS, there is a continuous flow of data items that enter the system, get processed through the queries to further produce results that leave the system. During this process, the data items may get materialized (either permanently or temporarily) in different types of stores. Additionally, in case of temporary materialization, after a while, the data items become outdated (i.e., no longer needed by the queries) and therefore must be deleted from the stores to free up the space. An SPE needs to access each of these stores in different ways during query processing, determined by the access patterns of the query operators. In this section, we present the access pattern parameters together with the possible values that they can take.

We consider streams as possibly infinite sets of data items (closely following the definition in Section 2.3). Each data item contains a tuple, composed of attributes, which are the same for all items in a stream if they comply to the same schema. When materialized into a store, each data item is assigned a unique id, which is further used to identify the item in the store.
Data items in a store can be accessed in two alternative ways: value-based and id-based. In the value-based access, given a set of attribute values, the matching items are returned; in the id-based access, the item with the corresponding id is returned.

Furthermore, there are two major categories of access patterns: read patterns and update patterns. Read patterns refer to the way data items get retrieved by the operators, while update patterns refer to how operator states and inter-operator stores get modified as a result of new data arrival, query processing, as well as outdated data eviction.

### 4.4.1 Update Patterns

Without losing generality, new data stream items arrival can be represented through append operations. In this context, the inserts follow a clear pattern. Therefore, in this section, we will only focus on the update patterns which occur in the form of deletes and value replacements.

A store gets updated with deletes or value replacements in two occasions: (i) as a result of data expiration, and (ii) as a result of data consumption.

Our analysis of data expiration is similar to that of Golab and Özsu [57]. The fundamental difference is that we are using the results of our analysis to determine how to configure the stores, while the related work uses them for operator implementation as part of an SPE.

To illustrate, consider two operators (Producer and Consumer) connected as shown in Figure 4.2. The Store connecting them temporarily keeps the intermediate query results that are generated by the Producer to be later processed by the Consumer. The Producer, in addition to the regular append-only insertion of new data items, has an update pattern on the Store denoted by UPD1, while the Consumer has an update pattern on the Store denoted by UPD2 and a read pattern denoted by READ. UPD1 is a result of data expiration, while UPD2 is a result of data consumption.

![Figure 4.2: Store Operations](image-url)
Next, we present the possible update operations that the Producer (UPD1) and the Consumer (UPD2) operators may impose on the store and their semantics (for each of the patterns we provide a simple query example written in an SQL-like language in Table 4.2). The combinations of these patterns will become the update patterns of the Store itself.

### 4.4.1.1 Producer Operator Update Patterns

Data items in a store may expire in four different ways:

**Never Expire.** The Producer generates a stream of results which are materialized in Store, but never expire. This means that there is no UPD1. For example, if it performs a selection operation (query a. of Table 4.2), the selected items are inserted in the Store as regular append-only, and they never expire. In theory, the resulting data items could be kept in the Store forever. In practice, the Consumer operator is to determine when some items are no longer needed and can be deleted (will be described in the next subsection).

**Ordered Expire.** In this case, the data items in the Store become outdated, and therefore deleted, in the order that they were created. For example, if the Producer operator selects some attribute values from a moving window of 24 hours, the query results materialized in the store will expire as the window moves forward (query b. of Table 4.2).

**Unordered Expire.** In some cases, the update pattern may depend on the values of the newly arriving input rather than its arrival order. In other words, the outdated items are determined by the values of some data items' attributes. Since insertions are always append-only, the items in the store will not necessarily be physically clustered by their key values. Therefore, value-based delete requests will usually be translated into randomly ordered deletes (i.e., two successive delete requests will not follow a specific order). An example is provided by the statement c. in Table 4.2, which deletes from the Store the items having the same values for attribute “Attr1” as the items in the input stream.

**Replaced Expire.** Some store items expire when a new item replaces their attribute values (e.g., if the Producer executes the query d. in Table 4.2, it will update the value of attribute “Attr2” in Store with new data from the input stream).

Note that Unordered Expire and Replaced Expire patterns require value-based access to the data items in the Store.
4.4.1.2 Consumer Operator Update Patterns

The updates are generated not only as data items expire, but also as they get consumed. Some of the consumer operators follow the usual stream-based semantics, i.e., they “look” at the input only once. In this case, some data items may redundantly remain in the system, resulting in waste of memory space. We solve this problem by having the consumer operators mark the processed items as “consumed”. Unlike in data expiration, these items need not be eagerly deleted; but instead, they could be removed from the stores lazily. Lazy removal is achieved by periodically running a “garbage collection” process to remove the marked items from the store.

Data items in a store, which have already been processed by all of its consumer operators, can be marked for eviction in three different ways:

**Never Consume.** In some cases, the Consumer operator may not be able to decide if any of its consumed items will be needed in the future. If this is the case, then no item is marked as consumed. An example would be when the Consumer operator executes a repeated scan of the store (query e. of Table 4.2).

```
SELECT Attr1, Attr2
FROM Stream
ON Stream S
DELETE FROM Store W
WHERE W.Attr1 = S.Attr1
a. Never Expire

SELECT Attr1, Attr2
FROM Stream
WINDOW 24 HOURS
b. Ordered Expire

ON Stream S
UPDATE Store W
SET W.Attr2 = W.Attr2 + S.Attr2
WHERE W.Attr1 = S.Attr1
c. Unordered Expire

ON Stream S
UPDATE Store W
SET W.Attr2 = W.Attr2 + S.Attr2
WHERE W.Attr1 = S.Attr1

SELECT FIRST(S.Attr1), SELECT Attr1, Attr2
LAST(S.Attr1), FROM Store S
WINDOW 24 HOURS
f. Ordered Consume

SELECT Attr1, AVG(Attr2)
FROM Store
WINDOW 24 HOURS

SELECT S.Attr1
FROM Store S WINDOW 24 HOURS
WHERE PREV(S.Attr1) != S.Attr1
AND S.Attr1 < NEXT(S.Attr1)
j. Clustered Read
```

Table 4.2: Query Examples for Access Patterns
4.4. Access Patterns

**Ordered Consume.** When a window operator changes state (i.e., a new window state is created), the outdated items (the ones which were previously part of the window but fell outside of it after the state change) are marked as consumed (if for example the Consumer operator executes the query f. in Table 4.2). This is done in a FIFO manner, since older items are consumed earlier than the newer ones.

**Eager Consume.** In some cases, a windowed Consumer operator can take an eager approach in marking the consumed items in the window. For example, when the window (materialized in the store) grows to large sizes, the operator may determine that some of the items inside the window will no longer be needed and therefore, can be immediately marked as consumed. This approach not only reduces memory consumption, but also decreases query processing time, since window lookups for smaller windows take less time. An example is provided by query g. in Table 4.2 in which the query only requires the first and the last items in a window (all items in between can be marked for removal).

### 4.4.1.3 Store Update Patterns

The update patterns of the Producer and Consumer operators can be combined to assign a single update pattern to a given Store. The purpose is to define an update pattern which can be used for optimizing the implementation of the store.

For our example in Figure 4.2 the patterns for UPD1 and UPD2 would determine the update pattern of Store. In this section, we consider all combinations of update patterns due to expiry and due to consumption, and come up with a number of possible update patterns for stores. Table 4.3 presents these combinations.

In general, the update pattern of the Store is determined by the update pattern of the Producer operator. The reason is that expired data items must be immediately removed from the Store to make sure that the Consumer operator has accurate inputs. On the other hand, if the Producer has the Never Expire pattern, then the update pattern of the Store is determined by the update pattern of the Consumer operator.

<table>
<thead>
<tr>
<th></th>
<th>Never Consume</th>
<th>Ordered Consume</th>
<th>Eager Consume</th>
</tr>
</thead>
<tbody>
<tr>
<td>Never Expire</td>
<td>NO-UPDATE</td>
<td>FIFO</td>
<td>FIFO/RANDOM</td>
</tr>
<tr>
<td>Ordered Expire</td>
<td>FIFO</td>
<td>FIFO</td>
<td>FIFO/RANDOM</td>
</tr>
<tr>
<td>Unordered Expire</td>
<td>RANDOM</td>
<td>RANDOM</td>
<td>RANDOM</td>
</tr>
<tr>
<td>Replaced Expire</td>
<td>IN-PLACE</td>
<td>IN-PLACE</td>
<td>IN-PLACE</td>
</tr>
</tbody>
</table>

Table 4.3: Store Update Patterns
Except for Replaced Expire, all other Producer updates are in the form of delete operations (plus, of course, the appends). Therefore, for the value replacement case, we need a special store update pattern, which we call IN-PLACE. For the other pattern combinations, we always choose as the store pattern, the operator update pattern with the most constraints. For example, a combination of Unordered Expire producer update pattern and Ordered Consume consumer update pattern will result in a RANDOM store update pattern, since the former imposes the support for deletes in random order.

From the producer-consumer update pattern combinations, we identify four types of store update patterns:

- **NO-UPDATE.** This store update pattern has no fixed behavior imposed by the Producer-Consumer operators. This is a special case for which a couple of different approaches can be taken, like: to archive materialized data (on disk) or to consider a FIFO pattern with some lifetime specification.

- **IN-PLACE.** This store update pattern is for stores that allow value replacements by key. If the key of an updated item does not already exist in the store, then that item is new and it is therefore inserted.

- **RANDOM.** This store update pattern is called RANDOM to account for the unordered execution of the delete operations.

- **FIFO.** The most common combination of operator update patterns observed in streaming applications is represented by a mix of inserts at the end and deletions from the beginning. For this specific behavior, we define FIFO, a store update pattern which favors queue-like operations. In this type of stores, either items never expire, or they expire in an ordered fashion. For Eager Consume, one of two approaches can be taken: FIFO, when memory consumption is low and we can ignore items being marked as outdated, or RANDOM, to support eager deletion of items.

The implementation of the store update patterns listed above will be described in Section 4.6.

### 4.4.2 Read Patterns

As in related work [39], we do an analysis of the read patterns, in our case, in a streaming environment.
4.4. Access Patterns

As shown in Figure 4.2, the way the Consumer operator requests items from the store determines the read pattern. Although windows are usually stored in main memory, executing a repeated scan of the complete window to find the requested items is not an efficient solution (especially if these windows cover large portions of the streams). Depending on the Consumer operator’s read access patterns, certain optimizations can be applied. In this subsection, we identify these patterns and in Section 4.6 we discuss their optimized implementation.

Streaming operators read data items from a store in three different ways:

**Sequential.** Some Consumer operators access the data items in a store sequentially. Selection inside a window is an example for sequential read. In this case, the items in the specified window will be accessed one after another (e.g., the query h. in Table 4.2).

**Random.** Some operators access the data items in a store in a random order. For example, an Consumer operator executing a windowed group-by aggregate operation will access the items of a window in random order of the groups (query i. in Table 4.2).

**Clustered.** In a clustered read access, there is locality of requests grouped around some items. For example, the query j. from Table 4.2 executes a series of read operations for previous, current and next item given the current position of a sequential cursor.

### 4.4.3 Sharing and Access Patterns

In the previous subsections, we made an enumeration of different read and update patterns. We were considering the simple case in which there was only one consumer operator. For optimization purposes though, stores can be shared among multiple consumers which require access to the same input data (Figure 4.3).

![Figure 4.3: Shared Store](image-url)
For determining the update pattern of a shared store, we propose a simple and safe (i.e., that meets all the constraints and produces correct answers) algorithm: we first determine the combined update pattern of the consumer operators, and then use Table 4.3 to obtain the final update pattern of that store.

The three update patterns on the consumer side (i.e., Never Consume, Ordered Consume, or Eager Consume) yield the following possible combinations:

- If at least one of the consumer operators has a Never Consume update pattern, then the combined update pattern should be Never Consume.

- If all consumers follow an Eager Consumer update pattern, then the combined update pattern may also be Eager Consume for performance reasons (less memory consumption). On the other hand, a less constrained pattern (e.g., FIFO) could be used when there is enough memory.

- All other cases generate Ordered Consume update pattern.

Sharing has an important impact on performance. One reason is that different operators require different data subsets of the shared input, and therefore, there may be many items in the store that are not needed by a certain operator but are still required by another. In this case, knowing how each operator reads the data is important for speeding up the search. Therefore, the read pattern of a store is determined by the read patterns of each of its consumer operators. In other words, the store must provide the storage structures (e.g., indexes) to support the read patterns of all of its consumers.

4.5 SMS Architecture

As outlined in the introduction, our work does not stop at identifying the parameters needed to tune storage in a DSMS, but it takes the next logical step to establish an architecture for decoupling the storage concerns from the processing concerns in order to allow for better optimization, generality and extensibility.

Figure 4.4 shows how a DSMS can be split in two components: the stream processing engine (SPE) and the stream storage, which we call SMS. The SPE is mainly in charge of query processing and optimizations, and uses SMS to perform all of its storage-related tasks (i.e., storing and retrieving data items as needed by the queries). As such, SPE and SMS together act as a complete DSMS.
Given an application, it is the SPE’s responsibility to analyze its requirements and then implement it (usually through a query network). During this process, the SPE uses our analysis of the parameters to determine the access patterns of the operators and to express its own architectural and functional requirements using the parameter values provided by SMS. The resulting combinations of values are then communicated through the dedicated interface to SMS, which, in turn will create store instances that meet the SPEs requirements. It is important to note that SMS takes a static approach when deciding how to use the information provided by the SPE in order to provide storage implementation. This is because we assume that queries are known a-priori. If this assumption is relaxed, it results that SMS would require a mechanism for automatically adjusting the implementation of the stores given possibly new read and update patterns. One solution would be to stop the execution, make all the necessary reconfigurations and then restart query processing with the new implementation. More relaxed approaches may be analyzed: e.g., smoothly adjust stores’ implementation (by e.g., maintaining two implementations for the same store) to the requirements of the new queries. These remain interesting questions and represent an avenue for future work.

In this section, we will present how SPE and SMS interact and how the detailed architecture of SMS looks like.

SMS consists of four main components:
Store Instance

A Store Instance represents a data source. It is characterized by the way its data is inserted, organized and updated. A store instance is configured to provide the required functionality through the set of parameters specified at its creation time (some examples are shown in parenthesis in Figure 4.4). Furthermore, the implementation of a store instance ensures that both the response time and the memory footprint are low. We would like to note that a store instance could either reside in memory or on disk, which is a property that is specified through the Persistence parameter at creation time.

Lifecycle Interface

The Lifecycle Interface is used by the SPE to create and gain access to store instances. Through this interface, the SPE communicates to SMS the properties of the store instances that it would like to create through a set of parameters such as the access patterns. These parameters have been identified as a result of the detailed analysis conducted in the previous sections.

This interface exports a set of operations that are used for the management of store instances:

- `createStore(Properties)` returns a Store Instance that represents the implementation of the required properties. The new store instance is added to the list of currently available stores.
- `removeStore(StoreInstance)` Deletes a store instance as requested by the SPE.
- `registerIndex(IndexSchema, StoreInstance)` This method is used for specifying an index schema, that is used for creating indexes, to allow value-based replacements and reads (the IndexSchema will be described later in the chapter).

Store Factory

As its name implies, the Store Factory is the component of the architecture that is in charge of creating, managing, and deleting store instances. The decisions regarding the implementation of the store instances are made by the Store Factory, which receives the requested parameter values through the lifecycle interface and uses rules for creating different store instances given the combinations of the parameters' values. (e.g., if a FIFO update pattern is specified in the properties, then the Store Factory
4.5. SMS Architecture

<table>
<thead>
<tr>
<th></th>
<th>FIFO</th>
<th>RANDOM</th>
<th>IN-PLACE</th>
</tr>
</thead>
<tbody>
<tr>
<td>Exported methods</td>
<td>deleteUpTo</td>
<td>deleteItems</td>
<td>update</td>
</tr>
<tr>
<td>Parameters</td>
<td>ItemId</td>
<td>ItemIdsList</td>
<td>IndexSchema,NewItem,Values</td>
</tr>
</tbody>
</table>

**Table 4.4: Update Interfaces**

applies the respective implementation for this type of update pattern).

**Access Interface**

The Access Interface enables the SPE to access a specific store instance. It provides basic per-item (and per-attribute) operations such as read and update. The supported operations depend on the actual parameter settings done at the instance creation time.

In the following we will present the interfaces exported when giving values to the store parameters. The **Sharing** and the **Synchronization** condition do not export any interface, as they are used for organizing the internals of the implementation.

**Insertion.**

```java
Interface PassiveRole { bufferItem(Item) }
```

The **bufferItem** method is exported if the **Role** parameter is set to **Passive** value. Otherwise, this method is used internally by the store instance.

**Update.** In Table 4.4 we present the interfaces for each of the three store update patterns (NO-UPDATE obviously does not export any interface explicitly).

In our implementation, the consumption interface only exports methods which allow the identification through item ids and are identical to the ones exported by the Update Interface, FIFO and RANDOM update patterns (columns 2 and 3 of Table 4.4).

The **update** method is exported if the IN-PLACE value for the update pattern is set. **Values** are applied on the index determined by the **IndexSchema** to search for the item to be replaced with **NewItem**.

**Read.** In Table 4.5 the read interfaces are presented.

The **retrieve** method is exported if the requirements specify that further requests will use an index. **Values** are applied on the index determined by the **IndexSchema** to search for the items returned as a result. The **ResultSet** can be further iterated to extract individual items.
Chapter 4. Storage Manager for Streams

<table>
<thead>
<tr>
<th>IndexRead</th>
<th>RandomRead</th>
<th>SequentialRead</th>
</tr>
</thead>
<tbody>
<tr>
<td>Exported methods</td>
<td>retrieve</td>
<td>getFirstItem</td>
</tr>
<tr>
<td></td>
<td></td>
<td>getAttribute</td>
</tr>
<tr>
<td>Parameters</td>
<td>IndexSchema, Values</td>
<td>ItemId AttributeId</td>
</tr>
<tr>
<td>Return values</td>
<td>ResultSet</td>
<td>Item Attribute</td>
</tr>
</tbody>
</table>

Table 4.5: Read Interfaces

Our SMS architecture provides a clean separation between the processing and the storage aspects of a DSMS. The ability to instantiate multiple, differently tuned store instances allows the fine-grained adaptation to the storage needs of an application, while enabling global storage optimizations via the store factory.

4.5.1 Automatic Tuning

In the current architecture, SMS makes decisions in a static manner: based on the queries (we assume known a-priori) it has to execute, an SPE communicates its requirements. These One of the assumptions made early in this dissertation is that (stream) queries are known in advance.

4.6 Store Instance Creation and Implementation

So far the “language” that the SPE can use for communicating with SMS was presented along with the interface it can use to do so. The next logical step is to follow the decision process that SMS executes when interpreting the received information. Given the parameter values, the storage manager configures a store instance that meets the requirements, while trying to provide an optimized implementation.

In the following, the decisions and actions that SMS takes when faced with each of the parameter values are presented.

4.6.1 Architectural Parameters

Role and Access Model: A passive value for the role determines the store instance to export the bufferItem method to be used by the SPE to push items. When the value is
set to active, the SMS will call a predefined API provided by the SPE to retrieve data items from the input stream. This interface should also provide a push operation to an output stream when the access model of the store is set to \textit{Push}. A \textit{Pull} value for the access model determines the store manager to export the read operations determined by the registered read pattern(s). The default values for the role and access model parameters are \textit{Passive} and \textit{Pull} respectively.

### 4.6.2 Functional Parameters

\textit{Persistence}: When receiving value \textit{Persistent} in a \textit{createStore} call, the storage manager currently uses a Relational Database System to persistently store the data. As a result, it requires some information regarding the RDBMS: server URL, port, table name etc. It therefore creates a special kind of store whose job is to redirect all the requests from the SPE to the specified database. One challenge in the implementation is to translate the requests into SQL queries. Our current solution offers a simple mapping from (attribute name, value) pairs to \texttt{SELECT} statements as well as key-based updates. The storage manager has the flexibility to decide on when to persistently store the items or if it should keep them locally using caching techniques. That way, it relieves the SPE from the burden of managing these tasks itself (in Chapter 5 we investigate more on the correctness of joining streams with relational database tables). The value \textit{Transient} tells the storage manager to create its own main-memory based implementation of the store. The default value for the persistence parameter is \textit{Transient}.

\textit{Synchronization}: If a synchronization condition is present, a special mechanism based on blocking/ notification is employed for delaying reads until the condition is met. For example, a synchronization condition may look like: (“minute”, eq, \texttt{<minute\_value>}), meaning that the store should notify whenever there is data for the given minute (\texttt{<minute\_value>}).

\textit{Schema}: The next parameter evaluated is the schema(s) information (if present). The meta-data about the attributes and types is stored. One important piece of information is if all the items in the stream comply to the same schema. If no schema information is available, the default action taken is to consider that the contained items may have different schemas and no further assumption is made.
4.6.3 Performance-related Parameters

In implementing the internals of a store, state-of-the-art techniques were used to speed up access operations as well as to keep low memory consumption.

4.6.3.1 Implementation of the Update Patterns

The four basic write operations involved in any update pattern are: insertion, deletion, value replacement and garbage collection. We call garbage collection the process of physically removing the consumed and/or expired items from the store's internal structures. Each of the update patterns implies only a subset of the first three, while garbage collection is a required operation for avoiding unnecessary memory usage.

RANDOM Store implementation

Our RANDOM store implementation is depicted in Figure 4.5. This update pattern has the most constraints as it requires delete operations anywhere on the live (items that are neither expired nor yet consumed) part of the stored data as well as insertions and garbage collection.

The most suitable data structure for a RANDOM store is a linked list, because it is very flexible and easy to manage in the face of high-speed updates (i.e., insert and delete operations). A linked list allows the size of the materialized data to dynamically grow and shrink as needed. Furthermore, processing updates one item at a time is not a good option when dealing with fast updates: creating a new entry for each arrival dramatically decreases performance. In order to support high insert rates, a linked list of directors to blocks of items is built, in which items are ordered by their insertion times. A block

\[\text{Currently being written}\]

\[\text{Marked as consumed}\]

\[\text{Deleted}\]

\[\text{Directors}\]

\[\text{Blocks}\]

2 This is somewhat similar to the “sub-window” implementation in related work [54].

Figure 4.5: RANDOM Store Update Pattern Implementation
itself is implemented as an array of items. Not only is the array implementation simple, but also provides fast random access. Each block has a fixed size, representing the number of items that it can accommodate.

When a new item is to be inserted, it is redirected to the most recently created block in the first free position. If all blocks are full, a new one is created and the process continues.

When an item is to be marked as expired, its corresponding block is located using the list of directors and inside the block the item is marked as deleted. Any subsequent request for an expired item will fail.

For fast access to a certain item a combination of logical (used by the SPE in requests) and physical ids (used internally) is used, with simple mappings based on arithmetical operations for reducing overhead.

An item or a set of items can be marked as consumed in the same manner as expired items. Expired and consumed items are physically deleted from the store lazily, when the system becomes short of memory to make space for the newly arriving items. There are two approaches to garbage-collect the marked items: (i) only blocks which are fully consumed (i.e., all items in the block are marked) are removed from the linked list by removing the respective directors; or (ii) all the blocks are shrunk to physically remove all the consumed items in the system. The second approach is more aggressive as it also garbage-collects the partially-consumed blocks. Outdated items in a shared store are determined as the intersection of the item ids marked as consumed by all the consumer operators.

**FIFO Store Implementation**

One of the shortcomings of the RANDOM implementation is that in its context, blocks are continuously deleted and created. Furthermore, a block has to be locked for access, because a read operation may be requested for a block which is concurrently being written. This has a very important impact on performance with the continuous flow of items.

A consequence of having a FIFO update pattern is that consumed blocks may be reused instead of being deleted. That is, a block which has all its items marked as consumed, can be reused by replacing the consumed items with the new-coming ones, therefore avoiding deleting it and probably creating a new one later on. This is achieved by creating a circular linked list of blocks (like in Figure 4.6).
Furthermore, a read can only be executed after the requested item has been materialized ("read-behind"). In this case, the synchronization is achieved based on the \textit{id of the last materialized item}. Therefore, the block can be made lock-free.

Specifying which items are consumed is a very simple task: the smallest id of the last reported consumed items (in the sharing case) is computed, and all items up to that id are considered ready for garbage collection.

**IN-PLACE Store Implementation**

In the case of the IN-PLACE update pattern, there are no deletions, sometimes inserts and mainly replacements and reads. If a single block would be used for all the data items, reads would be executed really fast, but the block would have to be locked as replacements can occur in the meantime. Therefore, a set of blocks is implemented, to reduce congestion (similar to horizontal partitioning in relational databases). This update pattern supposes the existence of a value-based index for locating the items that need to be updated. The value-based index is presented in the following subsection.

**4.6.3.2 Implementation of the Read Patterns**

To avoid a rescan with each new request for an item, the access pattern can be used to build specific indexing and caching structures on top of the layout.
4.6. Store Instance Creation and Implementation

Value-based Indices

As in the case of value replacement, value-based access requests the existence of a special structure which we call a *value-based index*. The schema information becomes very important for the implementation.

A value-based index is created using the predicate push-down feature. That is, the SPE pushes the *where* clause of a query operator to the storage level. First of all, the predicate is translated into an *index schema*, which the SPE registers on the store instance. This means that the engine specifies a set of attributes (i.e., columns) and logical operator pairs.

For example, consider the following query: *What is the previous minute’s “min” toll for the segment “seg” in direction “dir”?*

The conjunction-based predicate is translated into an index schema with the following structure: 

\[(\text{min, eq}), (\text{seg, eq}), (\text{dir, eq})\]  

The store will then create an index structure based on the three attributes. When a request identified by the index schema is sent, the values are applied to the index using the logical operators and the items are returned.

Read Patterns

Redirecting the item requests to the appropriate blocks may lead to overhead. Therefore, as shown in Figure 4.7, a store implementation uses a cache which holds the last block which was involved in a read operation. When an item is requested from the store, this cache is probed if it contains the corresponding block. If so, the cached item is returned. On the other hand, if there is a “cache miss”, then the block which contains

![Figure 4.7: Speed-up Read Operations](image)
the requested item from the linked list has to be located.

When the list of blocks is large, a complete scan of the list of blocks should be avoided, and as such, a special structure which we call id_index can be used. This index is a tree-based implementation of a map that is optimized for locating the block an item belongs to. Consequently, the corresponding block replaces the current one in the cache. Sequential reads are facilitated by the presence of the cache, in which case the same block can be used for all subsequent requests going to that specific block. Having the actual information though determines that the id-index is removed, as it is not needed anymore: when an item is not in the current processed block, the linked list navigation methods can be used to move to the next block. Removing the index reduces the number of operations executed for a read, and avoids one synchronization point.

In the case that there are more readers subscribed to a store, each of them will have its own last accessed block saved in the cache.

Random access is facilitated by the id-index. In this case, probing the cache should be skipped, as any block could be accessed next.

Having information about the schema of the input items is of great help when the SPE requests for the value of a certain attribute. This is achieved by allowing an identification scheme that locates attributes inside items. Our previous description about locating items does not work in this case. For this reason another structure is created for directly locating attributes. This implementation only works if all the items comply to the same schema.

The clustered read access is a pseudorandom access with some locality. Therefore, of great help will be the id-index backed up by the caching technique because two subsequent requests have a high chance of reaching the same block.

### 4.7 Performance Study

In this section, we will show the performance of our approach through an experimental study on the Linear Road stream data management benchmark presented in Section 3.7. As presented in Chapter 3, we implemented the benchmark using MXQuery. MXQuery uses SMS as its underlying storage manager for all of its storage-related tasks (the results presented in the experimental section of Chapter 3 were obtained using an early implementation of SMS which did not make use of all the ideas presented in the current chapter).
There are two main goals of this performance study:

- We would like to show that our fine-grained parameter analysis meets a broad range of requirements, while value tuning improves the overall query processing performance in terms of response time and memory consumption.

- We would like to analyze the sensitivity of the query processing performance to changes in some of the storage parameters.

### 4.7.1 Performance with the Linear Road Benchmark

#### 4.7.1.1 The Linear Road Benchmark

The Linear Road Benchmark simulates the traffic on a set of highways and provides variable tolling based on accident occurrences and traffic statistics. The benchmark involves all in all, five queries, three continuous and two historical (details can be found in Section 3.7).

The measure of this benchmark is given by the Load level, representing the number of highways that can be handled by the query processing engine. A certain load level is considered to be achieved, when all the queries involved in the benchmark are answered within at most 5 seconds after the request entered the system.

#### 4.7.1.2 Linear Road Stores Implementation

Our Linear Road implementation contains a set of queries connected by SMS stores as shown in Figure 4.8, essentially the same implementation used in Chapter 3.

Next we present in detail the types of stores involved in the implementation (more details about the queries can be found in Section 3.8).

There are three major types of SMS stores involved in our implementation: a persistent historical store for the tolls assessed to cars in the previous 10 weeks; two stores, which because of their nature, are implemented as main memory relations (allowing key-based updates and reads); and a set of “streaming” stores. Except for the HISTORICAL TOLLS store, all other stores reside in main memory.

For the HISTORICAL TOLLS store we created a Store Instance which has value Persistent for its Persistence parameter. Through our design, we have offered therefore
a method to allow the access from a streaming query to data stored on disk, in this particular case using a MySQL [7] database instance.

The two main memory relations (noted with I and II in Figure 4.8), both store items that are updated and/ or retrieved by key (the vehicle id). Given these requirements, the most suitable implementation is offered by the one corresponding to the IN-PLACE update pattern. These two stores have value Passive for the Role parameter.

All the “streaming” stores are implemented as active stores (they pull items from their input streams of items):

The INPUT store holds items from the input stream created by the Linear Road’s data generator. It is shared by three queries (i.e., Car Position Reports, Balance Request and Daily Expenses).

The POSITION REPORTS store holds the car position reports used for computing the accidents and the per-minute statistics, as well as to determine which cars are crossing to a new segment. Therefore, three overlapping windows are created on top of this store. The largest window is the one created for computing the statistics per segment (5 minutes worth of data).

The CARS SEGM CHANGE store contains the position reports of the cars which changed their segment so that the drivers can be notified about relevant events (accidents or tolls charged).
The ACCIDENTS store keeps the data items that describe the accidents. It is shared by the Accident Notification query and the query that calculates the tolls (Toll Computation).

The TOLLS store contains the per-minute, per-segment tolls. The ACCIDENTS and TOLLS stores are examples of using the *Synchronization* parameter as both contain a condition on time (a request in a certain minute is blocked until data is available for that minute).

The streaming stores are the ones we focus on in this performance study, as this type of store is typical in general stream processing settings, and moreover, they do not have specific constraints, leaving room for different implementations and optimizations.

### 4.7.1.3 Experimental Tools

The experiments were run on a Linux machine with a 2.2GHz AMD Opteron processor and 4GB of main memory. SMS, like MXQuery, is implemented in Java. For this study, we used Sun JVM Version 1.5.0.09 (we kept the default settings of Java garbage collection). The maximum heap size that represents the available memory was set to 3GB.

One experiment runs for 3 hours, until all items from the input are processed, unless the maximum amount of memory allowed is exceeded. The input load (number of streaming items per second entering the system) increases during the execution. In order to be able to demonstrate the full impact of our algorithms, in the experiments, we set the benchmark load level to 2.5, sufficient to stress the system such that it had to fully utilize the allocated system resources.

In our first set of experiments, we used several metrics as performance measures, including: the average and maximum query response time, the number of alerts which exceeded the 5 seconds threshold, and the total amount of memory consumed. The query response times are measured for the toll alerts, as they represent the majority of the alerts in the system and are generated on the query path with the highest computational load.

### 4.7.1.4 Setup Configurations

Our first attempt was to implement a simple queue as the underlying structure for the store. The continuous deletes and inserts of items, as well as numerous read opera-
tions (MXQuery is a pull-based engine) made the store very inefficient as no special optimizations were conducted to support them. Of course, this implementation failed (the execution ran out of memory quite soon after the beginning of the experiment). Therefore, we used as the baseline, an optimized version of a general streaming store (Figure 4.9). It is based on a RANDOM update pattern implementation (the most general, no assumptions), containing both an id-index for random access as well as caching techniques for speeding up reads (again, no assumption about the read patterns).

In the experiments we compare three different configurations:

- **Baseline** In this configuration all the stores in the query network are implemented as Baseline.

- **Baseline + Update pattern.** In this configuration, we use the information about the update patterns and change the implementation of each store to reflect its actual update pattern (read pattern stays Random for all).

- **Baseline + Update pattern + Read pattern.** This configuration adds the read pattern knowledge, therefore determining some of the storages to change implementation to either Sequential or Clustered.

**Baseline + Update Patterns.**

Analyzing the queries, we determine that the streaming stores 1, 2, 4 and 5 (i.e., INPUT, POSITION REPORTS, CARS SEGMENT CHANGE, and TOLLS stores) have a FIFO
update pattern, whereas store 3 (i.e., ACCIDENTS store) has a RANDOM update pattern (see Figure 4.8) since accidents are not necessarily cleared in the order that they were detected. As Baseline already implements the RANDOM update pattern, we keep this implementation for the ACCIDENTS store. Knowing that the access pattern is FIFO helps reducing the congestion, as individual blocks need not be synchronized anymore.

**Baseline + Update Pattern + Read Pattern.**

In this configuration, we tune our stores to exploit the read pattern knowledge:

As a result, the INPUT and CARS SEGM CHANGE stores are configured as FIFO update and Sequential read (all the queries reading data from these stores execute a FOR expression over their content). For example, the Car Position Reports query written in MXQuery and reading from the INPUT store looks like in the following:

```xml
declare variable $INPUT external;
for $w in $INPUT
where $w/@Type eq 0 return $w
```

As can be seen, the query executes a a FOR loop over the entire store content and filters the car positions (the events having value “0” for attribute “Type”) and as such exposes a Sequential read pattern.

The POSITION REPORTS store is configured as FIFO update and is shared, therefore exposing three read patterns for each of the consuming operators: (i) Random (for the “Toll Computation” query which executes a group-by operation to compute the toll per segment), (ii) Sequential (for the “Segment Change” query, which only iterates over the content of the store to check for cars which cross segments) and (iii) Clustered (for the “Accident Detection” query whose first expression (in the figure we only show a single box which merges multiple expressions) detects the cars having speed 0). For example, in the following we show the query which detects the cars with speed 0 (to simplify the exposition, we omit the creation and insertion of a special event marking the end of the minute):

```xml
declare variable $POSITION_REPORTS external;
forseq $w in $POSITION_REPORTS 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
return (for $r in $w where $r/@Speed eq 0 return $r)
```
The query exposes a Clustered read pattern because of the order the items are accessed when checking the window boundaries.

The TOLLS store is configured as FIFO update and Random read (this is because the Toll Notification query uses value-based (segment id) lookup on the store to extract the toll corresponding to a certain segment).

Finally, the ACCIDENTS store is configured as RANDOM update and Random read (the read access on the store is executed through value-based lookup to extract information for a certain segment).

### 4.7.1.5 SMS Performance

**Block Size.** The first experiment we conducted tries to understand the effect of block size on the performance. We chose the Baseline configuration and ran a number of experiments with various the block size. For each of the runs, we measured the maximum response time. The results are presented in Figure 4.10.

As the plot shows, the block size significantly influences the performance. For the configuration with 100 items per block, the experiment ran out of memory before processing...
all input. A small block size causes greater pressure on the id-index for reads (it contains more entries). Furthermore, new blocks are created more often, which leads to more updates on the id-index. Large block sizes on the other hand increase contention. The block size yielding the lowest maximum response time appears to be somewhere around 10,000 items.

**Performance with Access Pattern Tuning.** For the next experiments, we set the block size to 10,000 items. Table 4.6 shows the performance results of our SMS approach compared to the Baseline (we present the median results of multiple runs for the experiments).

As expected, the Baseline configuration behaves very poorly, because over 13% of the alerts come too late. The optimizations made possible by our parameter tuning reduced the average response time by a factor of 6 (factor of 2 for maximum response time). Furthermore, the number of toll alerts that exceed the 5 seconds threshold is reduced from 13% to 2%.

For the three configurations, we computed the average memory consumption: every minute, we checked the amount of memory used, summed that up and divided it by the number of minutes (180 minutes). The average memory consumption is approximately the same for all the configurations. The same situation can be seen for the maximum memory consumption as well: for (i) Baseline, it was 1.3GB, for (ii) Baseline with update pattern knowledge, it was 1.2GB and for (iii) Baseline with read and update knowledge, was 1.4GB. As it can be observed, while the response time seriously improves, the changes in memory consumption are practically negligible, a result which suggests that the improvement is actually in the efficiency of access.

<table>
<thead>
<tr>
<th>Results Configuration</th>
<th>Average response time (seconds)</th>
<th>Maximum response time (seconds)</th>
<th>No. of alerts over 5 seconds</th>
<th>Percentage of total alerts</th>
</tr>
</thead>
<tbody>
<tr>
<td>Baseline</td>
<td>3.17</td>
<td>57</td>
<td>767,182</td>
<td>13.4%</td>
</tr>
<tr>
<td>Optimized for update pattern</td>
<td>0.715</td>
<td>29</td>
<td>190,753</td>
<td>3.3%</td>
</tr>
<tr>
<td>Optimized for update and read patterns</td>
<td>0.52</td>
<td>28</td>
<td>126,688</td>
<td>2.2%</td>
</tr>
</tbody>
</table>

*Table 4.6: Performance with Access Pattern Tuning*
4.7.2 Effect of Fine Tuning

In the next experiments we conducted a more detailed analysis of the influence imposed by some parameter values on the performance. For this purpose, we generated a couple of scenarios using parts of the Linear Road implementation.

4.7.2.1 Even Finer Tuning

For the purpose of this experiment we simplified the “Accident Detection” query to only select the cars which have speed 0 on a per minute basis (a slightly modified version of the query presented in Section 4.7.1.4). This means that the query maintains a tumbling window on the car position reports it receives as input and accesses it in a sequential manner, like in the following:

```java
declare variable $Store external;
forseq $w in $Store tumbling window
    start curItem $s_curr when fn:true()
    end curItem $e_curr, nextItem $e_next
    when $e_curr/@minute ne $e_next/@minute
    return
    (for $r in $w
        where $r/@Speed eq 0 return $r)
```

For that we use an SMS Store Instance which should expose the best access performance for a FIFO update pattern and a Sequential read pattern (Figure 4.11), but we show results with different configurations as well (Random read and Clustered read patterns).

The input consists of over 6 million car position reports. Given different configurations of the Store, we measure how long it takes for the query to consume the input. We then divide the number of items in the input to the running time to determine the throughput.

![Figure 4.11: Speed 0 Experiment](image.png)
4.7. Performance Study

The results are presented in Figure 4.12. In this plot, we can observe that giving even slightly “wrong” configurations to the Store, the performance degrades. The most interesting is probably the difference between having a Clustered and a Sequential read pattern: the Clustered implementation behaves like the sequential one as long as the requests fall in the same block. Whenever a request has to go to another block, instead of moving to the next one in the list, the Clustered implementation uses the id-index to locate the block. Using a Random pattern has such a poor performance compared to the Sequential implementation because of the continuous probing of the id-index with every request, when this action is actually unnecessary.

4.7.2.2 Effect of Sharing Store Data

Next, we examine the effect of using the Sharing parameter. For this purpose, we designed a test scenario using two queries: the Speed0 query presented in the previous experiment and a part of the Toll Computation query, and more specifically the one which computes the traffic statistics every minute (“Segm Stats” in Figure 4.13). They both use as input the car position reports. Not shared is a test in which the input stream is generated twice and therefore a different store instance is created for each of the queries. In the Shared scenario, both queries connect to one single store instance (a Shared store) which materializes the input stream (Figure 4.13). The same number of input items was used and again we measured throughput and this time memory con-
Figure 4.13: Effect of Sharing Store Data Setup

As expected, the No share setup consumes more memory (Table 4.7). Apart from higher memory consumption, the Not Shared setup had to do more computation: it has to generate the input stream twice and to materialize it into two storage instances. Therefore, it is no wonder that throughput was higher for the Shared scenario as opposed to the No Share one (Table 4.7). This experiment proves how important is for performance to support and to implement intermediate results sharing. Moreover, having a Storage Manager capable of providing the setup to allow easy intermediate results sharing greatly simplifies an application developer’s task.

4.8 Related Work

While there has been an extensive amount of work in the area of data stream management systems, the subject of stream data storage management has not seen much attention.

Several research prototypes for DSMS have been built (e.g., Aurora [18], STREAM [82], TelegraphCQ [36]), all of which proposed different ways to deal with the above outlined storage requirements (for the commercial products we do not have detailed information about the storage implementation). However, all of these systems only support a limited set of the requirements. In addition, they have a tight coupling between query processing and storage management by embedding storage management within the stream processing engine, which is in contrast to the proposed separation of concerns.

<table>
<thead>
<tr>
<th></th>
<th>Shared</th>
<th>Not Shared</th>
</tr>
</thead>
<tbody>
<tr>
<td>Average Memory Used</td>
<td>835MB</td>
<td>900MB</td>
</tr>
<tr>
<td>Average Throughput (tuples/s)</td>
<td>23'474</td>
<td>15'600</td>
</tr>
</tbody>
</table>

Table 4.7: Performance with Sharing Store Data
in this work. In this section, we briefly summarize the storage-related solutions that were proposed by the previous streaming systems.

Aurora \cite{aurora} uses two main forms of storage structures: (i) tuple queues (FIFO, append-only) for storing intermediate results of continuous queries, and (ii) connection points as a persistent cache to store stream tuples for historical/ad-hoc queries as well as to plug in static tables for hybrid queries. Sharing is supported by allowing multiple operators to access the output queue of an upstream operator. If queues do not fit in main memory, the system spills them to disk selectively, based on the processing priorities of the operators.

STREAM \cite{stream} also provides two basic storage structures: (i) inter-operator queues to connect the operators, and (ii) synopses to maintain operator state (also used for approximation). Sharing on tuple queues is supported in the same way as in Aurora. For synopsis sharing, STREAM provides a single physical store for the tuples, but multiple stubs (representing multiple views) for access. Queues and synopses are allowed to overflow to disk in case of memory limitations, but no sophisticated algorithms are provided.

TelegraphCQ \cite{telegraphcq} builds on several different types of dataflow modules, each with a different functionality. A State Module (SteM) is used to temporarily store tuples for stateful operators (e.g., join), and supports build-probe-eviction operations as well as indexed and shared data access \cite{ste}. Furthermore, the Fjord API allows multiple modules to be connected via pull or push queues \cite{fjord}. Finally, the PSoup extension on SteMs enables historical data access \cite{stems}. Golab et al also proposed storage techniques and index structures for sliding windows over data streams \cite{golab}. Basic windows are implemented as a linked list of tuples, while sliding windows are implemented as a circular array of pointers to these basic windows. On top of this, higher level storage structures based on attribute value aggregation are proposed. This work showed that indexing sliding windows over data streams can increase query processing efficiency. A follow-up to this work observed that different operators have different update patterns to their internal state, and this can be exploited for efficient query execution \cite{golab_follow}. Query plans are annotated with update patterns of their constituent operators, and these annotations drive the use of different operator implementations as well as different data structures for state maintenance and result storage. Our SMS approach also exploits read patterns in addition to the update patterns. Furthermore, SMS is not concerned with using different operator implementations due to its decoupled design from the query processor.

More recently, with the evolution of requirements on the DSMSs, there is an increased
interest in leveraging the experience gained with database systems. More specifically, DSMSs are built by adding continuous processing capability to relational database systems. Examples of such lines of work are represented by the system presented in [50] or DataCell [76] 77, a streaming system built on top of MonetDB [5], an open-source database system. We also advocate for the use of state-of-the-art database techniques to meet the streaming application requirements, but, as opposed to these systems, we propose a general framework.

4.9 Summary

This chapter addresses the requirements of data-intensive stream processing applications. Such streaming applications do not only stress the processing engine of a DSMS, but put up new challenges for the management of data: large, quickly changing datasets need to be handled with low overhead in terms of memory and CPU while providing efficient access and low response times guarantees.

To address these challenges, we conduct a detailed requirements analysis. As a result of this analysis, a key set of storage parameters is identified together with the possible values they should take under different conditions.

The values of these parameters (in particular read and update patterns) provide the means to optimize stream storage to achieve the desired performance goals. We also describe their implementations based on state-of-the-art techniques.

While some of these implementations have already been used in an ad-hoc manner within existing DSMS, our work takes the next logical step and decouples storage management from the processing engine. Similar to traditional relational database systems, this approach provides flexibility, adaptation to specific requirements and allows performance improvement. Our implementation of the storage manager, called SMS, provides a general-purpose storage system for DSMS and offers well-defined interfaces so that it can be tailored to different applications needs according to the storage parameters.

An experimental study of SMS on the Linear Road Benchmark shows that significant improvements in query response time can be achieved by tuning the access pattern parameters differently for different stores involved in the benchmark. In the study, we also analyze the effect of fine-tuning with the help of simpler test scenarios. The results obtained are very promising to prove that our decoupled design can greatly improve performance.
One issue that we did not address in this chapter is providing mechanisms to support transactional properties in the storage framework. This aspect is very important for the development of correct, reliable streaming applications as will be explained in Chapter 5 where we present the design and implementation of a transaction manager in SMS.
Chapter 5

Transactional Stream Processing

A variety of stream processing applications require correlating data from heterogeneous sources (streaming, stored relational etc.). Despite this trend, currently, there is no clean semantics for continuous query execution over arbitrary combinations of streaming and stored data sources in the presence of concurrent access and failures. One reason is that the streaming systems have different processing models and consequently each exposes its own transactional model. Additionally, the transactional properties are hard-coded in the processing models of the systems, making them inflexible and hard to understand.

In this chapter, we investigate applying the traditional transactional theory to cleanly define the correct interaction of a set of continuous and one-time queries concurrently accessing streaming and stored data sources. As such, we propose a unified model. Using this model, we express the transactional behavior of a set of state-of-the-art Data Stream Management Systems. Moreover, we show that the implementation of a transaction manager as part of SMS, the storage manager for streaming systems presented in Chapter 4, incurs practically no performance penalty when running the Linear Road Benchmark. At the same time, it provides correct execution even in the presence of failures.

5.1 Introduction

Often times, data stream applications involve access to multiple data sources, including not only purely streaming, but also stored ones, e.g., for correlating streams with historical data or for enriching them with additional meta-data. As the scale and complexity of these applications increase, it becomes harder to ensure their correct execution in the
presence of concurrent processing or failures over these multiple data sources. This chapter investigates this challenge.

5.1.1 Motivating Scenario

To illustrate the problem, let us consider a simple scenario. Suppose that there is a set of devices whose functioning is sensitive to temperature. Each of the devices has a minimum and a maximum temperature specification, defining the interval in which it functions properly. These specifications are stored in a database table \( R \). Furthermore, a set of sensors take real-time temperature measurements (on a Celsius scale) for these devices and make them available as an input stream \( IS \), with schema: the id of the device, the temperature value and the timestamp of the measurement. We would like an alert containing the id of the device and the temperature value to be generated, whenever a temperature reading falls outside the acceptable temperature interval of the corresponding device. A typical DSMS would model this scenario with the following continuous query, as shown Figure 5.1: for each incoming temperature reading event in \( IS \), probe the specifications table \( R \), and raise an alarm whenever a violation is detected (e.g., as in the case of event \( (D2, 24, 4) \) which generates alarm \( (D2, 24) \)).

Now suppose that, after the arrival of event \( (D2, 24, 4) \), the temperature readings scale of the sensors is changed to Fahrenheit. Of course, the specifications in \( R \) must also be updated to reflect the change in temperature scale; but if the updates on the table do not happen before the new sensor measurements arrive, false alarms can be generated. For example, in Figure 5.1 event \( (D3, 50, 5) \) generates a false alarm - 50 °F actually represents 10 °C, which happens to be in the accepted range for device D3.

In order to obtain a correct execution, a DSMS should be able to order the updates on
table $R$ and the temperature readings in stream $IS$. Today, this task cannot be directly achieved by any DSMS.

In addition to the update problem illustrated above, failures can also lead to incorrect executions, while many streaming applications require that continuous queries be persistent over system crashes. In our example scenario, assume that while processing event (D2, 24, 4), the DSMS experiences a failure. Since all the temporary state created by this event is lost, the alarm it would have generated in a normal execution scenario, will never be generated. Therefore, it is important to be able to specify that this event should only be processed to completion (until the results it generates are committed to the output), or reprocessed in case of failure to do so.

### 5.1.2 The Problem

The fundamental problem described above stems from the differences between the two query processing worlds: stream processing operates on events, while traditional stored data sources work with operations (read/update). While an ordering may be defined among events (e.g., based on timestamps, arrival order, etc.) or among operations (e.g., defined by a transactional model), there is no well-defined order across events and operations, which makes their direct comparison impossible. Moreover, traditionally, queries are one-time, while stream processing uses long-running, continuous queries.

Intuitively, there are two alternative solutions to the above problem: (i) either non-streaming sources can be transformed into streams (e.g., by creating an event for each database operation), or (ii) streaming sources can be interpreted as data sources with regular read/update operations.

Most of today’s DSMSs implement solution (i). The main problem with this approach is that, as a result of converting the stored data sources into locally recognized inputs (e.g., streams) and because of the high degree of heterogeneity in these systems’ query execution models (as pointed out in Chapter 2), each ends up proposing its own implicit “transactional model”. Furthermore, the transactional properties are usually embedded into these systems’ execution models and operator semantics, making them hard to understand and inflexible to use.
5.1.3 Contributions

In this chapter, we propose a unified transactional model for streaming and stored data sources based on solution (ii). Our approach reuses decades of research on transactional database theory and exploits the page model [95], to define correct executions of both continuous and one-time queries over an arbitrary mix of stored and streaming data sources. The basic insight is that we treat streaming and stored inputs uniformly: they are simply data sources to the continuous queries on which reads and updates are executed. Moreover, a continuous query is modeled as a possibly infinite sequence of one-time queries. In this light, the problem becomes similar to the one addressed by the traditional transactional model.

We first analyze the page model [95] with respect to its expressive power of allowing the correct execution of streaming applications. We then extend traditional transactions with: (i) events, which are represented by write operations, and (ii) continuous queries, which are represented by read and write operations on the data sources (including both inputs and outputs). As such, events and/or individual continuous query executions can be grouped together into transactions, thus flexibly defining isolation units not bound to any specific query execution model or operator semantics. We want our model to be general enough to be applicable to any query language (e.g., CQL [22], CCL [1], XQuery, StreamSQL [11], etc.), by adding basic primitives such as commit and abort. Nevertheless, designing the right syntax to specify transactions for continuous queries is beyond the scope of this work.

We then propose conflict-serializability to define the correct interleaving of operations belonging to these transactions, but similar to the traditional case, weaker constraints on the ordering can be defined.

The result is a clean semantics for continuous query execution over streaming and stored data sources even in the presence of failures, which, we think could also be successfully applied to support high availability [62] or even “time travel” [16] on streams. Moreover, our analysis of a series of state-of-the-art commercial and academic DSMSs shows that their transactional behavior can be described using the unified model.

In summary, this chapter makes the following contributions:

- It defines the space of possible executions of continuous and one-time queries over arbitrary combinations of streaming and stored data sources, in the presence of concurrent processing and failures.
- It shows how events and operations can be arbitrarily grouped into transactions.
and that conflict-serializability can be naturally applied as an example criterion for defining correct executions. As such, we propose a unified transactional model.

- It presents the transactional models of some state-of-the-art streaming engines expressed in this execution space.
- Using the Linear Road Benchmark (Section 3.7), it shows the performance of the first implementation of a unified transaction manager, built as part of SMS (Chapter 4).

The remainder of this chapter is organized as follows: we continue in Section 5.2 where we present a brief summary of the basic concepts in the traditional transactional model and motivate its choice for defining the semantics of continuous queries over streaming and stored data. Then, in Section 5.3, we characterize the execution space and give the formal definitions for transaction and execution correctness in the unified transactional model. Section 5.4 presents the related work as well as the transactional behavior description of five academic and commercial DSMSs. Section 5.5 presents the implementation of a transaction manager for SMS, the storage manager for streams described in detail in Chapter 4. After presenting our experimental setup in Section 5.6, we show the results in Section 5.7. Finally, Section 5.8 presents a summary of this chapter.

5.2 Traditional Transactional Processing

This section presents a high-level description of the basic transactional concepts (a detailed description can be found in [95]). In addition, it explores the expressivity of the page model and motivates the decision of choosing it as the model for defining the semantics of (continuous) query execution over streaming and stored data sources.

5.2.1 Transaction

The transaction is an abstract concept designed to relieve the application developer from the burden of dealing with concurrent access on data and failures of the application or even of the entire system.

The canonical example for transaction usage is the money transfer between two bank accounts. In short, the transfer process contains two important operations which have to be executed together or none at all, otherwise the money can be lost: subtract the
amount to be transferred from the source bank account and add it to the destination bank account. If the two operations are grouped into the same transaction, the transactional system implementing the transfer guarantees that they will be executed together or none at all.

In summary, a transaction is defined as an ordered set of operations executed as a unit against a set of data objects (e.g., tables, records, pages) and having the following properties:

- **Atomicity.** The operations in a transaction are executed atomically: either all are executed or none is.

- **Consistency.** A transaction leaves the data objects it changes in a consistent state (e.g., in a relational database system, consistency is defined through integrity constraints).

- **Isolation.** Each transaction is executed as if in isolation, i.e., like there were no other transactions concurrently accessing the same data objects.

- **Durability.** As soon as a transaction successfully completes (i.e., it is committed), the changes it makes to the data objects survive subsequent failures.

In a transactional system, there will usually be multiple transactions competing for the same data objects.

If the transactions are executed serially (i.e., all the operations from a transaction followed by all the operations of another transaction), the performance of the system is poor: there can be only one single process at a time which accesses a resource. Therefore, to reduce the contention and consequently increase the performance, the operations composing the transactions should be allowed to interleave.

### 5.2.2 Anomalies

Executing multiple transactions concurrently on the same data objects, may result in a series of unwanted situations, which are called anomalies.

**Phantom Read.** A phantom read anomaly occurs when two distinct reads on a data object (e.g., a table) executed as part of the same transaction return different sets of results, because of a concurrent transaction inserting/ removing elements in/ from the data object.
**Dirty Read.** A dirty (uncommitted) read anomaly may arise when a transaction reads a data object (e.g., a record), whose content has been changed by a concurrent transaction but not committed. If the transaction which updates the data object is eventually aborted (and consequently the value of that data object is restored to the state before the transaction started), the other transaction will read a “dirty” value.

**Non-repeatable Read.** A non-repeatable read may occur when two transactions are not properly isolated and one reads the uncommitted effects of the other. The difference to a “dirty” read is that none of the transactions has to be aborted in order for the anomaly to occur.

**Lost Update.** A lost update anomaly occurs when two transactions concurrently modify the same data object. Obviously, one of the two updates will be overwritten, i.e., lost.

In order to avoid these anomalies, a series of isolation levels were defined, each suppressing one or more of the anomalies. Among them, serializability is the isolation level which prevents the occurrence of all these anomalies.

### 5.2.3 Transactional Model Design

When designing a transactional model, the following steps are taken [95]:

1. First, the indivisible operations which can be executed on the data objects have to be introduced.

2. Next, the transactions are defined as ordered sets of operations on the data objects.

3. Histories and schedules are defined as interleaving of operations from multiple transactions and represent the abstract notion of concurrent execution.

4. Among the producible histories, the correct ones must be identified.

5. Algorithms and protocols have to be built, which are guaranteed to generate correct schedules in an on-line manner as the operations are submitted to the system.

In this dissertation we analyze an already existing model, named the page model [95]. In this model, the data objects are represented by data pages (with reference to the main memory data pages, but not necessarily bound to them) and the indivisible operations are reads and writes on these pages. For example, operation $r(x)$ means read data page $x$, while $w(x)$ represents write data page $x$.

A transaction is defined as a partial ordering on a finite set of reads and writes executed against a set of data pages. The valid histories are identified through a notion of correctness (final state/view/conflict serializability), which is basically an ordering
condition on the operations that a valid history (or schedule) must satisfy.

In conclusion, the page model abstracts from the semantics of the operations and models concurrency based only on syntax correctness.

5.2.4 Transactions and Stream Processing

In essence, a transactional model defines visibility (by grouping operations in transactions and defining an order among the operations) and recoverability (a transaction also defines the recovery unit). These two aspects are exactly what we need in order to model the correctness of execution.

Through its simple design, the page model offers the necessary abstraction to allow the unification of streaming and stored data sources. Moreover, it allows us to detach the execution correctness definition from the semantics of operators, a very important concern considering the heterogeneity encountered among different DSMSs (these differences have been explored in detail in Chapter 2).

In addition, the algorithms for implementing schedulers are a natural fit for the streaming execution model: they have to allow on-the-fly scheduling of streams composed of access requests.

In the next section, we will formally present the unified model as well as the modifications to the page model which allow it to accommodate events and continuous queries.

5.3 Transactional Stream Processing

Following the transactional model design steps presented in Section 5.2.3 in this section, our unified transactional model over streaming and stored data sources which extends the page model is presented. Moreover, we show how the scenario presented in Section 5.1 can be correctly implemented using the unified model.

The main challenge in applying the page model is that streaming sources operate with events while the basic execution unit for the transactional model is the operation (more specifically, either a read $r$ or a write $w$). Moreover, continuous queries have different semantics than one-time queries. The problem is how to make these different units comparable, so as to be able to define correct executions.
5.3. Transactional Stream Processing

5.3.1 The Data Model

First, we need to characterize our space with respect to the data sources (query inputs) and sinks (query outputs), and their corresponding operations. We propose to treat all the data sources (streaming, relational, stored non-relational etc.) uniformly: they are all sets of data items on which operations (reads and writes) are executed, closely following the page model.

**Relation.** A relation, which is traditionally defined as an unordered set of tuples (data items) sharing the same schema, is easily integrated into this space. The operations exposed by a relation are insert (new tuples are added to the relation), delete (remove tuples), value update (replace existing tuple attributes with new values) and read (obtain the values of certain tuples). In the page model, these high-level operations are translated to reads and writes on data items (e.g., pages, individual records etc.) belonging to the relation.

**Stream.** A special type of data source, the stream has been defined in many ways. As explained in [80], the difference lies in the interpretation of the streams which is dependent on the application. In [80], an example is given for a sensor temperature readings stream that can be represented as: a sequence of readings (like in our scenario in Section 5.1), a sequence of changes to previous sampling points or a sequence of readings which are generated only if the temperature changes from the previous sampling point etc. Moreover, the stream definitions proposed so far embed both the semantics of the data as well as the semantics of the operations that can be executed on the stream [80].

In order to apply the page model, we need to abstract from a stream's data and operations semantics.

We adopt the general definition presented in Chapter 2: a stream is a possibly infinite partially ordered set of elements, i.e., events. An event is composed of a tuple and one or more ordering attributes. Thus, a stream can be viewed as an unbounded set of events on which the only possible update operation is insert (a new event), while delegating the actual interpretation of the content (e.g., whether the order attributes can be used in queries) and order to the DSMS.

As such, we convert each event arrival to a write operation (insert → write), therefore allowing the comparison of events with access operations. As an example, in Figure 5.1 assuming a timestamp-based ordering, the arrival of event (D3, 50, 5) composed

---

1In the following, for ease of exposition, we will generically name data sources both the data sources and the sinks, whenever the distinction between the two is not required.

2Data item = a general term to define the data units the sources are composed of
of tuple (D3, 50) and the ordering attribute (5), will be represented by a write on the input stream, \( w(IS) \).

A stream’s state is represented by its content and it can be retrieved through a read operation. As can be observed in Figure 5.1, a read operation after the arrival of event (D2, 24, 4) will return the state of the stream containing events \( \{(D3, 10, 3) \ (D2, 24, 4)\} \). A read operation on the stream after the arrival of event (D3, 50, 5) will return state \( \{(D3, 10, 3) \ (D2, 24, 4) \ (D3, 50, 5)\} \) and so on.

We consider the most general case in which there are multiple writers (processes which try to concurrently append new events to the stream) and multiple readers (continuous queries which process the events in the data streams). It is important to note that, because the only update operation executed on a stream is insert, a single anomaly may be generated as a result of concurrently accessing a stream and that is a phantom read (Section 5.2.2).

**Other Data Sources.** In the same way, other types of data sources with different sets of supported access operations (which could be represented through read and write operations) can be included in this space: e.g., a file in a file server or streams which allow future revisions.

### 5.3.2 The Queries

Another challenge arises when dealing with continuous queries. Continuous queries are issued once and run continually sending new generated results to the user application which registered them. Given their continuous nature, streaming queries are not a good match for a transactional model, as transactions are closely tied to a one-time query model.

Our proposal is to then represent a continuous query as a possibly infinite sequence of one-time traditional queries which are fired as a result of the data sources being modified (arrival of new events, update of existing inputs etc.) or by periodic execution (e.g., every second). So, basically, a one-time continuous query execution can be translated into read operations on all its input data sources plus (possibly) a number of write operations, corresponding to the results it may generate (or updates for non-streaming outputs).

For example, in the devices scenario (Figure 5.1 presented in Section 5.1), suppose the arrival of event (D2, 24, 4) triggers the query execution. The join execution is then represented by a read on the input stream, \( r(IS) \) and a read on the table, \( r(R) \) followed
5.3. Transactional Stream Processing

by a write on the output stream, \( w(\text{OS}) \), corresponding to the generation of the event (D2, 24).

In summary, our design space is composed of a set of data sources and sinks on which updates and one-time queries are executed, like depicted in Figure 5.2. As each update theoretically generates a distinct data source state, we need a way to specify which states can be made visible and what the recoverability units are in case of failures. Consequently, a notion of transaction has to be defined and this is what the next subsection is concerned with.

5.3.3 The Unified Transactional Model

A transaction in our design space obeys the traditional definition (a partially ordered set of operations), but we adapt it to the streaming environment in order to accommodate events and continuous queries.

Before giving the formal definition for transactions, we need to introduce the notion of partial order [95]:

**Definition 10 (Partial order)** A partial order on a set \( S \) is a relation \( R \subseteq S \times S \), such that:

- \( \forall s \in S, \text{ then } (s, s) \in R \) (Reflexivity)
- \( \forall s, t \in S \text{ such that } (s, t) \in R \land (t, s) \in R, \text{ then } s = t \) (Antisymmetry)
- \( \forall s, t, p \in S \text{ such that } (s, t) \in R \land (t, p) \in R, \text{ then } (s, p) \in R \) (Transitivity)

A transaction in the unified model is formally defined as in the following:
Definition 11 (Transaction) Given a set of data sources and sinks \( DS = \{ S_i, S_o, \ldots \} \) composed of data items, a transaction is represented by the pair \( T = (O_T, \leq T) \), where \( O_T \) is a finite set of operations (of the form \( r \) or \( w \)) on the data items and \( \leq T \) is a partial order on the operations such that:

- Each new streaming event \( e_p \), is represented by a write operation on the corresponding stream \( w(S_i) \) and each (one-time) continuous query execution is represented by a read on all the input data sources of the query possibly followed by a set of writes on the data sink(s) \( w(S_o) \).

- \( \forall o_p, o_q \in O_T \), two operations which access the same data item one of which is a write, then either \( o_p \leq T o_q \) or \( o_q \leq T o_p \) (i.e., operations on the same data item are ordered).

Transactions can be divided into two classes: local and global. A local transaction involves only data sources which are part of the same system (e.g., the same database system, the same streaming system). In a global transaction, the operations are executed on data sources belonging to multiple systems (e.g., a database system and a streaming system). In essence, a global transaction is composed of the union of all the local executions of that transaction in the different systems.

For example, in Figure 5.1, the arrival of event \( (D2, 24, 4) \) generates the join operation execution over the stream \( IS \) and the relation \( R \) which produces event \( (D2, 24) \) to the output stream \( OS \) as explained earlier. Suppose the application implementing this scenario specifies that all these tasks (arrival of event, query execution, output of results) should be grouped in a single transaction. The resulting global transaction will have the following structure: \( T_1 = (w_1(IS) r_1(IS) r_1(R) w_1(OS)) \). That is, the event's arrival in the input stream, \( w_1(IS) \), followed by a one-time continuous query execution: \( r_1(IS) r_1(R) w_1(OS) \). The local executions of this global transaction are then: (i) in the DSMS, \( T_1(DSMS) = (w_1(IS) r_1(IS) w_1(OS)) \) and (ii) in the DBMS, \( T_1(DBMS) = (r_1(R)) \).

Traditionally, transactions can end in two ways: successfully or not. To express the termination of a certain transaction (when that state is known), a special kind of operations is used: Commit and Abort. As such, a transaction can be in one of three (typically more, but these are the basic ones) states: active (is currently running and has not yet reached a final state), committed or aborted (corresponding to the two possible termination states).

In our model, similar to the traditional one, Commit is an operation which defines the successful termination of a transaction (all the operations in the transaction have been
executed with no error) such that the changes made on the data sources as part of this transaction or the new events in the streaming data sources can be made visible to other transactions. As for stored data sources, commit also specifies that these changes are made persistent.

An Abort operation expresses that while processing a transaction, something happened (a violation of consistency, system crash etc.) which interfered with the normal execution. As aborted transactions may leave the data sources in an inconsistent state, their effect should be undone. As a result, the data in the sources appear as if this transaction had never been executed at all.

Suppose that while executing transaction $T_1$, the streaming engine encounters an error, such that operation $w(OS)$ cannot be executed. Nevertheless, the event $(D2, 24, 4)$ is written in the input, most probably is part of the internal state of the join operator, but an alert will never be sent (the output result generation). To make sure that no alert is missed, the streaming engine should return to the state before the triggering event arrived in the stream ($r(IS)=\{(D3, 10, 3)\}$) and possibly try to execute the transaction again.

A important consideration on transaction visibility are the isolation levels [27], since many DBMSs relax isolation to improve performance. In this respect, for our unified model, the streams need some special consideration: since we restrict the update operations on streams to insert, data cannot be recalled or modified. In the strictest sense, the use of streams calls for serializability. If weaker isolation levels are required, instead of a stream, another type of data source that supports cascading deletes and/or compensations should be used. Such data sources would closely resemble those proposed in Revision Processing [88].

The definitions for (serial) execution history, schedule, conflict, conflict-equivalence and conflict-serializability can be reused without change [95]. For completeness, in the following, we list these definitions.

**Definition 12 (History)**  Given $T = \{T_1, T_2, ..., T_n\}$ a (finite) set of transactions where each $T_i \in T$ has the form $T_i = (O_{T_i}, \leq_{T_i})$, a history is a pair $H = (O_h, \leq_h)$, such that:

- $O_h \subseteq \bigcup_{i=1}^{n} O_{T_i} \cup \bigcup_{i=1}^{n} \{C_i, A_i\}$ and $\bigcup_{i=1}^{n} O_{T_i} \subseteq O_h$ (the operations belonging to the transactions are all contained in the history plus a control event (C or A) for each transaction)

- $\forall i, 1 \leq i \leq n, C_i \in O_h \iff A_i \notin O_h$ (for each transaction there is either a commit or an abort, but not both)
- \( \bigcup_{i=1}^{n} T_i \subseteq h \) (all transaction orders are contained in the history order)

- \( \forall i, 1 \leq i \leq n, \forall \text{op} \in O_{T_i}, \text{op} \leq h A_i \) or \( \text{op} \leq h C_i \) (the control operation (A or C) is always the last operation of a transaction)

- Every pair \((\text{op}_r, \text{op}_w)\) of operations, \(\text{op}_r, \text{op}_w \in O_h\) from distinct transactions, accessing the same data item, one of which is a write, are ordered in \(H\) in such a way that either \(\text{op}_r \leq h \text{op}_w\) or \(\text{op}_w \leq h \text{op}_r\).

**Definition 13 (Conflict)** Two operations on the same data item, such that at least one is a write, are said to be in conflict.

**Definition 14 (Schedule)** A schedule represents a prefix of a history.

**Definition 15 (Serial History)** \(SH = (O_{sh}, <_{sh})\) is a serial history if \(\forall T_i(O_i, \leq_i), T_j(O_j, \leq_j), T_i \neq T_j, T_i, T_j \in SH, \forall \text{op}_{ik} \in O_i \text{ and } \forall \text{op}_{jl} \in O_j, \text{ either all } \text{op}_{ik} <_{sh} \text{op}_{jl} \text{ or all } \text{op}_{jl} <_{sh} \text{op}_{ik}\) (a history \(SH\) is serial if for any two distinct transactions \(T_i\) and \(T_j\) contained in \(SH\), all the operations belonging to \(T_i\) are ordered in \(SH\) before all the operations in \(T_j\) or vice versa).

**Definition 16 (Conflict-Equivalence)** \(\forall H_i, H_j, H_i = (O_{hi}, <_{hi}), H_j = (O_{hj}, <_{hj}), H_i \equiv H_j \Leftrightarrow O_{hi} \setminus \{A, C\} = O_{hj} \setminus \{A, C\} = O \text{ and } \forall \text{o}_k, \text{o}_l \in O \text{ s.t. } \text{o}_k, \text{o}_l \text{ are in conflict, then } \text{o}_k <_{hi} \text{o}_l \Leftrightarrow \text{o}_k <_{hj} \text{o}_l\) (two histories \(H_i\) and \(H_j\) are said to be conflict-equivalent iff they contain the same operations and they order all the conflicts in the same way).

**Definition 17 (Conflict-Serializability)** A history \(H\) is said to be conflict-serializable if it is conflict-equivalent to a serial history.

Testing a history for its membership to the conflict-serializable class can be done efficiently by using **conflict graphs** [95]. A conflict graph is a directed graph in which the nodes represent the transactions and the edges symbolize the conflicts between them. The orientation of the edges follows the order of conflicting operations in the history. A cycle in the graph indicates that the underlying history is not conflict-serializable.

Before enunciating the correctness definition, we also present two notions: **local/global schedule** and **global conflict-serializability** [95]. In a multi-system environment, a global schedule represents the union of schedules executed locally at each of the systems (the local schedules). A global schedule (history) is globally conflict-serializable if it has the conflict-serializability property.
5.3.4 Execution Correctness

Our problem can be now described as follows: given a finite set of data sources and sinks \((DS_i, 1 \leq i \leq n)\), an infinite set of updates on the data sources \((U_j, 1 \leq j)\) and an infinite sequence of one-time queries accessing the sources, some of which (updates and/or queries) may be grouped into transactions, what defines a correct execution?

In this work, we propose to use conflict-serializability as a criterion to specify correct interleaving of operations:

**Definition 18** A history of operations belonging to a set of transactions \(T = \{T_1, T_2, \ldots, T_n, \ldots\}\) generated as a result of executing a set of queries (one-time and continuous) \(Q\) over a set of data sources and sinks \(DS\) is correct if it is (globally) conflict-serializable.

Conflict-serializability is an example of how our model can be used to define correct execution. As needed, it may be restricted or relaxed and, as in [98], different correctness levels can be defined.

For example, processing the events in their arrival order in a stream is important for the correct implementation of many streaming applications. Moreover, the majority of streaming systems rely their execution on the arrival order of the events. When we modeled the stream as part of the execution space, we abstracted from the semantics of a stream and part of the information we ignored was the order of events. To allow the correct execution of this type of applications, the events’ arrival ordering can be enforced as order between the transactions. As a result, a more constrained level of correctness can be defined (similar to Strong Consistency in the above-cited work):

**Definition 19 (Conflict-serializability with Arrival Ordering (CSAO))** A history of operations belonging to a set of transactions \(T = \{T_1, T_2, \ldots, T_n, \ldots\}\) generated as a result of executing a set of queries (one-time and continuous) \(Q\) over a set of data sources and sinks \(DS\) is correct if it is (globally) conflict serializable and conflict-equivalent to a serial history in which the order of transactions generated by sequences of events (stream updates and corresponding continuous queries executions) obeys the events’ arrival order in their respective streams.

Our main motivation behind choosing serializability is to provide a well-understood, general definition. Alternatively, we have also considered using other criteria such as view/final state serializability or snapshot isolation [27], but found problems with these. More specifically, the former is impractical to implement since schedules are not monotone [95] in that a new operation can transform an illegal schedule into a legal one [83].
Although Snapshot Isolation is the protocol of choice for many database systems because it improves performance, its main disadvantage is that it is not obvious how to define snapshots, or more specifically to assign timestamps to versions in a heterogeneous environment such as the DSMS. One the other hand, in the context of serializability, there are widely used protocols (e.g., SS2PL [95]) which generate globally serializable schedules, by only enforcing local serializability (Section 5.5).

5.3.5 The Unified Transactional Model in Action

We can now revisit our scenario in the Introduction (Figure 5.1) and analyze it with respect to our unified transactional model (to make the example easier to follow, we only assume committed transactions).

5.3.5.1 Defining Transactions

As previously assumed, for each new event, the streaming system probes the database table for specifications and outputs alerts in case of violations. Suppose that the arrival order of the streaming events is the one depicted in Figure 5.1 by looking at the stream’s representation from right to left (event (D3, 10, 3) followed by (D2, 24, 4), followed by (D3, 50, 5) etc).

As presented in the Introduction, the requirement of our use case scenario is that each device temperature measurement has to be compared to the device specifications in the same scale (the assumption we make is that once the first Fahrenheit measurement for a certain device arrives, all the following will have values on the same scale). More specifically, this requirement translates to the following two conditions:

- The update on the table has to happen before the first Fahrenheit measurement for the respective device, and
- No Celsius measurement should see specifications in Fahrenheit scale.

One way to meet the former requirement is to group the device’s specification update, \( w(R) \), and the first temperature measurement in Fahrenheit scale corresponding to that device into the same transaction, with the update on the table preceding the processing of the event. This way we obtain the desired visibility of the table update by taking advantage of the ordering of conflicting operations in a transaction. In fact, this grouping
also ensures that the specification update and the processing of the event happen atomically. As a result, we obtain four transactions:

1. The arrival of event \((D3, 10, 3)\) in the stream and its processing compose transaction \(T_1\):
   \[
   T_1 = (w_1(IS) r_1(IS) r_1(R) C_1)
   \]
2. The arrival of event \((D2, 24, 4)\) in the stream, its processing and the generated alarm \((D2, 24)\) form transaction \(T_2\):
   \[
   T_2 = (w_2(IS) r_2(IS) r_2(R) w_2(OS) C_2)
   \]
3. The arrival of event \((D3, 50, 5)\) in the stream, D3’s specification update \((D3, 32, 59)\) on the table and the processing of the event are grouped in transaction \(T_3\):
   \[
   T_3 = (w_3(IS) w_3(R) r_3(IS) r_3(R) C_3)
   \]
4. The arrival of event \((D2, 61, 5)\), D2’s specification update \((D2, 14, 68)\) in the table and the processing of the event, all belong to transaction \(T_4\):
   \[
   T_4 = (w_4(IS) w_4(R) r_4(IS) r_4(R) C_4)
   \]

As the update on the relation appears in a transaction before the reads corresponding to the processing of the event, the application developer can reason about the order in which the operations will be executed. Moreover, if a failure occurs before the transaction commits, neither the update on the relation nor the processing of the event will be performed. As a result, the update on the relation will only be executed in combination with the successful processing of the event (one would naturally expect a streaming system which requires that no event is missed, to retry processing an event until successful completion).

### 5.3.5.2 Applying the Correctness Criterion

Next, the correct interleaving of operations corresponding to the previously defined transactions needs to be specified.

Taking into account the assumption we made about the order of temperature readings, processing the events in the order of arrival guarantees that no Celsius measurement will be compared with Fahrenheit specifications. Therefore, the correct histories for our example application will be Conflict-Serializable with Arrival Ordering, the correctness criterion defined earlier in the section. More specifically, a correct history has to be conflict-equivalent to the following serial execution: \(T_1 \rightarrow T_2 \rightarrow T_3 \rightarrow T_4\).

Alternatively, as the previously defined serial ordering is quite restrictive and allows little
parallelism, a more relaxed ordering can be defined: the processing of temperature measurements corresponding to a certain device has to happen before the update of that device’s specifications to Fahrenheit scale. That is, a history for this scenario is correct if it is conflict-equivalent to a serial execution in which \(T_1 \rightarrow T_3\) and \(T_2 \rightarrow T_4\), i.e., \(T_1\) appears in the serial history before \(T_3\) and \(T_2\) before \(T_1\).

### 5.3.6 ACID for Streams

Serializability is based on the assumption that the transactions obey the ACID (Atomicity, Consistency, Isolation and Durability) properties. In this subsection, we analyze these properties from a streaming perspective.

**Atomicity** and **Isolation** is what we covered previously in this section.

Traditionally, **Consistency** refers to maintaining the integrity constraints of a database. For streams, Stream Schema \([48]\) can be used to model constraints (and validate their consistency), like (dis)order or compliance to a certain schema.

The most interesting property from a streaming point of view is **Durability**. Traditionally, durability specifies that the changes made by a transaction are made persistent if the transaction is committed. As streams are not persistent, it appears that the streaming transactions do not have the durability property. Nevertheless, if we think of durability as a property stating that operations of a committed transaction survive failures, the durability semantics for streams is then: the events of committed transactions will never be reprocessed or duplicated.

### 5.4 Related Work

In this section we will place our work in the current stream processing context, both with respect to already implemented systems, as well as theoretical concepts. Moreover, we will describe the transactional execution behavior of five state-of-the-art industrial and academic Data Stream Management Systems.

#### 5.4.1 Concepts

Conceptually, the transactional stream processing model that we propose relates to the previous work in four areas:
Relational Database Systems. There has been a lot of research on transactional models for computational systems, and more specifically relational database systems, for which concurrency control is specified through a formal model, what we called the traditional transactional model. Starting with the degrees of consistency [60] which later became the basis for the SQL-92 standard and continuing with more relaxed isolation levels [27], various extended models were proposed, like nested transactions [81] or sagas [51].

Data Warehousing Systems. The similarity between stream processing and materialized view maintenance [61] has long been recognized. Materialized views are like continuous queries in that a view has to be continuously updated in response to changes in its base relations. By treating streaming and stored inputs uniformly, our problem becomes similar to materialized view maintenance. In this domain, we have found three lines of work that are closest to ours:

First, Zhuge et al propose a set of algorithms to maintain the consistency of a data warehouse at various levels of correctness, when the view computation is disconnected from the updates at the sources [98]. While we focus more on capturing isolation properties as a definition of correctness, this work proposes the algorithms to ensure a predefined set of consistency levels.

Second, Chen et al suggest a transactional model that defines the view maintenance process as a special transaction. The transaction consists of two parts which can commit separately: the update at the source and the view maintenance query [38]. In this case, the warehouse update anomaly problem can be rephrased as the serializability of these transactions. Our work is more general: the update and the continuous query execution are not necessarily part of the same transaction.

Third, Jagadish et al propose the chronicle data model which extends materialized views to include chronicles – a form of data streams [64]. A chronicle algebra over relations and streams includes a join operator that synchronizes each tuple from the chronicle with the relation version which existed at the temporal instant of that tuple. As our approach, this work defines the data model, the types of update operations supported by the data sources considered as well as the queries. However, the focus of this work is on ensuring that the views defined in the chronicle algebra can be maintained incrementally, without having to store the chronicles.

Data Stream Management Systems. The semantic difference between relations and streams as well as the fact that the role of relations in the continuous queries execution semantics is not clearly defined were previously pointed out by Golab and Özsu [57]. This work proposes to model relations as look-up time-varying relations and defines
the following order for events and updates: any update on the relation at time \( t \) will affect only the stream events which arrive after \( t \). Our model on the other hand allows more general order definitions (e.g., for applications which do not have a time-based semantics).

Transactional concepts for streams have also arisen in the context of fault tolerance and high availability in distributed DSMSs [62, 63]. Previous work has defined some properties (e.g., repeatable / deterministic operators) and recovery methods, which can be used to implement low-overhead recovery protocols in case of failures.

**Stream Data Warehouses.** A more recent research direction tries to bring the real-time capabilities of the stream processing model to data warehousing [56, 86]. An example is the DataDepot project of the AT&T Labs. In this setting, in [55], different consistency levels are defined based on application requirements. The difference to our work is that the consistency notion this paper describes is different from the one generated by concurrent updates and failures. Moreover, we believe that our unified transactional model work can be naturally applied to stream data warehouses as well.

### 5.4.2 Real World Systems

In this section, we present the result of our analysis of a representative set of state-of-the-art DSMSs. These systems were chosen because of the heterogeneity they expose through the execution models, as pointed out in Chapter 2. As previously noted, for most of these systems, no explicit transactional properties were defined; therefore, the content we present here is based on our interpretation of the behavior, after having experimented with these systems, having read the provided documentation, and/ or having discussed with the authors. A summary of this analysis is presented in Table 5.1.

#### 5.4.2.1 Systems Detailed Analysis

When dealing with multiple input sources, the streaming systems offer the application developer a set of primitives which s/he can use to define isolation units as well as ordering among them. These primitives are different from system to system and, for most of the studied systems, embedded in the processing model and/ or operator semantics.

---

\(^3\)We name *processing model* the execution model of the system plus any other execution rules that the streaming engine defines.
<table>
<thead>
<tr>
<th>DSMS</th>
<th>Transactions</th>
<th>Synchronization Primitives</th>
<th>Producible Histories</th>
</tr>
</thead>
<tbody>
<tr>
<td>Coral8</td>
<td>Single event (remote inputs); Timeslice (explicit isolation and recovery unit)</td>
<td>Processing Model; Operator Semantics; Data Source Semantics</td>
<td>Conflict-equivalent to a serial history in which transactions involving events are increasingly ordered by the events’ timeslice values (only local inputs)</td>
</tr>
<tr>
<td>STREAM</td>
<td>Events sharing the same timestamp</td>
<td>Processing Model</td>
<td>Conflict-equivalent to a serial history in which transactions are increasingly ordered by the events’ timestamps</td>
</tr>
<tr>
<td>StreamBase</td>
<td>Single event; Group of events</td>
<td>Processing Model; Operator Semantics; Data Source Semantics</td>
<td>Conflict-equivalent to a serial history in which transactions involving events appear in the order of the events arrival in a stream</td>
</tr>
<tr>
<td>StreamInsight</td>
<td>Single event (remote inputs); Sequences of events between consecutive CTIs</td>
<td>Processing Model; Operator Semantics</td>
<td>Conflict-equivalent to a serial history in which transactions involving events are ordered by the CTI events values (only local inputs)</td>
</tr>
<tr>
<td>TruSQL</td>
<td>Window of events (explicit isolation and recovery unit)</td>
<td>?</td>
<td>?</td>
</tr>
</tbody>
</table>

**Table 5.1: Transactional Properties of Data Stream Management Systems**

NOTE: Not enough information about the execution model of TruSQL was available to us in order to emit a statement.
Chapter 5. Transactional Stream Processing

The execution model (what is named Tick in Section 2.4.0.4) defines the unit of execution for some systems, basically what can be viewed as a transaction (second column of Table 5.1). Another aspect that the execution model implies is the order among these units: e.g., time-driven systems process events in the increasing order of their timestamps. Operators also enforce order: the join operator in CEDR [26] synchronizes input streams based on the timestamps (validity intervals) of the events.

Furthermore, most of DSMSs today allow access to non-streaming data sources. For example, when dealing with remotely stored inputs like tables in DBMSs, the application developer has two options: one option is to use an adapter to transform the table into a locally recognized type of data source (e.g., stream) and, in combination with other ordering primitives, to implement the desired isolation units. A second option is to execute a retrieval operation on the stored input for each new event in a stream. In this case, besides restricting the isolation unit to a single event, some engines impose further restrictions on the streaming part: e.g., in Coral8 [1], only a stream (i.e., no local tables etc.) is allowed as input in a join with a table located in a relational database.

Next, we present a detailed description of each engine’s methods to define isolation units, ordering and recovery from failures. Whenever possible, we will present the behavior in terms of our proposed unified transactional model.

**Coral8** [1] defines explicit transactional properties for continuous query processing. That is, it specifies that the minimum recovery unit in case of a failure is the timeslice and although timeslices may be processed concurrently, it appears that each timeslice is executed separately and the order between them is preserved. A timeslice represents a row or a sequence of rows having the same timestamp and arriving in the same stream or window. Moreover, the timeslice is different from the engine’s execution model unit which in Coral8’s case is the batch (Section 2.5).

Coral8 can have as query data sources streams, windows or remote sources (like relational tables located in remote database systems). A window allows table-like operations but the events they contain can expire automatically, when a certain condition defined on the window is met. Moreover, a window, as opposed to a table also contains a notion of time through the events it contains.

If all events with the same timeslice are made part of the same transaction, a history generated by Coral8 is conflict-equivalent to a serial history in which transactions are increasingly ordered by their timestamp values (assuming that a transaction generated by events is assigned for example the timestamp of the corresponding timeslice). Moreover, as defined by the Coral8’s processing model, a transaction on a window having
the same timestamp as a transaction on a stream is executed before the transaction on the stream.

If some events come out-of-order (e.g., some events with timestamp $t-1$ come after an event with timestamp $t$ has been received), they are discarded. The transactions generated by these events can be modeled as aborted transactions (which are never retried) in the unified transactional model.

When joining a stream and a table located in a DBMS, Coral8 offers two options: either the isolation unit is a single event (this way, serializability is guaranteed), or an adapter which transforms the content of the database and its future updates into a stream can be written by the user. In the former case, the stream (i.e., no window) is the only accepted data source which can be joined with the remote input.

For dealing with failures, Coral8 offers mechanisms like State Persistence and Guaranteed Delivery. State Persistence basically offers the possibility to checkpoint the processing state periodically. The minimum unit is the timeslice. Nevertheless, it does not guarantee that events are not lost in case of failures. Guaranteed Delivery makes sure that an event is received by its destination at least once. In the unified model terms, Guaranteed Delivery may violate the Durability property by possibly duplicating events.

STREAM [22] accepts streams and time-varying relations as inputs and treats them uniformly. A time-varying relation is a relation in the traditional sense, but, in addition, it contains a notion of time. That is, a time-varying relation $R$ represents a mapping from a time domain $T$ to a finite but unbounded bag of tuples belonging to the schema of $R$.

As described in [22], the STREAM engine’s processing model is time-driven: time advances to $t$ from $t-1$ when all the events with timestamp $t-1$ have been processed.

This behavior translates to a transactional model in which all events with the same timestamp belong to the same transaction regardless of whether they arrive on the same stream or not, which is a difference to Coral8. In this context, a history generated by STREAM is conflict-equivalent to a serial history in which transactions are increasingly ordered according to the generating events’ timestamps.

StreamBase As explained in Section 2.5, StreamBase is a tuple-driven system. In short, this translates to: a continuous query is (re)executed whenever a new event arrives in one of the query’s streaming inputs.

One guarantee that StreamBase provides is that the order of arrival in a streaming input is the exact order of processing. Nevertheless, there is no predefined inter-stream
ordering: if two events are enqueued in two inputs, e.g., in Input1 and then in Input2 in quick succession, it is still possible that the event in Input2 is processed before the one in Input1. Therefore, if an application requires a certain ordering for its input streaming events, the developer should make sure that they are placed on the same stream, or that s/he uses specialized operators to order them (e.g., the operator which merges two streams).

StreamBase supports both streams and tables as inputs. The tables can be defined locally in the engine and updated through new events arriving in a stream. StreamBase also provides a lock operator so the application developer can do table-level locking. The lock key is an arbitrary expression and it is the responsibility of the programmer to ensure the desired synchronization.

Another type of data source that StreamBase accepts is represented by tables located in relational databases. For example, a stream can be joined with a table residing in a relational database system, in which case for each new event arriving in the stream, a new database lookup is executed.

In terms of the unified transactional model, in StreamBase, each event defines its own transaction (multiple events can be also logically grouped in transactions by using lock/unlock operators to guard access to tables). In this context, a history generated by a StreamBase engine is conflict-equivalent to a serial history in which transactions generated by events appear in the events’ order of arrival in a stream.

StreamBase also offers high availability features to enable fast recovery from failures.

StreamInsight [4] models streams as changing relations [26]. Each event is composed of a payload (the tuple value) plus application timestamp attributes defining an event’s validity interval. StreamInsight reacts whenever it reaches a special event, called CTI (Current Time Increment). The CTIs allow the engine to advance the time by specifying that all events with a timestamp smaller than the CTI have arrived. When reaching a CTI, a query is notified that the results corresponding to that part of the stream (composed of all the events with timestamps smaller than the CTI) will not change anymore and can be confidently committed as output.

When a table located in an RDBMS is involved in a join with a stream in StreamInsight, there are two options: either for each event in the stream, a table lookup is executed, or the content of the database table is transformed into a stream by specifying a validity interval for each of the rows. Further updates on the tables will change the existing validity intervals or add new events. By assigning timestamps to rows and updates, StreamInsight can order events and operations.
Additionally, StreamInsight offers as synchronization primitive, the semantics of operators. For example, the join operator matches events from different input streams which have overlapping validity intervals.

Given the previous observations, if we consider all the sequences of events between two CTIs as being part of the same transaction, a history generated in the StreamInsight engine is conflict-equivalent to a serial history in which transactions generated by events arriving in a stream between two CTIs (assuming they are assigned the timestamp of the corresponding CTI) are ordered by the CTI values.

TruSQL [13] is a commercial engine which extends a relational database system. One important area of reuse is the transactional system, which the authors extend to continuous processing [50, 72]. In this respect, we follow the same approach: streaming and stored data are not intrinsically different and the traditional transactional model can be reused. One difference to our model is that continuous queries are defined as long-running transactions. Moreover, the authors propose the window as the isolation unit for the interaction of streams and relations [40] which also represents the durability unit for stream archival. A window becomes therefore a sub-transaction and the execution behavior, similar to sagas [51]. By defining the unified transactional model, we take a step further and formally define the interactions, while allowing the definition of more flexible transaction boundaries.

5.4.2.2 Summary

The reader can observe that what defines a transaction differs from one system to another. The synchronization primitives are usually embedded in a given system’s processing model as well as in its operators’ semantics. Moreover, the processing models expose synchronization rules which are different from system to system.

In this context, it is hard to implement correct streaming applications. In addition, because of the tight dependence of the synchronization primitives to the processing models of the engines, very often, a change in the isolation properties requires the modification of an entire query plan, or it may even be impossible to implement.

The studied systems also offer mechanisms for recovery from failures, but it is not always clear how these mechanisms relate to the defined isolation units.

By reusing the traditional transactional model, our model defines a clean semantics for

---

\[\text{4} \text{The information we found about the TruSQL was not sufficient to make more statements about its behavior}\]
concurrent access and recovery from failures not bound to any specific model. As it is general, it supports all the histories generated by the analyzed streaming engines, thereby allowing a superset. In addition, it offers the application developer the possibility to arbitrarily group events and/or one-time/continuous queries into transactions, thereby permitting more control and flexibility to make a choice in the wide execution space.

5.5 Implementation Considerations

In this section, we present the implementation of the transaction manager (TM) we built for SMS, the general-purpose storage manager for Data Stream Management Systems presented in Chapter 4.

5.5.1 Transaction Manager Implementation

The job of a transaction manager is to both order the storage access operations it receives from the streaming system in order to obtain correct histories as well as to provide rollback execution for aborted transactions.

Just like in the traditional case, we designed the transaction manager as an additional component between the access and the storage layers, as in Figure 5.3.

We implemented the SS2PL (Strong 2-Phase Locking) [95] protocol in our TM. SS2PL is a locking protocol that defines an acquiring phase in which a transaction is only allowed to acquire locks (for the resources it is about to access). These locks are

![Figure 5.3: SMS with Transaction Manager Architecture](image-url)
all released after the transaction has ended (in the second phase of the protocol, the releasing phase), as opposed to other 2PL variants which allow early releases (a lock can be released even if not all the operations in a transaction are executed).

SS2PL is the protocol of choice for many general-purpose database systems, because of its desirable properties: recoverability and commitment ordering. The former is a property which specifies that, by using this protocol, the resources modified by an aborted transaction can be brought back to the states before the transaction started. The latter is very useful in a heterogeneous environment: if each system on which transactions are executed implements SS2PL, then the resulting global histories are serializable, a property required by our correctness criterion as specified in Section 5.3. In addition, global serializability is provided without explicit measures from the transaction manager [95].

For the lock management part of the TM, we adopt a simplified version of the design presented in [95]. Two types of data structures were used to implement the lock manager: RCB (Resource Control Block) and TCB (Transaction Control Block).

An RCB is defined for each resource (data item) being concurrently accessed by transactions. When a lock request for a certain resource is received, there has to be a way to check whether another transaction already holds a conflicting lock on this resource. The RCB offers an efficient way to execute this check operation.

Figure 5.4: Transaction Manager Data Structures
A hash table maps each resource object to its corresponding RCB. As can be observed in Figure 5.4, an RCB contains the ID of the resource it represents, the lock status of the resource (whether it is locked or not and if it is, in which mode: either shared or exclusive), how many transactions are waiting for a lock on this resource and the resource type (whether it is a stream, a relation etc.).

The field depicting the resource type is important for the implementation of the Commit operations, but in the same time useful for optimizations, as will be explained in Section 5.5.4. The RCB also contains a field (named “Order Info” in the figure) which is used to implement the desired ordering among transactions. For example, if histories have to obey the CSAO criterion (Section 5.3.4), the order information models the arrival ordering of the operations on a resource.

The TCB (Transaction Control Block) maintains a list of the resources (RCBs) a transaction has accessed and in which mode. The reason is to efficiently lookup and release the resources accessed by this transaction at termination time (commit or abort). The TCB also contains two fields representing the id of the transaction and its state (active, committed, aborted).

### 5.5.2 Transaction Manager Interface

The transaction manager’s interface exposes five methods:

- **startTransaction()** Whenever a new transaction is started, this method has to be called before the execution of any operation. Internally, the method creates a new transaction object which will be assigned an id and added to list of active transactions. Each transaction object (Transaction Control Block) maintains a list of the resources it locks and in which mode (shared or exclusive).

- **write(resource R, Transaction TID)** When calling this method, a transaction attempts to access the resource given as parameter in write mode. First, the RCB corresponding to resource R is obtained. Then, the TM has to check whether this transaction should be allowed (at this point) to access the resource or to be put on hold until the access conditions are met: the resource is not locked by another transaction (check the lock status of the RCB) and the defined ordering on the transactions, if any, is not violated by the execution of the current operation (check the order information of the RCB). If the request permission is granted, the transaction sets an exclusive lock on the resource (if not already holding it), the RCB of the resource is updated and the transaction is allowed to execute the update. Otherwise, the operation is put on hold. If an application...
failure is encountered, the transaction is aborted and a rollback procedure is executed (details in Section 5.5.3).

**read**(resource R, transaction TID) A transaction TID attempts to read a resource. First the RCB of this resource is obtained and the TM checks its lock status. If the RCB is exclusively locked, the request is put on hold until the resource is freed. Otherwise, if executing the read operation still maintains the defined ordering of transactions, then the transaction obtains a shared lock on the store and is allowed to access it (of course, multiple readers are allowed to access the store in the same time).

**commit**(transaction TID) This method is called by a transaction when it requests a commit operation. At this point all the locks held by this transaction are released (the corresponding RCBs are updated) and its changes are made persistent (depending on the type of resource this actions means making all the writes persistent, discarding rollback information etc.). As the resources accessed by this transaction are released, the TM can proceed to schedule the next operations.

**abort**(transaction TID) The engine specifies that the given transaction should be aborted. The modifications made by this transaction are undone through a rollback operation and all the locks it acquired are released.

When a resource R is located in another system (a DBMS for example), read and write requests are packaged (the system’s interface is used to specify which operations belong to the same transaction) and then forwarded to that respective system. It is then the responsibility of the local transaction manager to enforce local conflict serializability. As we discussed before, if a protocol which offers commitment ordering (like for example SS2PL) is used by each of the systems containing data sources, the TM does not need to implement any other logic, as global conflict-serializability is guaranteed in this case [95].

The abort and commit requests are handled in the same way: they are forwarded to the systems using the interfaces provided.

### 5.5.3 Recoverability and Recovery

Our model also allows to define correct executions in the presence of failures. Recoverability is a property of a concurrency control algorithm which defines whether it generates recoverable schedules or not. A recoverable schedule is one in which the effects of an aborted transactions can be undone and the system can be brought back to the state before this transaction has occurred.
SS2PL (Strong 2-Phase Locking) is a recoverable protocol and as a result, our TM implementation allows correct recovery from failures.

In our TM implementation, for each write operation, the change performed is logged: the modified resource plus the applied function. When a transaction is aborted, the following steps are taken: for each write operation belonging to the aborted transaction an inverse operation is executed (e.g., if the write operation added value 10 to a column in a table then the inverse operation will deduct 10). The resources which were only read as part of this transaction are not affected by the abort. After this phase, all the locks held by the transaction are released. The SS2PL protocol is recoverable because the locks are only released after the termination of a transaction and therefore no other transaction will read a dirty value.

5.5.4 Performance Optimization

Just like the Linear Road Benchmark (Section 3.7), many streaming applications have response time constraints. As a result, the performance of the transaction manager is important for the applications to meet these requirements. Some optimizations are possible because of the special properties that the streams expose.

5.5.4.1 Increased Parallelism

The page model is general enough to describe many implementation related issues. Nevertheless, its major drawback is that it does not take into account the semantics of the data access on the sources. That is, one could use the access pattern of the queries on the data to implement different locking granularities (a tuple, a group of tuples etc.) and therefore reduce contention and increase the execution parallelism.

For example, the state of the art in relational database systems is record-level locking in combination with index locking [95].

Our proposal is to use the semantics of the data sources to define different lock granularities and as such, aim at increasing the access performance on data items.

The write operations on a stream can be modeled as appends. The consequence is that read operations will practically not interfere with writes. As a result, to reduce lock contention, different lock granularities can be defined: e.g., one could change the scope of the read and write operations from the whole stream to individual events. Another solution is to use ranges: events having ids in a certain range a locked for reading or
writing (similar to predicate locking [95]).

5.5.4.2 Low-overhead Recovery

Recovery can be a time consuming process as all the changes made by aborted transactions have to be undone and the system has to be brought back to the state before the failure.

By using the semantics of the data sources, the rollback process can be optimized in some cases: when the data sources are append-only, inconsistencies from aborted transactions can only occur from the incomplete execution of the last active transaction. As a result, bringing the data source to the state before the transaction started merely requires undoing a number of append operations.

Another issue is related to temporary state maintenance. In order to meet the low latency requirements of streaming applications, the streaming engines employ a mechanism which is called incremental processing: that is, instead of recomputing the results from scratch (for the whole input from the arrival of the first data item) with each continuous query (re)execution, intermediate states are used to maintain partial results. For example, a continuous query computing the average value for an input stream of temperature readings, maintains a running sum and a running number of readings. When a new temperature reading arrives to the system, the sum is updated to reflect the new temperature reading and the number of readings is incremented.

In the transactional model, when a failure occurs, the intermediate state is lost and as such, the recovery process would have to recreate it, an operation which may take unacceptably long. In order to account for this optimization, the intermediate state may be exposed as a data source and included in the execution space. As such, it takes part in the rollback process and does not have to be recreated each time the system recovers.

5.5.5 Transaction Definition

One way to define transactions on streams (which we chose for our implementation) is to use special events similar to punctuations [93] (marks which indicate the end of a sequence) to specify the boundaries. That is, there is one event for each of the important actions: begin, commit or abort a transaction. When the query encounters a punctuation event, it calls the corresponding method (startTransaction, commit etc.)
exported by the TM.

If the transactions are not composed of sequences of events (e.g., events with the same value for a certain attribute) other methods can be used: e.g., add a special attribute to maintain the id specifying the transaction that a certain event belongs to.

Exploring the right interface to define transaction boundaries over streamed and stored data sources is an avenue for future work.

5.6 Experimental Setup

In this section we present the details of our experimental setup: the benchmark description, the implementation details, the hardware on which we ran the experiments as well as the metrics used.

The goal of our performance study is two-fold:

- Show that even with the overhead generated by the presence of a transaction manager we can achieve an acceptable performance, i.e., very close to the one obtained with an implementation with specialized synchronization methods (the kind that other streaming systems use).

- Show that a streaming application implementation can benefit from the presence of a transaction manager: the correctness requirements of the benchmark can be easily translated into isolation properties and implemented with very little effort only by specifying transaction boundaries (which can be easily modified). Moreover, the application developer does not have to deal with failures as this task is successfully handled in the transaction manager.

5.6.1 The Linear Road Benchmark

The Linear Road Benchmark simulates a network of highways and processes traffic information. The system is composed of a set of highways, divided into segments. The cars traveling on these provide location information every 30 seconds in the form of events and based on the traffic statistics (number of cars, speed) and accidents occurrences, variable tolling is provided. The accident and toll information are provided to the drivers in the form of notification. The drivers also have the possibility to request their balance of assessed tolls as well as historical information on their toll expenses.
over the previous 10 days. A more detailed description of the benchmark is given in Section 3.7.

An engine which implements the Linear Road Benchmark correctly has to generate the exact number of alerts and correct alert values, which are verified using a validator [3].

For the experiments whose results we present in the next section we used the Linear Road Benchmark implementation in MXQuery, presented in Chapter 3.

As explained in Chapter 4, one of MXQuery’s design decisions was the separation of query processing from storage management. MXQuery uses SMS as its underlying storage manager. We created two versions of the Linear Road implementation in MXQuery (ADHOC and TM) which will be presented next.

5.6.2 The ADHOC Implementation

This Linear Road implementation uses the version of storage manager presented in Chapter 4 with no transactional capabilities, but which contains built-in synchronization methods. We implemented two variants: ADHOC SERIALIZABLE, which orders the operations on the stores so that the execution behaves like a serializable one and ADHOC NOT SERIALIZABLE. In the non-serializable version, the only restriction is that the writers have exclusive access to the object (i.e., store) they modify.

This implementation contains a set of queries connected by stores as shown in Figure
(the rectangles represent queries, while the cylinders, the stores). The arrows express the data flow between the different queries. It is important to also note that as a result of the benchmark’s requirements, the stores in this implementation expose high heterogeneity: there are streams (e.g., containing car position reports), in-memory relations (the BALANCE store), a static relation stored in database system (HISTORICAL TOLLS) as well as files (e.g., for the results of the queries).

In the studied benchmark, two queries are particularly challenging because of their requirements: Accident Notification and Toll Notification.

The first query generates a notification for each driver who reports a position from a new segment in the vicinity of an accident. The query description specifies that alerts should be sent for all active accidents up to and including the minute before the car position report. As the information about the accidents which happened in the past minute may not be available when a certain car crosses a segment (due to the fact that the events come on different processing paths with different speeds), we need to keep the request waiting until the most recent accident data is available. This logic is implemented through built-in synchronization methods (the Synchronization parameter described in Section 4.3.2) in the stores themselves: a read operation is blocked until a predicate is evaluated to true.

In the second query, for computing the toll of a specific segment on a highway, the traffic statistics of that segment are used. Statistics are obtained every minute from analyzing a window with traffic information (number of cars, speed etc.) created for the past five minutes. Again, if the tolls computed for a certain minute are not yet written in the TOLLS store, the Toll Notification query is delayed until that data is available.

The ADHOC implementation requires that the developer has a good understanding of both the application semantics as well as the synchronization primitives. Next, we show how the requirements of the benchmark are met by defining transactions and order among them.

### 5.6.3 The TM implementation

In this section we present the implementation version in which the storage manager, SMS, is enhanced with the integration of a transaction manager (TM), which generates histories obeying the Conflict-Serializable with Arrival Ordering correctness criterion (Section 5.3.4). The details of this implementation are provided in Section 5.5.

The Toll Notification query is triggered for every position report sent by a car entering
a new segment (each translated to a write operation on the CARS SEGMENT store). As required by the benchmark, we would like that none of the notifications is lost and that the correct toll value is charged to each car. For that, we group all the operations of a one-time Toll Notification query execution in the same transaction, which will then be composed of four operations: read the car position report, \( r(\text{CARS SEGMENT CHANGE}) \), extract the toll for that car, \( r(\text{TOLLS}) \), update the driver's balance with the new assessed toll, \( w(\text{BALANCE}) \) and generate the toll alert, \( w(\text{TOLL ALERTS}) \). After the last operation is executed, the transaction is committed.

Moreover, all the toll values computed for a certain minute are written to the TOLL store as part of the same transaction, whose structure will then be: \( r(\text{POSITION REPORTS}) \) to read the position reports, followed by looking up the accidents store \( r(\text{ACCIDENTS}) \) and then writing the toll values \( w(\text{TOLLS}) \). If the execution of these transactions is serializable, we make sure that all the toll values read by the Toll Notification query are a result of the most recent computation and that there is no mix of old and new values.

We do the same for the Accident Notification query and group the following operations in a transaction: \( r(\text{CARS SEGMENT CHANGE}) \) to get the most recent car position report, followed by \( r(\text{ACCIDENTS}) \) to select all the accidents in the recently entered segment and possibly \( w(\text{ACCIDENT ALERTS}) \) if a notification needs to be sent. Again, all the accident events detected during a minute are made part of the same transaction: \( \{r(\text{POSITION REPORTS}) w(\text{ACCIDENTS})\} \).

For the Balance Request query, a transaction is composed of a read on the INPUT store to obtain the most recent car position report, \( r(\text{INPUT}) \), followed by a lookup on the BALANCE store to retrieve the driver's balance, \( r(\text{BALANCE}) \) and the balance report, \( w(\text{BALANCE REPORTS}) \). While the first two queries require a strict ordering of the transactions, this third query allows a more relaxed ordering: a driver can be notified of her most recent or the one-minute old balance of assessed tolls. This property accepts more schedules, permitting the Balance Request query to lower latency.

All other transactions have a simple structure, basically a read on an input store followed by a write on an output store.

### 5.6.4 Experimental Tools

The experiments were run on 2 x Quad Core AMD Opteron 2376 machines with 2.3GHz processor and 16 GB of main memory. We ran Linear Road load \( L=4.0 \) to completion which means 3 hours of execution (we used JVM version 1.6.0_02 and set the maximum heap size to 3 GB). Load 4.0 was chosen because it is the highest one achieved by the
ADHOC implementation (Section 5.6.2) on this type of machine.

The experimental results are presented using four metrics: the maximum response time, the average response time (both measured in milliseconds), the percentage of alerts of the total number which had a response time over the benchmark’s limit of 5 seconds and the correctness of results.

The response time (latency) of the queries is measured as the difference between the time the generated result is written to the file (or committed in the case of transactions) and the time the request enters the system. We repeated each experiment 15 times and the maximum response time values are presented as an average of those runs, plus an error bar representing the standard deviation.

We chose to measure the latency of the Toll and Accident Notification queries because they are the ones on the processing paths with the highest load and the transactions they define are more complex.

The correctness of results is represented as the number of wrong (which did not have the values expected by the validator), duplicated or missing results.

5.7 Experimental Results

In this section we show the results of running the Linear Road implementation in the two setups presented in the previous section.

5.7.1 Performance

First, we measured the overhead of the transaction manager. We wanted to understand how much the presence of the TM (compared to the ADHOC implementation) and enforcing the serializability property (compared to non-serializable implementations) increase the latency of the results.

For that, we compared the maximum response times when running the TM and ADHOC setups in both the serializable and non-serializable versions. The results are presented in Figure 5.6 (the same trend can be observed when plotting the average response times like in Figure 5.7 but the maximum response time is a more accurate measure of performance in the transactional case).

Comparing the ADHOC NOT SERIALIZABLE and the TM NOT SERIALIZABLE setups shows the overhead generated as a result of maintaining transactions. As expected,
the presence of the TM increases the maximum response time, but the performance penalty is quite low. The overhead of providing serializability can be observed by comparing the TM NOT SERIALIZABLE and the TM SERIALIZABLE setups (as well as ADHOC NOT SERIALIZABLE against ADHOC SERIALIZABLE). The maximum latency is lower for the not serializable versions (about 250ms in average) as the read operations don’t wait for the correct data to be available, but rather they return results based on whatever is contained in the store at the time of execution.

Nevertheless, the differences between the ADHOC and the TM setups (the serializable versions) are small and the majority of the alerts have low response times, meeting the requirements of the benchmark. More specifically, in the TM SERIALIZABLE setup, in
average, 97% of the accident alerts and 97.5% of the toll alerts have response times lower than 5 seconds (Figure 5.8 shows the average percentage of alerts - out of the total number - which have response times over 5 seconds; the ADHOC NOT SERIALIZABLE implementation is not represented in the figure because the results showed no alerts with response time higher than 5 seconds in this setup).

5.7.2 Correctness of Results

Although it incurs some performance penalty, serializability is important for providing correct results. To get a better understanding of its importance we measured the correctness of the results. We randomly picked one run of the Linear Road Benchmark with the non-serializable implementation and compared the results with the expected ones (obtained with a serializable implementation). Results are presented in Table 5.2. As shown, the ADHOC NOT SERIALIZABLE setup misses 94 accident alerts (because of accident information not being available at the time of request), similar to the TM setup which misses 99 accident alerts. Moreover, 52'618 toll alerts are wrong when the ADHOC implementation does not provide serializability (because of the reasons explained in Section 5.6.3), while the TM NOT SERIALIZABLE setup generates 32'226 wrong toll alerts. Of course, the serializable implementations generate the correct number of accident alerts and the toll notifications have the expected values.
5.7. Experimental Results

<table>
<thead>
<tr>
<th>Setup</th>
<th>Accidents Notification (missing alerts)</th>
<th>Tolls Notification (wrong alerts)</th>
</tr>
</thead>
<tbody>
<tr>
<td>ADHOC NOT SERIALIZABLE</td>
<td>94</td>
<td>52’618</td>
</tr>
<tr>
<td>ADHOC SERIALIZABLE</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>TM NOT SERIALIZABLE</td>
<td>99</td>
<td>32’226</td>
</tr>
<tr>
<td>TM SERIALIZABLE</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Total Number of Alerts</td>
<td>142’433</td>
<td>9’115’887</td>
</tr>
</tbody>
</table>

Table 5.2: Results Correctness: Number of Wrong/Missing Alerts

5.7.3 Failure Handling

An important advantage of having a transaction manager is that the client application does not have to deal with failures. One of the features that a TM provides is automatic rollback (the objects modified by a transaction are brought back to the state before the transaction started), which is not possible when transaction support is missing. To illustrate this, we present the following scenario: suppose that because of faulty behavior, every 2 seconds, a position report (for a car which crosses a segment) is duplicated. As a result, duplicate toll and accident alerts are generated. Moreover, the balance for those cars will be wrong as the same toll will be charged twice. The problem is that, when detecting this error, the ADHOC implementation has no way to rollback the execution of the query generated by the duplicated event. And whereas the duplicated alerts may just not be published to the output, the BALANCE store would still contain erroneous values which will generate wrong alerts in the Balance Request query. More specifically, as can be observed in Table 5.3, comparing the Balance reports obtained when the input has duplicates (‘ADHOC SERIALIZABLE (duplicates)’ in the table) with the correct ones, we could observe that in average about 8’000 (of a total of 242’458) were wrong.

This is not a problem for the TM SERIALIZABLE setup: when detecting the duplicate, the current transaction is rolled back and the car balance is restored to the value it had before the transaction started. Moreover, this benefit comes with pretty much no penalty in performance as the results in Table 5.3 show: the maximum response time is practically not affected by rolling back 5’395 Toll Notification transactions (for the Accident Notification query, the duplicated events had no impact on the output, as it turned out that the events which were duplicated did not generate accident alerts).
<table>
<thead>
<tr>
<th>Query</th>
<th>Toll Notification</th>
<th>Balance Request</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Number of</td>
<td>Maximum</td>
</tr>
<tr>
<td></td>
<td>Duplicated</td>
<td>Response</td>
</tr>
<tr>
<td></td>
<td>Alerts</td>
<td>Time (ms)</td>
</tr>
<tr>
<td>ADHOC SERIALIZABLE</td>
<td>0</td>
<td>5’108</td>
</tr>
<tr>
<td>ADHOC SERIALIZABLE (duplicates)</td>
<td>5’395</td>
<td>4’954</td>
</tr>
<tr>
<td>TM SERIALIZABLE</td>
<td>0</td>
<td>5’902</td>
</tr>
<tr>
<td>TM SERIALIZABLE (duplicates)</td>
<td>0</td>
<td>6’048</td>
</tr>
</tbody>
</table>

Table 5.3: Failure Handling: Performance vs Correctness

5.7.4 Sensitivity to Transaction Size

In the next experiment, we measured the sensitivity of performance to transaction size. In our previous scenario, each one-time Toll Notification query execution was part of a distinct transaction. Nevertheless, multiple one-time Toll Notification query executions may be grouped in the same transaction and this way, possibly reducing the response time for some of the alerts. For example, we expect that grouping all the one-time query executions corresponding to car position reports having the same timestamp (i.e., the same second) will have that effect (the same can be done for the Accident Notification query). The intuition is that some query executions will then not pay the penalty of acquiring the necessary locks, especially for the transactions which are executed at minute boundaries (right after a commit of new toll values or accident information).

We also measured the performance of two more setups: we grouped all queries corresponding to events having timestamps in subsequent time intervals of 2 or 3 seconds, which resulted in approximately 1’700 and 2’500 events respectively (in average, 850 events in 1 second). This was simply done by moving the punctuations (special events which mark the end of a sequence of events) from after each second, to after every 2 or 3 seconds. Moreover, grouping the queries does not affect the correctness of results as long as the transaction size does not exceed one minute (all one-time query executions corresponding to events with timestamps corresponding to the same minute). We did not try other transaction sizes, as the maximum response time exceeded 7 seconds when defining transactions composed of one-time query executions corresponding to alerts arriving in sequences of 3 seconds.
5.7. Experimental Results

![Bar chart showing maximum response time for different transaction sizes and types of notification.](image)

**Figure 5.9:** *TM SERIALIZABLE: Sensitivity to Transaction Size (Maximum Response Time)*

The maximum response time results are presented in Figure 5.9 (for completeness, we present the results for average response time in Figure 5.10 and for alerts with response time higher than five seconds in Figure 5.11). As expected, the maximum response time increases with the size of the transaction. There seems to be no statistical difference in the maximum response time when defining the transactions at 1 second boundaries compared to having one transaction for each event (the reason is that the overhead incurred by acquiring locks was low to begin with). Nevertheless, this...

![Bar chart showing average response time for different transaction sizes and types of notification.](image)

**Figure 5.10:** *TM SERIALIZABLE: Sensitivity to Transaction Size (Average Response Time)*
Figure 5.11: TM SERIALIZABLE: Sensitivity to Transaction Size (Alerts over 5 seconds)

experiment shows the flexibility and ease of use of the transaction manager: the visibility of updates (events) is effortlessly modified by moving a punctuation event along the stream.

5.8 Summary

In this chapter, we present a unified transactional model for streaming applications over an arbitrary mix of streaming and stored data sources and show how conflict-serializability can be used as a criterion to define correct executions.

The idea behind our model is simple, yet powerful: we treat all data sources to a continuous query uniformly. That is, they are all sets of data items on which updates and queries can be executed. Moreover, we model events as updates on streaming data sources and continuous queries as sequences of one-time queries. As a result, events and operations can be flexibly grouped into transactions. By enforcing a certain order among these transactions, the developer can define the update visibility required by the streaming application.

In addition, the unified model relieves the application developer from the task of dealing with concurrency control and recovery from failures.

Our model is also general enough to express the implicit transactional behaviors of a set of real-world DSMSs, which are otherwise difficult to understand because of the high
heterogeneity among streaming systems and the fact that they hard-code transactional properties in the processing models and operator semantics.

To test our approach, we implemented a transaction manager as part of SMS (Chapter 4) and we ran experiments using the Linear Road Benchmark implementation in MX-Query (Chapter 3). The results prove that the unified model is expressive enough to meet the correctness requirements of the benchmark even in the presence of failures and all this with very modest performance overhead.
Data streams processing is a technology which allows timely processing of data that comes in streams. More recently, as a result of applying the streaming model to new application domains, the requirements imposed on Data Stream Management Systems evolved, including besides low-latency of query results, other features like correlations of live streams with stored data or executing analysis queries over large portions of streams.

In order to meet the continuously evolving requirements of streaming applications, clean, high-performance and flexible streaming system solutions are required.

In this dissertation we present SMS (Storage Manager for Streams), a general storage management framework for Data Stream Management Systems which offers specialized store implementations for improving the performance of continuous queries. Moreover, we introduce the design and implementation of a unified transactional model for (continuous) query execution over combinations of streaming and stored data sources in order to allow the easy development of correct and reliable streaming applications.

6.1 Summary of the Dissertation

The exposition in this dissertation begins with presenting the background and context of the work.

Due to the lack of a standard semantics for stream processing, different execution models have been proposed. In order to properly present the background of our work, we first state the definitions for the most important stream processing concepts we use as well as the assumptions made. Next, we present a general model for analyzing
the behavior of DSMSs (Chapter 2). This model allows a better understanding of the state-of-the-art in the stream processing domain and clearly positions our work.

We use MXQuery (Chapter 3) as the context for our research. MXQuery is a Java-based open-source XQuery engine extended with window functions for continuous processing of data streams. While implementing the Linear Road Benchmark in MXQuery, it became obvious how important the performance of main memory management is for the performance of highly data-intensive streaming applications with low-latency requirements.

An important contribution of this dissertation is to propose the decoupling of (continuous) query processing from the data storage management (Chapter 4), similar to the relational database systems architecture design. In addition, we recommend the use of operator access patterns to customize the implementation of different stores (operator states, queues which connect operators etc.). We then show that the ideas borrowed from the relational database systems can be successfully combined with techniques made possible by the special characteristics of streaming applications (e.g., the update patterns of the continuous query operators) to create a clean, general and optimized storage solution.

We also present in Chapter 4 the architecture and interface of SMS (Storage Manager for Streams), a general purpose storage manager for DSMSs which is build on the previously described design ideas.

The performance experiments using the Linear Road Benchmark show a clear reduction of the response times of the queries when the stores have implementations tailored to the access patterns of the operators as opposed to implementations with wrong assumptions about the access patterns or which do not consider them at all. In addition, a clean storage solution allows to easily implement well-known stream processing optimizations like intermediate results sharing as presented in Chapter 4.

Another problem we address is defining the semantics for correct continuous query execution over arbitrary combinations of streaming and stored data sources in the presence of concurrent access and failures. In this regard, the dissertation makes more contributions by proposing a unified transactional model in Chapter 5: we minimally extend the page model to support, besides the read and write operations on stored data sources, the events composing the streams and the continuous queries. As a result, transactions can be flexibly defined over events and queries. We then propose conflict-serializability and a more restricted version (Conflict-serializability with Arrival Ordering) as correctness criteria to define the correct interleaving of the operations composing these transactions.
6.2. Future Work

We test our model again with the Linear Road Benchmark implementation in MXQuery and a transaction manager built as part of our Storage Manager for Streams. The results show that the transaction manager guarantees correctness in case of concurrency and failures without sacrificing from performance (Chapter 5). In addition, the correctness specifications can be flexibly changed by, e.g., effortlessly modifying the transactions boundaries.

6.2 Future Work

In this dissertation we present a solution for data storage management in the stream processing world, but we also open the door to other interesting research questions. A first direction for future work refers to distributed stream processing. It would be interesting to understand the implications on storage management brought by the challenges of a distributed setting and as such, the role a distributed stream storage manager would play. One particular issue refers to obtaining “precise recovery” [62], a type of recovery for Distributed Data Stream Management Systems which guarantees the same query results in a faulty execution as in one with no failures. Precise recovery is similar to the guarantee on recovery after failures offered by the serializability property. As such, the unified model could be a good match for enforcing this type of recovery.

Also, it would be interesting to understand the benefits of applying the unified model to several other areas of stream processing. For example, out-of-order processing [72] allows the processing of streaming items which, e.g., do not obey the time-based ordering on the streams (within certain boundaries). More specifically, by relying on specific information on progress (e.g., all items with timestamps lower than $t$ have arrived) a streaming system can process inputs which do not arrive in increasing timestamp order. To help the systems process streams with a certain degree of “disorder”, the unified transactional model could be used. That is, transactions can be created to group events, and as such, to define the visible states of the streams (the ones which give the streaming systems guarantees on the order).
# List of Tables

<table>
<thead>
<tr>
<th>Table Number</th>
<th>Table Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>2.1</td>
<td>Tick Example</td>
<td>22</td>
</tr>
<tr>
<td>2.2</td>
<td>SECRET Parameters Values for Different DSMSs</td>
<td>24</td>
</tr>
<tr>
<td>4.1</td>
<td>SMS Tunable Parameters</td>
<td>66</td>
</tr>
<tr>
<td>4.2</td>
<td>Query Examples for Access Patterns</td>
<td>70</td>
</tr>
<tr>
<td>4.3</td>
<td>Store Update Patterns</td>
<td>71</td>
</tr>
<tr>
<td>4.4</td>
<td>Update Interfaces</td>
<td>77</td>
</tr>
<tr>
<td>4.5</td>
<td>Read Interfaces</td>
<td>78</td>
</tr>
<tr>
<td>4.6</td>
<td>Performance with Access Pattern Tuning</td>
<td>91</td>
</tr>
<tr>
<td>4.7</td>
<td>Performance with Sharing Store Data</td>
<td>94</td>
</tr>
<tr>
<td>5.1</td>
<td>Transactional Properties of Data Stream Management Systems</td>
<td>119</td>
</tr>
<tr>
<td>5.2</td>
<td>Results Correctness: Number of Wrong/Missing Alerts</td>
<td>137</td>
</tr>
<tr>
<td>5.3</td>
<td>Failure Handling: Performance vs Correctness</td>
<td>138</td>
</tr>
</tbody>
</table>
List of Figures

1.1 Typical Data Stream Management System Architecture .................................... 2
1.2 Schematic Road Map of the Dissertation Outlining Contributions ......................... 7

2.1 Sliding Windows of Example 2 ........................................................................ 13
2.2 SECRET of a Query Plan .................................................................................. 17
2.3 Scope of a Window .......................................................................................... 18
2.4 Tick Models .................................................................................................... 21

3.1 Annoying Authors: XQuery 1.0 ........................................................................ 30
3.2 Annoying Authors: Extended XQuery ............................................................... 30
3.3 Grammar of Extended FLWOR Expression ...................................................... 32
3.4 Window Types ................................................................................................ 34
3.5 Window Automaton .......................................................................................... 36
3.6 Example Plan (First Query of Section 3.4) ......................................................... 42
3.7 MXQuery Architecture ................................................................................... 42
3.8 Stream Buffer .................................................................................................. 45
3.9 Data Flow of the Linear Road Benchmark Implementation .............................. 53

4.1 Role Store Property ........................................................................................ 64
4.2 Store Operations .............................................................................................. 68
4.3 Shared Store .................................................................................................... 73
4.4 DSMS Architecture with SMS ......................................................................... 75
4.5 RANDOM Store Update Pattern Implementation .............................................. 80
List of Figures

4.6 FIFO Store Update Pattern Implementation ........................................ 82
4.7 Speed-up Read Operations ................................................................. 83
4.8 Linear Road Stores Implementation .................................................. 86
4.9 Baseline Implementation ................................................................. 88
4.10 Effect of Block Size on Performance ............................................... 90
4.11 Speed 0 Experiment ..................................................................... 92
4.12 Effect of Read Patterns on Performance ......................................... 93
4.13 Effect of Sharing Store Data Setup ................................................ 94

5.1 Example Scenario ................................................................. 100
5.2 Design Space Architecture .......................................................... 109
5.3 SMS with Transaction Manager Architecture .................................. 124
5.4 Transaction Manager Data Structures ........................................... 125
5.5 Linear Road Implementation .......................................................... 131
5.6 ADHOC vs TM Maximum Response Time ..................................... 135
5.7 ADHOC vs TM Average Response Time ....................................... 135
5.8 ADHOC vs TM Percentage of Alerts with Response Time Higher than 5 Seconds ............................................................. 136
5.9 TM SERIALIZABLE: Sensitivity to Transaction Size (Maximum Response Time) .................................................. 139
5.10 TM SERIALIZABLE: Sensitivity to Transaction Size (Average Response Time) .................................................. 139
5.11 TM SERIALIZABLE: Sensitivity to Transaction Size (Alerts over 5 seconds) .................................................. 140
Bibliography


[14] Uppsala University Linear Road implementation in SCSQ. http://user.it.uu.se/udbl/lr.html.


[46] Mary Fernandez et al. XQuery 1.0 and XPath 2.0 Data Model (XDM), 2006.


Curriculum Vitae: Irina Magdalena Botan

Affiliation During Doctoral Studies

November 2006 - May 2011
ETH Zurich
Research Assistant (Ph.D. Student)
Department of Computer Science, Systems Group

Education

September 2005 - July 2006
“Babes-Bolyai” University, Cluj-Napoca, Romania
M.Sc. from The Faculty of Mathematics and Informatics
Major: Distributed Systems in Internet and Intranet
GPA: 9.25/10.0
M.Sc. Thesis: “Ten in Ten”, research work at ETH Zurich

September 2000 - July 2005
University “Politehnica” of Bucharest, Romania
B.Sc. from The Faculty of Automatic Control and Computers
Major: Systems Software
GPA: 9.40/10.0
B.Sc. Thesis: “XML Partitioning in XL”, research work at ETH Zurich
September 1996 - July 2000
National College of Computer Science, “Traian Lalescu”, Hunedoara, Romania
High-school leaving examination grade: 9.83/10.0

Projects

November 2006 - May 2011
Storage Management for Streams
Ph.D. thesis topic
SMS is a general purpose storage management framework for Data Stream Management Systems, based on a clean, loosely-coupled, and flexible system design that also facilitates performance optimization.

August 2006 - May 2011
MXQuery
A low-footprint, extensible Open-Source XQuery Engine implemented in Java (www.mxquery.org).

May 2007 - September 2010
MaxStream
The goal of this project is the design and build a federated stream processing architecture that seamlessly integrates multiple autonomous and heterogeneous Stream Processing Engines (SPEs) with traditional databases, behind a common SQL-based declarative query interface and a common API.

May 2007 - August 2007
Compressing Data from A Wandering Distribution
The project was developed during an internship at IBM Almaden Research Center. The goal: to develop a method for online creation of dictionaries for data compression when the data values come from different probability distributions, while still allowing random access on the compressed data. As a result, an algorithm for discrete dictionary maintenance was developed.

September 2005 - July 2006
Ten in Ten
The project is intended as a baseline regarding the storage of data streams in a relational database system for use in event-based systems (explores index usage, garbage collection strategies, table creation strategies).
March 2005 - August 2005
XML Partitioning in XL
Part of the XL project, developed at ETH Zurich, the goal of the work is to increase the performance of the distributed storage of the XL system by reducing the congestion on the shared XL variables. The solution is to partition the variable content, XML data, by analyzing XL code access patterns.

Teaching Experience

Summer Semester 2010
“Introduction to Database Systems”
Department of Computer Science, ETH Zurich
Teaching Assistant

“Informatik I” lecture
Department of Information Technology and Electrical Engineering, ETH Zurich
Teaching Assistant

Summer Semester 2005
“Data Modeling and Databases” lecture
Department of Computer Science, ETH Zurich
Teaching Assistant

Professional Experience

May 2007 - August 2007
IBM Almaden Research Center, San Jose, CA, SUA
Internship in the Database Technology Institute
Involved in a project that investigates methods for extreme data compression while allowing queries to be executed directly on the compressed data.
September 2005 - October 2006
ETH Zurich, Switzerland
Department of Computer Science, Databases and Information Systems group
Research Assistant

July 2004 - February 2005
SOFTWIN SRL, Bucharest, Romania
Software Engineer in the Data Security Division, Windows Servers
Responsibilities: to develop new modules and to do research on new technologies to be included in the BitDefender project.

Publications


