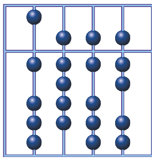


Technische Universität München
Fakultät für Informatik
Lehrstuhl III - Datenbanksysteme

Query Processing on Data Streams

Diplom-Informatiker Univ.
Bernhard Stegmaier

Vollständiger Abdruck der von der Fakultät für Informatik der Technischen Universität München zur Erlangung des akademischen Grades eines

Doktors der Naturwissenschaften (Dr. rer. nat.)

genehmigten Dissertation.

Vorsitzender: Univ.-Prof. Dr. Helmut Seidl

Prüfer der Dissertation:

1. Univ.-Prof. Alfons Kemper, Ph.D.
2. Univ.-Prof. Dr. Christoph Koch,
   Universität des Saarlandes, Saarbrücken

Die Dissertation wurde am 16.11.2005 bei der Technischen Universität München eingereicht und durch die Fakultät für Informatik am 19.06.2006 angenommen.

Abstract

Data stream processing is currently gaining importance due to the rapid increase in data volumes and developments in novel application areas like e-science, e-health, and e-business. In this thesis, we propose an architecture for a data stream management system and investigate methods for query processing on data streams in such systems.

In contrast to traditional database management systems (DBMSs), queries on data streams constitute continuous subscriptions for retrieving interesting data rather than one-time ad-hoc queries. To meet the challenges of the new "streaming" paradigm, we propose *StreamGlobe* as a *distributed data stream management system* for efficiently querying and processing XML data streams in the spirit of a traditional DBMS. Beyond processing XQuery subscriptions, StreamGlobe in particular addresses the problem of efficiently distributing data streams in Peer-to-Peer networks by means of *data stream sharing* to avoid network and peer congestion.

For the evaluation of XQuery subscriptions, StreamGlobe employs our novel streaming XQuery processor, *FluX*. FluX represents an extension of the XQuery language supporting event-based query processing and the conscious handling of main memory buffers to achieve a scalable execution of queries on data streams. XQueries are rewritten into the event-based FluX language by exploiting order constraints from the schema of a data stream to schedule event processors and to thus minimize the amount of buffering required for evaluating a query. Performance experiments prove the effectiveness of our approach.

StreamGlobe further allows the use of user-defined operators for enabling expressive query processing. We discuss the implementation of such operators using our novel class of *best-match join* operators as an example. These operators address the problem of finding best matching pairs of data objects in multi-dimensional spaces. Considering multiple dimensions leads to a partial order on the pairs of objects. Since partial orders naturally have more than one minimum, traditional approaches aiming at determining a single "best" pair most likely fail to produce satisfying results. In contrast, our best-match join computes the best matching pairs having a maximum similarity on each individual dimension. We assess the effectiveness of this approach by means of performance experiments.

Contents

1 Introduction **1**

1.1 Purpose of this Thesis 2

1.2 Outline of this Work 3

2 The *StreamGlobe* Data Stream Management System **5**

2.1 Motivation 6

2.2 The StreamGlobe Architecture 13

2.2.1 The Fundamentals: Open Grid Services Architecture 13

2.2.2 Network Topology 14

2.2.3 Peer Architecture 17

2.2.4 Client Interface 18

2.2.5 Metadata Management 19

2.3 Subscription Evaluation 20

2.3.1 Optimization Goals and Strategy 20

2.3.2 Optimization Algorithm 23

2.3.3 Query Execution Basics 27

2.4 Further Example Scenarios 29

2.5 Related Work 32

2.5.1 Peer-to-Peer Data Management 32

2.5.2 Data Stream Management and Processing 33

2.5.3 Multi-Query Optimization and Execution 34

2.5.4 Networking Aspects 35

2.5.5 Grid Computing and E-Science Applications 36

2.6 Discussion 37

3 The *FluX* Streaming XQuery Processor **39**

3.1 Motivation 40

3.2 Preliminaries 43

3.3 Efficient Checking of Schema Constraints 45

3.4 Query Language 48

3.4.1 An XQuery Fragment: XQuery$^-$ 49

3.4.2 Syntax and Semantics of FluX 51

3.4.3 Safe Queries 54

3.5 Translating XQuery$^-$ into FluX . . . 56
3.5.1 A Normal Form for XQuery$^-$ . . . 56
3.5.2 Rewriting Normalized XQuery$^-$ into FluX . . . 58
3.5.3 Examples . . . 60
3.6 Algebraic Optimization of XQuery$^-$ . . . 66
3.6.1 Rewrite Rules for Algebraic Optimization . . . 66
3.6.2 Examples . . . 70
3.7 Implementation . . . 75
3.7.1 The *XSAX Parser* . . . 76
3.7.2 Query Execution . . . 77
3.7.3 Buffer Management . . . 83
3.8 Extending the FluX Query Language . . . 88
3.8.1 Aggregate Functions . . . 88
3.8.2 Data Windows . . . 101
3.9 Performance Evaluation . . . 121
3.9.1 XSAX Parser . . . 122
3.9.2 Basic Performance Tests . . . 123
3.9.3 Pipelining Behavior and Buffer Allocation . . . 127
3.9.4 Extensions: Aggregate Functions and Data Windows . . . 129
3.10 Related Work . . . 131
3.11 Discussion . . . 134

4 The *Best-Match Join* 137
4.1 Motivation . . . 138
4.2 Definition of the Best-Match Join Variants . . . 141
4.2.1 Comparing Pairs Using Partial Orders . . . 141
4.2.2 The Best-Match Join Operators . . . 142
4.2.3 Constrained Best-Match Joins . . . 148
4.3 Evaluating Best-Match Joins on Data Streams . . . 150
4.3.1 Best-Match Joins and Data Streams . . . 150
4.3.2 The Window-Based Approach . . . 152
4.3.3 Exploiting Fuzzy Orders on Data Streams . . . 155
4.3.4 I/O-Scheduling Using the ϵ-Grid-Order . . . 157
4.3.5 Dealing with Time-Stamped Data Streams . . . 164
4.4 Performance Evaluation . . . 166
4.5 Related Work . . . 168
4.6 Discussion . . . 170

5 Conclusion 171

A Complete XQuery$^-$ Grammar 173

B Translating XQuery$^-$ into FluX: The Rewrite Algorithm 175

C FluX Benchmark Experiments **177**
C.1 Modified XMark Benchmark Queries 177
C.2 Additional Benchmark Queries 180
C.3 Benchmark Results 181

D Translating XQuery^{-} into FluX: The Rewrite System **185**

Bibliography **189**

List of Figures

2.1 Example Astrophysical Scenario . . . 8
2.2 Visualization of the ROSAT Photon Data . . . 11
2.3 Traditional Subscription Evaluation . . . 12
2.4 Optimized Subscription Evaluation . . . 12
2.5 StreamGlobe Architecture Overview . . . 13
2.6 Network Topology of the Extended Example Scenario . . . 16
2.7 Architecture Overview of Different Peers . . . 17
2.8 XML Representation of Query Plan . . . 26
2.9 Query Plan of the Introductory Example . . . 27
2.10 Integration of "RXJ-alert" . . . 30
2.11 Integration of "Bursts" . . . 30

3.1 Glushkov Automaton for $\rho = (a^*.b.c^*.(d|e^*).a^*)$ and $S = \{b\}$ . . . 47
3.2 XQuery$^-$ Grammar . . . 51
3.3 Normal Form Rewrite Rules . . . 56
3.4 Function rewrite(Variable $\$x$, Set$\langle\Sigma\rangle$ H, XQuery$^-\beta$) **returns** *FluXQuery* . 59
3.5 Algebraic Optimization Rewrite Rules . . . 67
3.6 Architecture of the FluX Query Engine . . . 75
3.7 Query Plan for Query XMP-Q1 . . . 81
3.8 Fragment of Query Plan for Example 3.5.5 . . . 83
3.9 Buffer Trees of Example 3.7.5. . . . 85
3.10 XQuery$^-$ Grammar Fragment for Handling Aggregate Functions . . . 89
3.11 Additional Normal Form Rewrite Rules for Extended XQuery$^-$ . . . 93
3.12 Fragment of Query Plan for Example 3.8.1.4 . . . 98
3.13 Fragment of Query Plan for Example 3.8.12 . . . 100
3.14 Element-Based and Time-Based Data Windows . . . 102
3.15 XQuery$^-$ Grammar Fragment for Handling Data Windows . . . 103
3.16 Extension of Function "rewrite" for Handling Data Windows . . . 111
3.17 Fragment of Query Plan for Example 3.8.22 . . . 116
3.18 Fragment of Query Plan for Example 3.8.23 . . . 119
3.19 XSAX Performance . . . 123
3.20 FluX Overall Performance: Execution Time . . . 124
3.21 FluX Overall Performance: Memory Consumption . . . 125

3.22 Comparison of FluX Optimization Variants: Execution Time 126
3.23 Comparison of FluX Optimization Variants: Memory Usage 127
3.24 Scalability 127
3.25 Throughput 127
3.26 Execution of Query 13 128
3.27 Execution of Query 8 129
3.28 Execution of Query 8b 129
3.29 Performance of FluX Extensions 129
3.30 Execution of Query W2 130
3.31 Join Optimization on the Example of Query 8 135

4.1 Examples of Best-Match Join Computations 144
4.2 Nested-Loops Left-Outer-BMJ 148
4.3 Constrained Left-Outer-BMJ: Initial Computation Step 153
4.4 Constrained Left-Outer-BMJ: Moving Data Windows 154
4.5 Constrained Left-Outer-BMJ: Updating Current Best Pairs 155
4.6 MatWin Algorithm 156
4.7 Pages in the Data Space 159
4.8 Pairs of Pages to be Processed 159
4.9 Gallop Mode: Thrashing 160
4.10 Crabstep Mode 160
4.11 MatWindows Algorithm 162
4.12 Filling Data Windows on Fuzzy Ordered Data Streams 163
4.13 Fixed Time Window Evaluation 165
4.14 Overall Performance of the MatWin Algorithm 167
4.15 Streaming Behavior of the MatWin Algorithm 168

B.1 Function rewrite(Variable $\$x$, Set$\langle\Sigma\rangle$ H, XQuery$^{-}\beta$) **returns** *FluXQuery* . 176

C.1 XMark Benchmark Results 182
C.2 Extensions Benchmark Results 183

D.1 Rewrite Rules for Translation of XQuery into FluX (Using a DTD) 186

Chapter 1

Introduction

Recent advances in research and technology, e. g., miscellaneous mobile devices being able to participate in ubiquitous (ad-hoc) networks, enable the gathering of huge amounts of data in various fields. In e-business, the advent of RFID-tags will produce a huge amount of relevant information to support and to improve business processes. In e-science, more sophisticated experiments and analysis methods, e. g., in the areas of high energy physics, astronomy, and life-sciences, will deliver more and more data that has to be evaluated.

All of these applications continuously produce data as *data streams*, e. g., by RFID-readers and various other sensor devices, by simulation experiments, and so on. In the following, we will denote these data stream providers as data sources. Data streams are usually delivered as long as their data sources are operational. Thus, the amount of delivered data is—in the extreme—infinite. The traditional approach of processing such data streams is to store some kind of a "snapshot" of the stream, which represents a certain amount of time over which data has been collected, in an appropriate data management system. Then, analysis tasks are carried out upon this set of (now persistent) data. Though this approach of storing data prior to processing it is still state-of-the-art, it will not be feasible considering the rapid growth of data volumes in the near future. Further, the streaming nature of the data opens up for novel application scenarios of *monitoring data streams* to be able to detect and to possibly react to interesting situations in real-time as the data streams in. Hence, *data stream management systems* (DSMSs) that are able to efficiently process data streams will inevitably be needed.

In traditional data management systems users issue ad-hoc queries against persistent data to get all results at once. In contrast, in a data stream management environment, users subscribe to interesting data by means of continuous queries which describe, transform, and process the data in which users are interested in. As the name suggests, these queries are continuously evaluated and qualifying results are subsequently delivered—again, as data streams. Such subscriptions provide both the ability to process and analyze data streams on-the-fly as well as to retrieve interesting data, e. g., to store it for off-line computations.

During the last decade, we have seen a substantial effort on turning the Internet into a global database management system for searching, querying, and retrieving persistent data [LKK+97]. Now, a major challenge is to build a distributed DSMS in the same

manner. Users should be able to join such an information retrieval network to publish their data and/or to subscribe to all the data they are interested in. Such a DSMS has to provide capabilities for performing expressive information retrieval and fusion tasks as well as for tailoring the retrieved information exactly to the users needs, e. g., for being viewed on devices ranging from powerful workstations to small mobile devices such as PDAs and cell phones. Although data management systems for persistent data and data stream management systems seem to face the same problem of efficiently querying information, DSMSs have to cope with several challenges stemming from the streaming nature of data streams [BBD+02]. One problem is, that data streams can only be read once, i. e., if a data object of a data stream is not stored for later usage, it cannot be obtained in the future anymore. Hence, novel algorithms accounting for the volatility of data streams and, at the same time, being parsimonious with resources are needed. Another problem is the fact that the result of a subscription (or a query) is also a data stream. On the one hand, this again depends on novel algorithms continually producing results—in contrast to traditional data management systems, where many operations are of a blocking nature. Conversely, this means that in a distributed DSMS results have to be transferred unceasingly through the network. Although network bandwidths will most probably continue to increase (or, at least, high bandwidths will become widely available), it has been observed time and again that the amount of information grows much faster than technology improves (or traffic costs decrease, respectively). Hence, an optimization goal of a DSMS must be to minimize the amount of network traffic by only transmitting relevant parts of data streams and to leverage previous work on routing data in computer networks to avoid network congestion.

Currently, most research addresses individual aspects of these challenges of a distributed data stream management system. Various novel techniques have been developed to enable streaming computation of miscellaneous operations known from persistent data management systems. There are also attempts to adopt data management systems for persistent data to the streaming model. However, to the best of our knowledge, no principled work exists on building a distributed data stream management system specifically optimizing for parsimonious handling of both computational and network resources.

1.1 Purpose of this Thesis

In this thesis, we address the following aspects of building a general-purpose data stream management system facing the challenges outlined above:

- We provide an architecture of a distributed DSMS optimizing for both network and computational resources.
- We show buffer-conscious optimization techniques for efficiently and scalably executing queries on data streams.
- We propose a novel (streaming enabled) operator for information retrieval and show how to efficiently execute it on data streams.

We propose *StreamGlobe* as a distributed data stream management system for efficiently querying and processing data streams. Since XML [W3C04a] is the preeminent data exchange format on the Internet, StreamGlobe exchanges information by means of XML data streams and employs XQuery [W3C05b] for the specification of subscriptions. To further ensure interoperability, StreamGlobe is built on top of Grid standards [FK04], which is a common platform for distributed systems—especially in the area of e-science applications. Finally, StreamGlobe employs Peer-to-Peer networking techniques for enabling users to flexibly join and leave the information retrieval network. Beyond processing XQuery subscriptions, StreamGlobe in particular addresses the problem of efficiently distributing data streams in Peer-to-Peer networks by means of *data stream sharing* to avoid network and peer congestion.

For the evaluation of XQuery subscriptions on XML data streams, we have developed our streaming XQuery processor, *FluX*. We introduce an extension of the XQuery language supporting event-based query processing and the conscious handling of main memory buffers. Purely event-based queries of this language can be executed on streaming XML data in a very direct way. XQueries are rewritten into the event-based FluX language by exploiting order constraints from a DTD to schedule event processors and to thus minimize the amount of buffering required for evaluating a query. Therefore, a scalable execution of queries on data streams is achieved.

To be able to carry out expressive information retrieval and analysis tasks, StreamGlobe enables the execution of user-defined operators for query processing. We discuss the execution of such operators using the example of our novel class of *best-match join* operators. These operators address the problem of finding best matching pairs of data objects in multi-dimensional spaces. Considering multiple dimensions leads to a partial order on the object pairs. Since partial orders naturally have more than one minimum, traditional approaches based on determining a single "best" pair, e. g., by means of rating functions, most likely fail to produce satisfying results. In contrast, our best-match join computes the best matching pairs having a maximum similarity on each individual dimension. To overcome its inherently blocking nature and to improve the quality of the results we employ constraints in combination with physical properties of the data streams to enable our pipelined best-match join algorithms.

1.2 Outline of this Work

The remainder of this thesis is organized as follows. In Chapter 2, we provide an overview of the architecture of our StreamGlobe data stream management system. We further sketch our ideas for optimizing subscription evaluation with respect to minimization of network traffic and peer load. Chapter 3 presents our FluX query language, optimization techniques for buffer-conscious query execution, and core concepts of the implementation of our FluX query engine used for subscription evaluation in StreamGlobe. In Chapter 4, the family of best-match join operators is formally defined and techniques for evaluating these operators on data streams are presented. Finally, Chapter 5 concludes this thesis.

Chapter 2

The *StreamGlobe* Data Stream Management System

Recent research and development efforts substantiate the increasing importance of processing data streams, not only in the context of sensor networks, but also in information retrieval networks. With the advent of various (possibly mobile) devices being able to participate in ubiquitous (wireless) networks, a major challenge is to develop data stream management systems for information retrieval in such networks. In this chapter, we present the architecture and core concepts of our *StreamGlobe* system, which is focused on meeting the challenges of efficiently querying and distributing data streams in an (possibly ad-hoc) network environment. StreamGlobe is based on a federation of heterogeneous peers ranging from small, possibly mobile devices to stationary servers. On this foundation, self-organizing network optimization and expressive in-network query processing capabilities enable powerful information processing and retrieval. Data streams in StreamGlobe are represented in XML and queried, i. e., subscribed, using XQuery.

Parts of this chapter have been presented at the *International Workshop on Data Management for Sensor Networks 2004* (held in conjunction with *VLDB 2004*) [SKK04], at the workshop *Dynamische Informationsfusion* (in conjunction with the *34. Jahrestagung der Gesellschaft für Informatik 2004*) [SK04], and have been published in [KSH+04]. A demonstration of StreamGlobe has been given at the *International Conference on Very Large Data Bases 2005* [KSKR05].

This chapter is organized as follows: Section 2.1 demonstrates the need for data stream management systems for distributed information retrieval networks and outlines the core concepts of StreamGlobe. In Section 2.2 an overview of the StreamGlobe system architecture is presented. In Section 2.3 the key concepts of optimization and query processing in StreamGlobe are discussed. Further application scenarios are presented in Section 2.4. Related work is addressed in Section 2.5. Section 2.6 concludes this chapter.

2.1 Motivation

In the past few years, two designs of distributed systems have gained much attention in the research community: *Grid Computing* and *Peer-to-Peer networks*.

With the increasing amount of data to be processed, computing more and more addresses collaboration, data sharing, cycle sharing, and other methods of interaction involving distributed resources. Embraced by the term *distributed computing* in the early seventies, first attempts have been made to harness unused CPU cycles by sharing them in a network. With the advent of the Internet in the nineties, distributed computing has become interesting on a global level, e. g., in the SETI@Home project—which is still considered as a proof for the effectiveness of distributed computing. Todays developers typically target a specific platform, such as Windows, Unix, Java, .NET, and others, which provides a hosting environment for running applications. The heterogeneity of the available platforms is a major drawback with respect to interoperability for the goal of distributed computing. Especially work within the community of large-scale scientific research has led to the development of *Grid technologies* [FK04], which have been widely adopted in scientific and technical distributed computing since then, to overcome this drawback and to provide platform-independent standards. In particular, the open source *Globus Toolkit* [Glo04] has emerged as the de facto standard for the construction of Grid systems. Grid technologies, and the Globus Toolkit in particular, have evolved to an *Open Grid Services Architecture (OGSA)* [FKNT02] in which a Grid provides an extensible set of services, which are built on concepts of both the Grid community and established Web-Service standards. Therefore, OGSA defines a uniform exposed service semantics allowing for an easy aggregation and composition of services provided in the Grid. As the World Wide Web has began as a network for scientific collaboration and later has been adopted for e-business, we are likely to see a similar evolution in the field of Grid computing.

The *Peer-to-Peer (P2P) network architecture* has gained much attention both in the media and in the research community in the past few years. This is due to the stunning success of P2P file-sharing networks like Napster and Gnutella in the special application domain of exchanging (music-) files. Basically, in a P2P network all participants, denoted as *peers*, have the same capabilities and may act both as client and server. Further, peers are allowed to join or leave the network at any time. The goal is a flexible and decentralized community of peers pooling their resources to benefit everyone. However, the well-known file-sharing systems, which have made P2P systems popular, ignore a crucial aspect: the semantics of the data. A very challenging issue for database researchers is how data management can be applied to P2P and what we can learn from the P2P area. Hence, the P2P paradigm has been picked up for the realization of dynamic and adaptive distributed information retrieval systems in the research community. Various techniques, topologies, and prototype systems have been developed to efficiently query, locate, and process information available in such P2P networks, with the goal of creating *peer database management systems (PDMSs)* in the spirit of traditional (distributed) DBMSs[1]. Another

[1]See Section 2.5 for a detailed overview.

indicator for the growing importance of P2P systems is the fact that in commercial systems a shift towards the P2P collaboration paradigm also takes place.

Altogether, the combination of Grid computing and P2P techniques provides a good basis for building flexible and adaptive information retrieval networks. To realize such networks, efficient, adaptive, and self-organizing *data stream management systems* are needed to cope with the huge amount of data that will have to be processed in the e-business and e-science application scenarios which we outlined in the introduction. On the one hand, peers can register themselves as data sources in such a network to provide their data as a continuous data stream. Conversely, peers can subscribe to interesting data in the network by means of queries. Now, a major challenge is to not only match subscriptions to available data and deliver this data, but to exactly tailor the data to a specific query, which represents the specification of a subscription, and to deliver only the specific part of the data a user is really interested in (or has subscribed to, respectively). Furthermore, the aspect of routing data streams in the network is of great importance. In traditional PDMSs various techniques for locating data matching a query have been developed. Since these queries are only one-time ad-hoc queries, no special efforts for delivering the results are needed. This problem facing data streams has to be addressed, since a subscription will establish a continuous data flow in the network, which naturally has to be routed in a way that reduces network traffic.

We pursue these goals with our *StreamGlobe* system, which is based on the techniques of its predecessor ObjectGlobe [BKK+01] addressing distributed query processing on persistent data. The main contributions of StreamGlobe are as follows: StreamGlobe enables peers to register and efficiently query data streams by means of subscriptions. Furthermore, query processing is pushed adaptively into the network to optimize the data flow. This is achieved by *data stream sharing*, which denotes the re-usage of data streams which are computed to fulfill other subscriptions (or parts of other subscriptions) and are already available in the network. In addition, common parts of various subscriptions are computed only once (if possible) and again the results are re-used. This leads to a reduction of the data volume flowing through the network and reduces the workload of the individual peers.

Let us show by means of an introductory example how StreamGlobe implements such an adaptive DSMS. It is based upon a visionary scenario in the field of astronomy, which we have developed in the context of data stream management for the *German Astrophysical Virtual Observatory (GAVO)* [GAV04] in cooperation with the *Max Planck Institute for Extraterrestrial Physics (MPE)*[2] and the *Astrophysical Institute Potsdam (AIP)*. The data used in this example has been collected by the *ROSAT* satellite during the *ROSAT All-Sky Survey (RASS)* [VAB+99]. This scenario is visionary, because most of the analysis tasks in astronomy are conducted on persistent data to date. However, this will not be feasible in the future due to tremendously growing data volumes. Hence, direct processing of data streams will be needed. The information retrieval network used in this simplified example is depicted in Figure 2.1. It is organized as a *super-peer network*[3] with SP_0 to SP_3

[2]Located in Garching near Munich, Germany

[3]This topology will be described in Section 2.2.2.

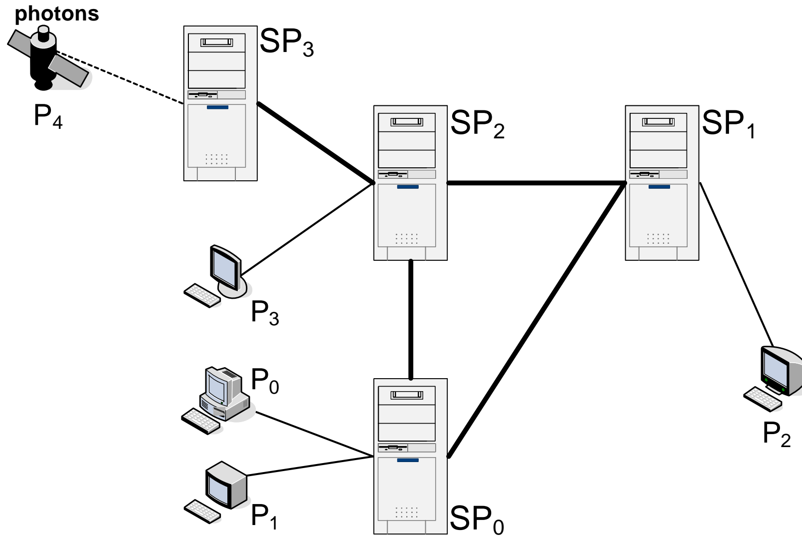


Figure 2.1: Example Astrophysical Scenario

constituting a *super-peer backbone*. Four (possibly mobile) clients P_0 to P_4 are connected to this backbone via super-peers. Peer P_4 depicts the ROSAT satellite. The detector of this satellite detects single photons and delivers each photon and some additional measurements directly as an XML data stream of photon events to the network. Peers P_0 and P_2 register subscriptions in this network to gather interesting information from the stream of photons. Such subscriptions have a broad scope: They can be used to acquire specific data researchers might be interested in for online analysis or to store it for offline computations. Further, in the field of virtual observatories, which are currently of great interest in the astronomical community, they can be used to automatically trigger actions based on observational data. For example, if data in an observational data stream has certain properties, an alert could be triggered, which automatically directs robotic telescopes to the area in the sky where something interesting happens to gather additional data.

We start our example with two basic subscriptions and we will extend it in Section 2.4. At first, let us have a look at the data stream, which is delivered by peer P_4 (the ROSAT satellite). As described before, the detector delivers an event for each detected photon, which contains a number of measurements. These measurements consist of temporal, positional, and spectral aspects. In detail:

- The position, where the photon came from, described by right ascension (`ra`) and declination (`dec`).
- Details about the error between real and measured position of the photon (`pos_error`).
- The detector pulse (`phc`) of the detected photon.
- The energy of the photon (`en`) computed from the detector pulse in keV.

- The number of the RASS field the photon came from (`field_id`). The RASS field is an artificial classification of the sky in 1378 areas with an approximate size of six times six degrees.

- The time, at which the photon was detected (`det_time`).

- The coordinates of the detector-pixel which recorded the photon (`dx`, `dy`).

All these measurements are converted into an XML stream compliant with the following schema. For the sake of brevity, we show the schema using a DTD [W3C04a]. However, StreamGlobe is also capable of using XML Schema [W3C04b].

```
<!ELEMENT photons   (photon*)>
<!ELEMENT photon    (ra, dec, pos_error, phc, en, field_id, det_time,
                     dx, dy)>
<!ELEMENT ra        (#PCDATA)>
<!ELEMENT dec       (#PCDATA)>
<!ELEMENT pos_error (#PCDATA)>
<!ELEMENT phc       (#PCDATA)>
<!ELEMENT en        (#PCDATA)>
<!ELEMENT field_id  (#PCDATA)>
<!ELEMENT det_time  (#PCDATA)>
<!ELEMENT dx        (#PCDATA)>
<!ELEMENT dy        (#PCDATA)>
```

Note that `photons` is the root tag of the XML document specified by this DTD. Such a root tag might not seem reasonable in a streaming environment, since the start tag `<photons>` is only delivered once at the very beginning of the data stream. If the data stream is subscribed at any later time, this tag would not be seen and no results would be produced. However, we try to adhere as close to the XML standards as possible and hence require XQuery subscriptions to always include the root element. Internally, StreamGlobe uses the root tag for synchronizing query processing. That is, whenever a subscription is registered, the stream is read until a consistent starting point is detected. In this example, this is the case whenever the end of a `photon` element is encountered. Then, StreamGlobe inserts an artificial `<photons>` tag into the input stream of this subscription to signalize the query engine to start its evaluation.

Now, let P_0 and P_2 be devices used by two different groups of astronomers. The first group wants to receive data about all photons in the area of the *Vela Supernova Remnant*. Hence, all relevant photon events with their coordinates, energy measurements, and the detection times shall be delivered to P_0. To verify and eventually recompute the energy of a photon its detector pulse shall be contained additionally. To acquire this data, the

following subscription “Vela”, phrased in XQuery, is registered as a subscription at peer P_0:

```
<photons>
  { for $p in stream("photons")/photons/photon
    where $p/ra > 120.0 and $p/ra < 138.0 and
          $p/dec > -49.0 and $p/dec < -40.0
    return
      <vela>
        {$p/ra} {$p/dec}
        {$p/phc} {$p/en}
        {$p/det_time}
      </vela> }
</photons>
```

The function `stream(...)` is one of our extensions to the XQuery language to specify a (possibly infinite) XML stream as the input of an XQuery in the same style as `doc(...)` is used for persistent XML documents. In this example, `photons` denotes the identifier of the stream generated by the satellite.

A second group working at peer P_2 shall only be interested in high-energetic photons of the smaller area denoted as the *RXJ0852.0-4622 Supernova Remnant* [Asc98]. Hence, P_2 should only get photon events with an energy of more than 1.3 keV. For receiving this data, that group registers the following XQuery subscription “RXJ” at peer P_2:

```
<photons>
  { for $p in stream("photons")/photons/photon
    where $p/en > 1.3 and
          $p/ra > 130.5 and $p/ra < 135.5 and
          $p/dec > -48.0 and $p/dec < -45.0
    return
      <rxj>
        {$p/ra} {$p/dec}
        {$p/en}
        {$p/det_time}
      </rxj> }
</photons>
```

Figure 2.2 shows the effect of selecting only the high-energetic photons. The graphs depict the number of detected photons with respect to their energy and their distribution in the sky. The upper two graphs show a section of the area of the Vela Supernova Remnant, i. e., the area of the RXJ0852.0-4622 Supernova Remnant, without performing a selection on the energy of the photons. Here, no new structures can be determined. When performing an additional selection on the energy of photons, as shown in subscription “RXJ”, a new structure in the lower two graphs appears as the RXJ0852.0-4622 Supernova Remnant, which was discovered this way.

Traditional systems handle this situation in the network we have given at the beginning of the example as depicted in Figure 2.3. For each subscription, the data stream is

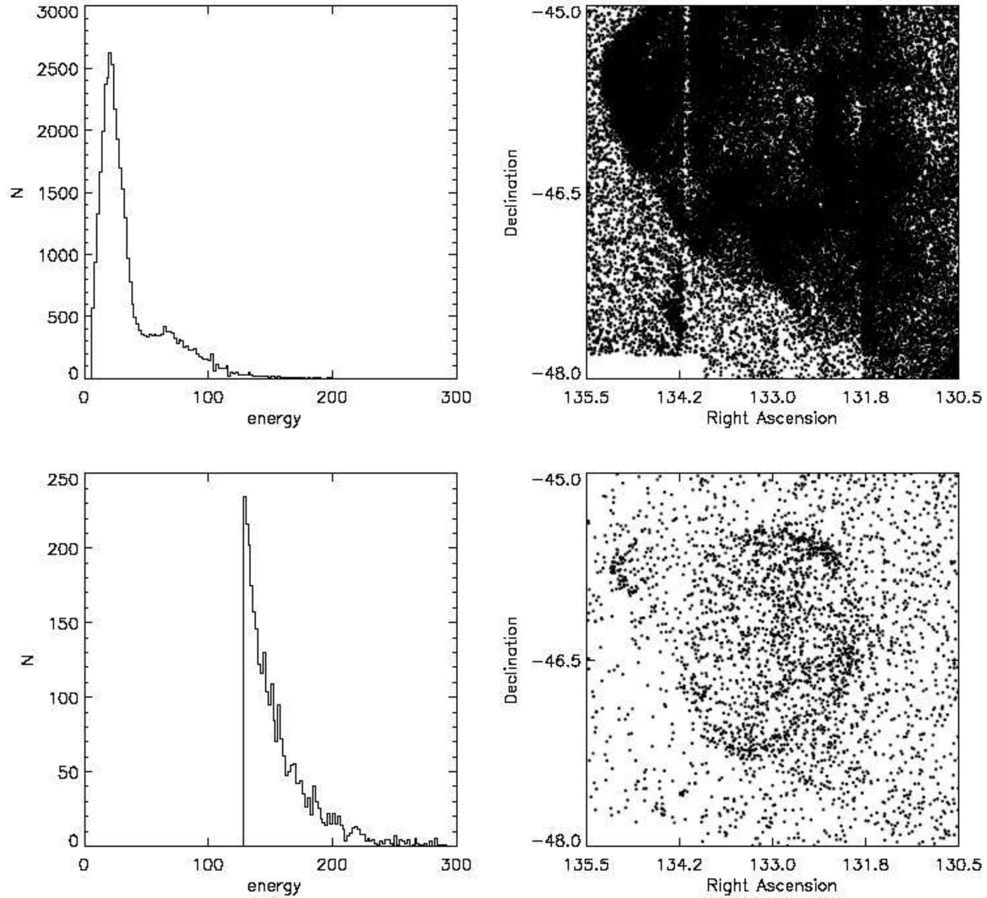


Figure 2.2: Visualization of the ROSAT Photon Data

transmitted individually to the peer where the subscription is registered. On this peer the query representing the subscription is evaluated and the results are delivered to the users. Such architectures naturally congest the underlying network. First, unused data is transmitted, e. g., P_0 does not need the measurements `dx` and `dy`. Second, the data streams are individually delivered and therefore a single stream is transmitted multiple times in the network, e. g., at the network connections $P_4 - SP_3$ and $SP_3 - SP_2$[4].

StreamGlobe handles this scenario as depicted in Figure 2.4. Data from P_4 is transmitted to SP_3 and filtered at this peer to remove those parts of the data as early as possible, which are not needed by any of the subscriptions. This is done by applying suitable projection and selection operations. In this case, it is easy to see that the area of the sky of

[4]Of course, we exaggerated our example in the sense that the satellite transmits the stream two times to SP_3. In reality, this would be avoided by using some kind of "proxy", e. g., the satellite would transmit its stream to SP_3. From there it would be routed to the other peers. However, this does not solve the basic problem, but only shifts it to the problem of statically determining a suitable network topology having such proxies at special nodes.

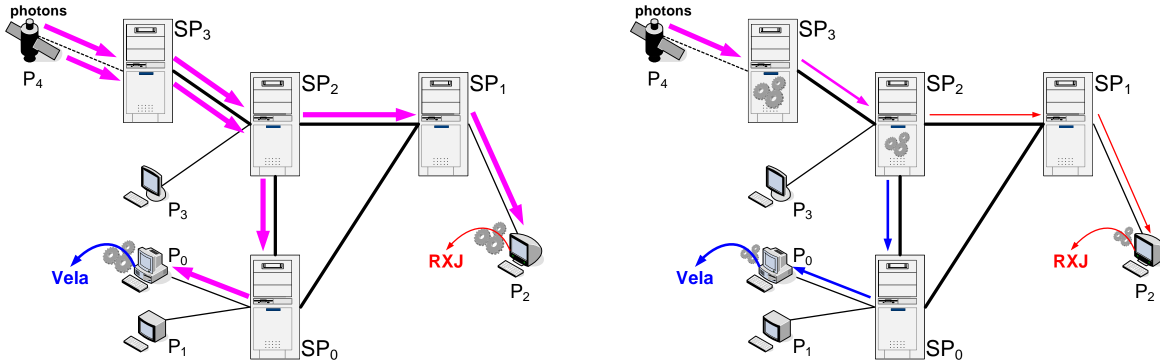


Figure 2.3: Traditional Subscription Evaluation

Figure 2.4: Optimized Subscription Evaluation

query "RXJ" is completely contained in the subscription "Vela", so that all photon events not stemming from the Vela Supernova Remnant can be eliminated. Further, the elements `pos_error`, `field_id`, `dx`, and `dy` are not needed by either query and thus are projected out. Additional selections, e. g., with respect to the energy of a photon, cannot be applied at this peer, since subscription "Vela" at P_0 does not impose further restrictions regarding the energy of a photon. This stream, containing the combined information for the two subscriptions (visualized by the purple color) and having a smaller data volume than the original stream (visualized by thinner arrows), is routed to SP_2, where it is split up into two streams. The first stream is routed to P_0 via SP_0. This stream only contains the information needed to compute the query "Vela" (marked in the figure in blue color). A second stream is generated by applying the selection of high-energetic photons and delivered to P_2 via SP_1. Again, this stream (marked red in the figure) only contains data for the subscription "RXJ".

Altogether, StreamGlobe combines routing techniques and in-network query processing capabilities to reduce the data volume being transmitted in the network, as it can be seen easily by comparing Figures 2.3 and 2.4. Moreover, filtering of the data streams also reduces the processing cost at the peers, since smaller streams have to be processed. This can be seen as some kind of parallel processing in the network, since the computation of parts of subscriptions is spread over suitable peers of the network. This concept of combining routing and query processing to reduce network traffic is an important contribution of StreamGlobe and distinguishes it from existing systems and techniques, e. g., multicasting. Another contribution, which we have not shown in this first example, is capability-based execution of subscriptions. That is, (parts of) subscriptions are not only executed on peers with respect to minimizing network traffic and peer load, but also with respect to the individual capabilities of peers. For example, mobile peers, e. g., PDAs, cell-phones, and so on, do not provide extensive query processing capabilities. If a subscription is registered at such a peer, it will be processed on some other suitable peer in the network and only the results will be delivered to the target peer.

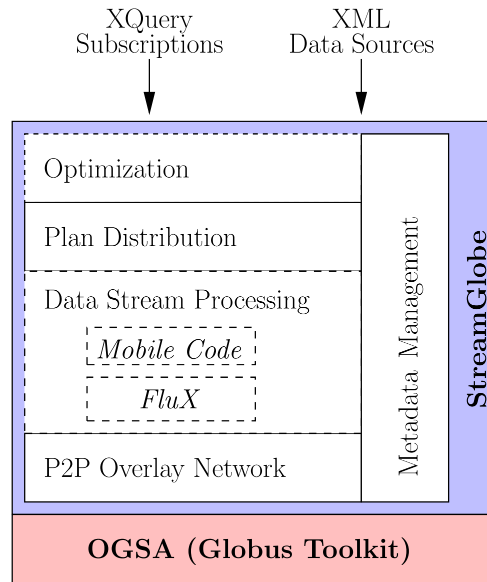


Figure 2.5: StreamGlobe Architecture Overview

2.2 The StreamGlobe Architecture

StreamGlobe constitutes a federation of servers (i. e., peers) which carry out query processing tasks according to their capabilities. The basic structure of a peer is depicted in Figure 2.5. We will sketch the various layers of this structure in a bottom-up fashion in the following. We will omit the "Optimization" and "Data Stream Processing" components in this section and present them in more detail in the next section. The dashed lines of these components of StreamGlobe denote that they are optional depending on the capabilities of the actual peer.

2.2.1 The Fundamentals: Open Grid Services Architecture

The StreamGlobe architecture is based on Grid standards. As mentioned in the motivation, Grid Computing [FK04, FKT01] and the associated Open Grid Services Architecture (OGSA) [FKNT02] have recently gained considerable attention, especially in the high-performance and distributed computing communities. Grid Computing denotes a distributed computing infrastructure where computers are able to exchange data and to perform large-scale resource sharing over the Grid. To achieve this, the Open Grid Services Architecture has been developed for integrating heterogeneous dynamic services while guaranteeing certain quality-of-service requirements.

Despite the growing importance of the Grid standards, data stream processing in the Grid Computing context has hardly been investigated so far. We have decided to implement our StreamGlobe prototype as an extension of the Globus Toolkit for Grid Computing [Glo04], which is a reference implementation of the Open Grid Services Architecture.

Our goal is to use existing Globus techniques for our purposes where possible and to implement the StreamGlobe system and its functionality on top of the Globus toolkit as an extension for data stream processing. This enables a seamless integration of StreamGlobe into already established Grid infrastructures of the various communities.

The StreamGlobe system itself is implemented as a set of collaborating Grid services, which represent the different layers of Figure 2.5, based upon the Globus Toolkit. The main technical aspects of Globus used in StreamGlobe are communication mechanisms and *service data elements*. Service data elements can be associated with any service in the Grid. They are essentially XML documents satisfying a given XML Schema and describing properties of the service they are associated with.

Unfortunately, the OGSA framework does not yet provide any means for stream communication or P2P networking. Hence, we have implemented these layers on our own directly in the StreamGlobe system. For instance, we have implemented a stream communication protocol between peers on top of TCP/IP interfaces provided by Java. However, there are ongoing efforts [GGF05] for integrating both stream communication mechanisms and a P2P networking layer into the OGSA framework. As soon as results in this area are available, we will utilize them in StreamGlobe to be only based on native Grid technology.

2.2.2 Network Topology

In the OGSA framework, direct communication between all participating Grid services is allowed. However, this behavior is not the natural way of communication in networks including mobile devices. It might not even be desirable in a scenario that tries to reduce network traffic as we intend to do. For instance, mobile sensors will normally communicate via some kind of access point which they are connected to.

In StreamGlobe, we distinguish two types of communication: *data transfers* and *control messages*. The first are data streams fed into the network by peers acting as data sources. The latter are used for managing the network, e. g., metadata requests. Obviously, data transfers have a high data volume, whereas control messages are only small one-time messages. Hence, we establish a logical P2P overlay network only for data transfers to be able to control the data flow in the network. As usual, each peer in the overlay network has a set of other peers as neighbors. A peer establishes data transfers only with its neighbors, i. e., no direct data communication takes place between two peers not being neighbors. If data has to be transferred between any two peers, a *route* between these two peers has to be established so that two successive peers on this route are neighbors and the starting point and the end point of the route are the source peer and the destination peer, respectively. This notion of the P2P-Overlay network distinguishes StreamGlobe from many other PDMSs: Searching data is carried out similarly by walking from a peer to neighbored peers until suitable data is found. But once the data has been found, it is normally directly sent back to the peer that has issued the query. Such a technique is obviously not well suited for establishing and optimizing continuous data transfers in a network as our approach aims at. For the implementation of the overlay network, previous work on P2P network topologies can be employed, e. g., a structured approach based on Cayley graphs

as used in the HyperCup [SSDN02] topology. However, developing a research platform, we do not restrict ourselves to utilize a special P2P network topology at the moment. The analysis of suitable topologies for StreamGlobe is planned for future work.

Since a major goal is building a network of highly heterogeneous peers with respect to computing power—ranging from small, mobile devices to stationary workstations or servers—, we have to classify peers according to their capabilities. *Thin-peers* are devices with low computational power, like sensor devices, PDAs, cell phones, etc., which are not able to carry out complex query processing tasks. In contrast, *super-peers* are stationary workstations or servers providing enough resources for extensive query processing. These super-peers establish a backbone taking over query processing tasks which cannot be performed by other peers. Thus, they constitute a super-peer backbone network [YGM03]. This network topology renders the StreamGlobe network different from other P2P networks in the sense that we require the super-peer backbone to be relatively stable, i. e., super-peers should not join or leave the network frequently. Of course, thin-peers are allowed to arbitrarily join or leave the network similar to traditional P2P networks.

Experiences with different P2P systems like Napster or Gnutella have shown that the performance and the scalability of a P2P network heavily depends on metadata management. Napster uses a centralized directory, which makes it easy to process metadata requests of peers, but eventually becomes a bottleneck with a growing number of peers. On the other extreme, Gnutella is organized completely decentralized. This results in flooding the network (up to a certain "horizon") whenever a peer asks for metadata, since it has to query all its neighbors, all those neighbors their neighbors, and so on. To bring out the best of both extremes, we took the following hierarchical approach: The whole network is divided into non-overlapping subnets. Each subnet consists of a set of super-peers and peers. Among the super-peers, a *speaker-peer* is elected[5] which is responsible for managing this subnet. In detail, the speaker-peer has the following management tasks:

Subnet Management Peers entering this subnet contact the speaker-peer. The discovery of a speaker-peer is implemented via broadcast mechanisms. The speaker-peer determines the best suited super-peer to be registered at and forwards the registration to this super-peer. If the subnet grows beyond a certain limit, the speaker-peer initiates a split of the subnet into two subnets. In each of the new subnets again a speaker-peer will be elected for management. Similarly, if a peer leaves the subnet, the super-peer at which it was registered notifies the speaker-peer. If the subnet shrinks below a certain size, the speaker-peer initiates a merge of this subnet with a neighboring subnet. As before, in this new subnet a new speaker-peer is elected among the set of super-peers.

Subscription Optimization The speaker-peer is responsible for optimizing the evaluation of subscriptions of its subnet, as it has been outlined in the motivation. The concepts of this optimization will be presented in detail in Section 2.3.

[5] Such an election can be implemented, e. g., by means of the Bully Algorithm [GM82].

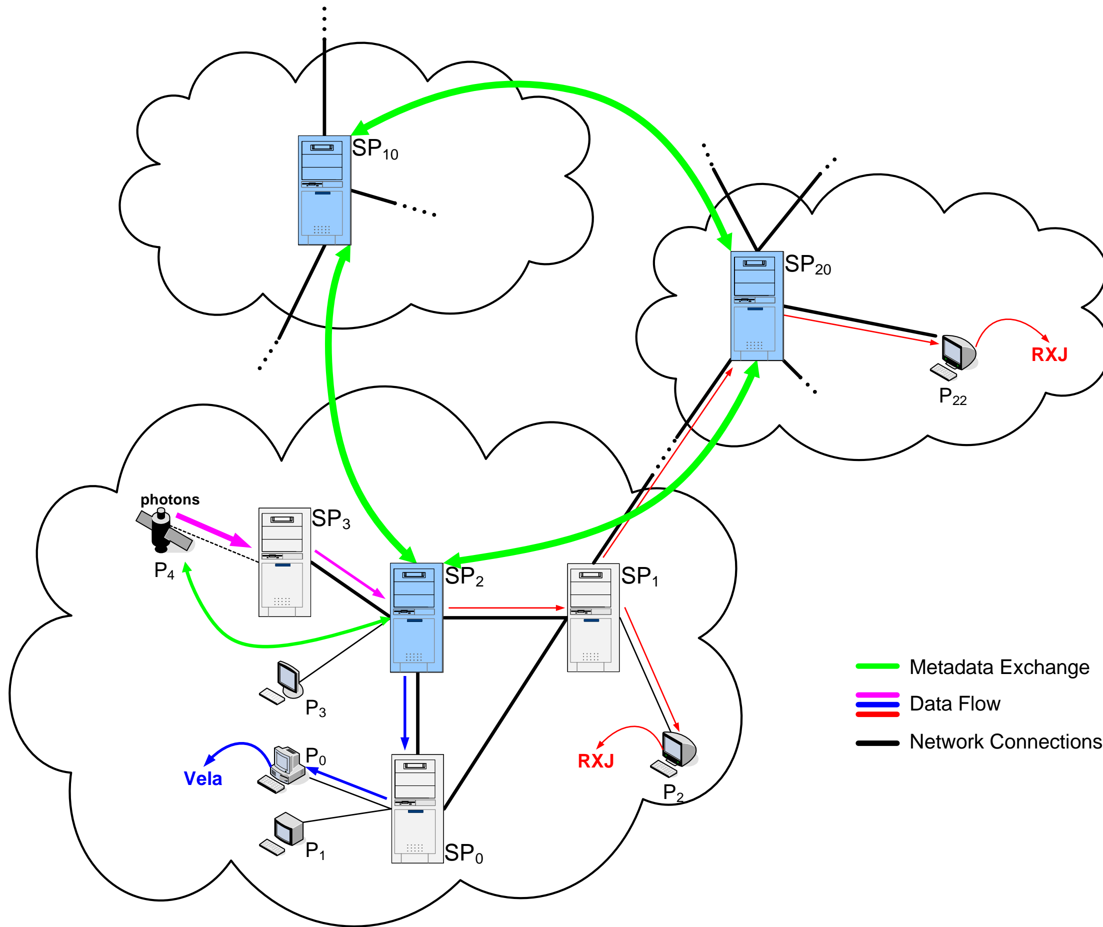


Figure 2.6: Network Topology of the Extended Example Scenario

Metadata Management The speaker-peer maintains and provides global metadata of its subnet. Further details on the metadata management will be shown in Section 2.2.5.

Inter-Subnet Communication Speaker-Peers are responsible for the communication between different subnets. For instance, if a peer subscribes to a data stream, which is not yet known in the subnet of this peer, the speaker-peer contacts the speaker-peers of neighboring subnets to locate this data stream. Another example is a peer joining the network. If the discovered speaker-peer decides that this peer should not participate in this subnet—due to, e. g., local aspects—it forwards this peer to the speaker-peer of another subnet.

An overview of this hierarchical network topology is shown in Figure 2.6 by means of an example which is a slightly extended version of the example already known from Section 2.1. Three subnets are shown, with one of them being exactly the network depicted in Figure 2.1. In each subnet, the speaker-peers are marked by blue color. The P2P overlay network is

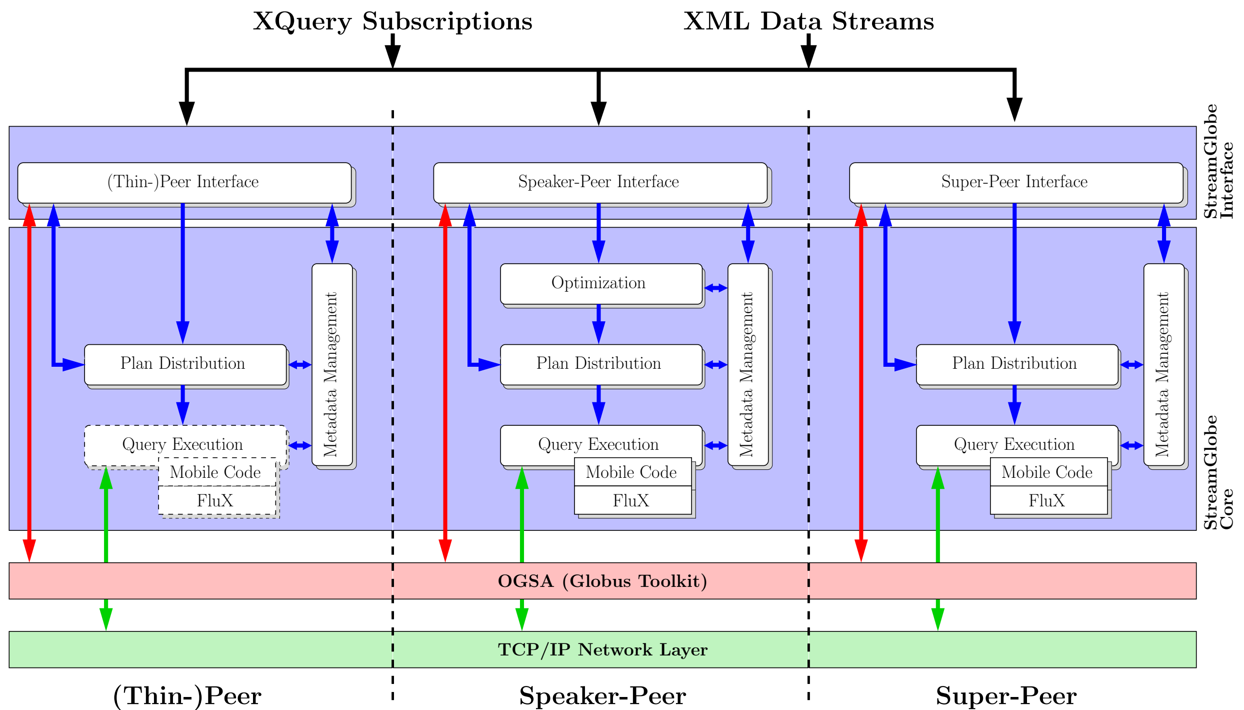


Figure 2.7: Architecture Overview of Different Peers

represented as black connections between the peers. Of course, peers of different subnets are allowed to be connected to each other as neighbors. The colored arrows depict the data flow in the network. Control message communication, e. g., metadata exchange, is shown by green arrows. The speaker-peers also form a metadata-backbone above all subnets. In each subnet, the speaker-peer is able to query metadata from a peer—for instance, in our example from peer P_4, which represents a data source and publishes the metadata of this data stream. The separation of data transfer and the exchange of control messages not being bound to the P2P neighborship relations is clearly visible in this example. As outlined before, we have chosen this approach as a trade-off between flexibility with respect to management of the network and being able to optimize the high-volume data flow.

2.2.3 Peer Architecture

At the beginning of this section, we have shown a high-level overview of the StreamGlobe architecture. Figure 2.7 provides a deeper insight into its design.

Basically, all layers are implemented as collaborating Grid services, which are represented by rounded rectangles. According to the classification of peers, the provided services of a peer are shown individually for each type. The Grid services are divided into two groups: *Interface services* represent the different peers in the network. These services constitute the interfaces of peers to the rest of the StreamGlobe system and forward all

requests to appropriate *core services*. The core services implement the functionality of a peer. Every peer runs exactly one instance of an interface service and a set of core services according to its capabilities as follows.

Thin-peers are peers with the minimum amount of functionality. They basically are only able to publish data streams into the network and to receive the results of their subscriptions, but do not carry out query processing. Such a peer mainly runs the metadata management service, since it also has to provide metadata, e. g., statistics of a data stream. Beyond it, only the plan distribution service and a restricted query execution service is provided. The plan distribution service is needed for all peers, since this component is responsible for correctly setting up data communication with other peers and instantiating the queries in the query execution service. The query execution service of a thin-peer is only able to display results of subscriptions, to publish data streams, and to maintain statistics of data streams published by this thin-peer, if needed.

With respect to capabilities, more powerful peers are super-peers. They provide basically the same set of services than thin-peers, but offer extensive query processing capabilities. Thus, their query execution service is able to invoke fully fledged query processing for data streams. For carrying out query processing tasks, the query execution service has two possibilities: the general XQuery engine *FluX* for data streams, which will be presented in Section 3, or user-defined stream operations may be utilized. These user defined stream operators are implemented by users as mobile Java code and therefore any query execution service is able to load and execute such an operation. This provides for a great amount of flexibility for specifying subscription rules.

Last but not least, speaker-peers are basically super-peers with the additional task to optimize and manage their subnets. Hence, their basic structure is equal to that of super-peers. Additionally, speaker-peers provide an optimization service taking over all the optimization tasks and the management of the subnet.

In Figure 2.7, the communication paths are depicted by arrows. Blue arrows represent direct communication between different services on a single peer within the StreamGlobe system. Inter-peer communication, which is shown in red, uses the RPC mechanisms provided by the Globus Toolkit. Finally, data stream transfers are shown in green. Since the OGSA framework does not yet provide any suitable means for data stream transfer, query execution services directly communicate with each other using TCP/IP networking techniques.

Note that there is no service representing the P2P-Network layer. This layer is implicitly contained in the metadata and optimization services. Since the optimization service computes and instantiates the data-flow at the involved peers of the network, this component ensures that data streams are routed consistently within the P2P network structure.

2.2.4 Client Interface

User interaction in StreamGlobe is depicted at the top of Figure 2.5 and at the interface layer of Figure 2.7. Clients specify subscription rules for information processing and retrieval using the XQuery language. Currently, not the whole XQuery language is sup-

ported, but an expressive fragment being significant for data stream processing. This fragment will be specified in Section 3.4. Subscription rules are registered at certain peers, i. e., normally at the devices users are working with, e. g., their laptops, PDAs, cell phones, etc. In our context, subscriptions are transforming queries and not just queries for retrieving and delivering matching files or documents. In fact, StreamGlobe enables expressive transformations of data streams using subscription rules. Thus, it allows clients to flexibly tailor data streams to their individual requirements.

Similarly to clients subscribing to needed information, data sources also register the provided data streams at a certain peer within the StreamGlobe system. Data streams can be registered in two ways. A data source may register a data stream as an individual stream, which is then published using a unique identifier. Subscriptions refer to streams using this identifier. Another possibility is registering a data stream as part of a *virtual data stream*, which again is accessible using a unique identifier. StreamGlobe multiplexes all the data of the participating data sources into one single stream, which will (internally) be available at one single peer chosen by the StreamGlobe optimizer.

Schemata of data streams are specified using XML Schema. Non-XML data streams are fed into StreamGlobe using *wrappers*, which are running on corresponding peers and transform the data into a suitable format, e. g., by converting raw sensor data to XML. These wrappers for non-XML data streams have to be provided by the peer which is publishing a data stream.

2.2.5 Metadata Management

As Figure 2.5 suggests, metadata is needed in all layers of the StreamGlobe architecture. The metadata management (MDV) is based on techniques of the distributed metadata management of ObjectGlobe [KKKK02]. As shown in Section 2.2.2, speaker-peers constitute a metadata backbone that peers exchange metadata with.

In particular, the metadata management component records the following information:

- **Network:** The MDV stores the neighborhood relationships between peers needed for establishing the P2P overlay network.

- **Subscriptions:** All subscriptions and registered data sources are recorded. For each registered data source, the schema of the data stream is stored.

- **Optimization:** The metadata management provides information needed for optimizing the network. Among others, it maintains properties of network connections, like bandwidth and current amount of network traffic. It also maintains the computational capabilities of peers and statistics of data streams, i. e., size, cardinality, and histograms of the elements of a data stream. The statistics can be provided either by the data source itself or by computing them online as the corresponding wrapper feeds the data stream into StreamGlobe.

Super-Peers and thin-peers store their own metadata locally. That is, a peer knows which peers are its neighbors and which subscriptions and data sources are currently registered and provided, respectively. Furthermore, peers publishing data streams provide details about these streams like their schemata and statistics. To maintain a consistent state, a peer notifies its super-peer of changes, e. g., if it leaves the network, new subscriptions or data streams are registered, or existing subscriptions or data streams are de-registered. These notifications are forwarded to the speaker-peer of the corresponding subnet. As a result, the speaker-peer is able to maintain additional "global" metadata about the subnet needed for performing subnet management and optimization. For instance, the speaker-peer maintains a graph of the current network topology of its subnet. All other metadata requests, e. g., other speaker-peers querying metadata about a certain data stream, are forwarded to the appropriate peer.

Altogether, the speaker-peers of the entire network constitute a metadata backbone, which enables an efficient and scalable metadata management.

2.3 Subscription Evaluation

In Section 2.1, we have briefly introduced our concepts of optimizing the data flow in a network using in-network query processing and data stream sharing. In the following, we explain the optimization and evaluation strategy employed in StreamGlobe in more detail.

2.3.1 Optimization Goals and Strategy

Basically, the optimization component of a speaker-peer analyzes all subscriptions to identify common parts, determines the peers at which subscriptions or common parts of subscriptions should be evaluated, and decides how to route the data streams in the subnet. The optimization has the following major goals:

1. Enable users to register arbitrary subscriptions at any device regardless of its processing capabilities.
2. Achieve a data flow in the network which does not congest it with redundant transmissions of data streams.
3. Optimize the evaluation of a large number of subscription rules by means of multi-query optimization.
4. Avoid overload situations at both peers and network connections, e. g., caused by evaluating too many subscriptions on a single peer or exceeding the maximum bandwidth of network connections.

The general approach for achieving these optimization goals is to rewrite subscriptions into one or more queries and to spread the execution of these queries all over the network. To clarify this idea, let Q be a new subscription to be registered at some peer. During

the registration process, the optimization component analyzes Q with respect to all currently registered subscriptions in the speaker-peer's subnet and therewith computes a set of queries $q_1, \ldots, q_n$, such that the successive execution $q_n \circ \cdots \circ q_1$ yields the same result as executing the original subscription Q. We distinguish two types of queries q_i: *reducing queries* and *non-reducing queries*.

Definition 2.3.1 (Reducing Query) *A query q_i is denoted as a* reducing query*, if the data volume of the computed data stream is less than the data volume of its input data streams.*

Definition 2.3.2 (Non-Reducing Query) *A query q_i is a* non-reducing query*, if it is not a reducing query.*

The optimization goals are accomplished by pushing the execution of $q_1, \ldots, q_n$ into the network. These queries are evaluated at suitable peers on the route starting at the data sources and ending at the peer having registered the subscription. A peer is suitable for executing a query, if it is capable of processing the query, has been chosen by the optimizer with respect to the optimization goals, and will not get into an overload situation executing the novel query. In the following, we will show the achievement of some of the individual goals in detail.

Establishing a good distribution of data streams in the network by avoiding redundant transmissions is achieved using two techniques complementing one another. The first is *early filtering* of data streams. That is, reducing queries are executed on suitable peers on the route of a data stream as close to the data source as possible. These reducing queries normally consist of projections (structural filtering) and selections (content-based filtering) contained in subscriptions, but could also be a highly selective join. Non-reducing queries of a subscription are executed as close to the peer, which has registered the subscription, as possible. Thus, the data volume of data streams transmitted in the network is reduced. The complementary technique is *data stream sharing*. The idea of data stream sharing has already been outlined in the introductory example of Section 2.1: Instead of transmitting a data stream individually to each peer having subscribed to this stream, only a single data stream serving the needs of all subscriptions is transmitted on the common parts of the routes from the peers to the data source. This approach is similar to multicasting techniques, e. g., known from the TCP/IP protocol. These techniques also provide some kind of stream sharing over single network connections, but they disregard content based aspects (i. e., in-network query processing capabilities) by always transmitting the whole data from the data source to all recipients. In contrast, our combination of data stream sharing and filtering is able to shrink data streams on the way from the data source to the recipients with respect to the subscribed content of the data stream and therefore further reduces network traffic. Of course, the computation of reducing queries and data stream sharing schemes are closely related. The biggest challenge for the optimizer is to compute both the best possible data stream sharing scheme, i. e., routes for each data stream, and the appropriate reducing queries.

Data stream sharing implicitly also contributes to the achievement of the third goal of multi-query optimization. As outlined before, the optimizer analyzes a novel subscription with respect to all existing subscriptions. The computed set of queries $q_1, \ldots, q_n$ may contain a query q_i, which constitutes a common part of the novel and already existing subscriptions[6]. To perform multi-query optimization, query q_i is executed only once at a suitable peer. Therefore, already existing subscriptions are rewritten to re-use the data stream generated by q_i. Besides reducing the load of peers, this technique may also contribute to the goal of reducing network traffic. For instance, a common task in a sensor network is computing aggregates of sensor measurements. Instead of transmitting the measurements data stream to every peer computing the same aggregations, e. g., a sliding window average, the computation of the aggregate operation is executed near the data source and only the results, which will in general be much smaller, are transmitted to the recipients. Such aggregated data streams can be re-used even if different aggregation parameters, e. g., different window sizes for sliding window averages, are needed by applying techniques similar to roll-up and cube operators known from the data-warehouse context.

Having discussed our optimization goals and how to achieve them, we now discuss optimization strategies in a distributed architecture. There are basically three strategies for performing optimization tasks in such an environment:

1. A central optimization component which has global knowledge of all needed metadata performs optimization with a global view of the network.

2. Every peer has only local knowledge of its own metadata (including that neighbors can be asked for their metadata) and tries to achieve an optimal network state by making locally optimal decisions.

3. A hybrid approach, in which special peers have knowledge of (small) subnets which are individually optimized by the responsible peer.

Since we assume a large, distributed environment, a centralized optimization component as depicted in (1) is obviously infeasible because of the huge search space an optimizer would have to consider.

The second approach at first glance fits quite nicely into the context of a distributed P2P network. To be able to deliver good results, such an optimization strategy depends on the validity of Bellmann's principle of optimality, i. e., that a globally optimal solution only consists of locally optimal solutions. But, yet in the field of distributed query optimization this principle only holds under certain stringent restrictions. For example, it is violated, if semi-join filters are possible query operators. In this case, a query plan consisting of locally optimal subplans might not constitute the globally optimal query plan, because at a higher level a bigger benefit might be achieved from non-optimal efforts on a lower level. We are facing a similar situation in our context, which most likely renders such an approach unsuitable to deliver good optimization results.

[6]Of course, this is possibly the case for more than one query q_i.

Hence, we focus on the hybrid approach: As already outlined in Section 2.2.2, a speaker-peer is responsible for optimizing its local subnet. Thus, the search space of the optimizer can be reduced to a feasible size by limiting the maximum size of the subnets. On the other hand, in contrast to approach (2) knowledge of the whole subnet enables more powerful optimizations. For example, a peer may reduce a data stream, which might not be the optimal solution from this peers point of view, but routing this reduced data stream to other peers might lead to a significant reduction of network traffic in the subnet.

Another important aspect is when and what kind of optimization is invoked. Basically, StreamGlobe—or, the speaker-peer of a subnet, respectively—carries out optimization whenever a new subscription is registered. For this optimization, two possible approaches exist:

Global Optimization All previously registered subscriptions and the new subscription to be registered are taken into account to compute a completely new, optimal routing and data stream sharing.

Incremental Optimization The optimizer tries to fit the new subscription into the existing routing and data stream sharing situation as good as possible—without affecting already existing subscriptions.

Incremental optimization can be carried out by analyzing existing data streams and therewith deciding what data streams may be shared or not and where a new subscription is additionally executed. In contrast, global optimization is a very costly task. First, finding the best data stream sharing and routing scheme in the subnet is a much more complicated optimization problem than only analyzing what data streams can be shared for serving the new subscription. Second, after a new data stream sharing and routing scheme has been computed, the whole subnet has to be restructured accordingly. In particular, this only can be done by synchronously stopping all affected queries, instantiating the new situation, and again synchronously starting the new queries without loosing any data in the meantime.

Hence, whenever a new subscription is registered, StreamGlobe integrates it into the network using incremental optimization to enable efficient subscription registration without affecting the performance of the system too much. Because of this, the state of the network could degrade over time with more and more subscriptions being registered. To overcome this problem, additional global optimization is triggered in an event-based fashion. Such events can either be periodically generated or alerts of a monitoring component indicating a non-optimal state of the network can be used. Therefore, the whole network is reorganized if necessary and hence can be kept in a near-optimal state in terms of efficient query processing and data flow, without the need for explicit external intervention.

2.3.2 Optimization Algorithm

In this section, we present a brief overview of the optimization algorithm. Details on the optimization algorithms can be found in [KSK05] and are beyond the scope of this thesis.

All optimization algorithms get a set of metadata, consisting of, e. g., the network topology, statistics about data streams, and the current state of the network, and a set of subscriptions as input. The output is a query plan describing where to execute which reducing or non-reducing queries. This query plan is then forwarded to the plan distribution services of the corresponding peers, which instantiate and initiate query processing services.

We start by explaining how the incremental optimization is done whenever a new subscription is registered on the example introduced in Section 2.1. We assume that the query "Vela" has already been registered in the network. As a result, a filtered data stream for this query—as described before—is routed through the network from P_4 to P_0 via SP_3, SP_2, and SP_0. Now, query "RXJ" is registered at peer P_2. As a first step, this novel subscription is analyzed with respect to what part of the original data stream is needed by considering contained projections and selections. Then, a reducing query is generated to reduce the data volume of the data stream that has to be routed through the network. In this example, the reducing query, denoted as "RXJ-filter", is as follows:

```
<photons>
  { for $p in stream("photons")/photons/photon
    where $p/en > 1.3 and
          $p/ra > 130.5 and $p/ra < 135.5 and
          $p/dec > -48.0 and $p/dec < -45.0
    return
      <photon>
        {$p/ra} {$p/dec}
        {$p/en}
        {$p/det_time}
      </photon> }
</photons>
```

Using this filter query the original subscription is rewritten to work on the filtered data stream generated by the filter query. The resulting query "RXJ-trans" is:

```
<photons>
  { for $p in stream("rxj-photons")/photons/photon
    return
      <rxj>
        {$p/*}
      </rxj> }
</photons>
```

Note that the filter query does not change the basic structure of the stream apart from discarding unneeded parts. In particular, it does not introduce the novel tag `<rxj> ... </rxj>` instead of the original `<photon> ... </photon>` tag. This is accomplished by the transformational query. Also note that the input stream of "RXJ-trans" has changed to `stream("rxj-photons")`, which identifies the stream generated by the filter query.

Now, these two queries generated from the original subscription have to be executed on suitable peers in the network. To find the optimal execution peers, the optimizer

enumerates all possible query plans in the current network situation with different execution peers for these two queries. To do so, the algorithm has to compare existing data streams to a query with respect to whether they may be re-used as an input for this query, i. e., if all the needed content is contained. Therefore, for any data stream in the network—either stemming originally from a data source or being the result of some other query—, *properties* are stored. These properties contain the identifier of the data stream, the data stream it was derived from, and operators that have been applied to it, e. g., projections or selections, together with their parameters. By comparing the applied operators contained in the properties of a data stream to the operators of the query to be matched against the stream, the optimizer is able to decide whether this stream is suitable for sharing, or not. To enumerate only presumably good candidates, only routes with a minimal number of hops between the peer, at which the subscription is registered, and the data source are considered. Note that the term "data source" in the context of optimization not only denotes the peer feeding the subscribed data stream into the network, but also all other peers having a suitable data stream which can be shared. Having enumerated all possible query plans, the optimal query plan is determined by comparing all query plans using a cost model. For that purpose, we have developed a cost model accounting for network traffic, maximum bandwidths of network connections, and workload of peers [KSK05]. With this, the optimal plan in terms of the goals stated in the previous section can be found.

Coming back to our example, the optimizer at first searches for data sources for the filter query "RXJ-filter". A candidate is super-peer SP_2 since it can provide a filtered instance of the original data stream. Comparing the properties of this data stream to the requirements of "RXJ-filter" yields, that this stream is suitable for being shared, since the selection predicates and projections are more general than that of "RXJ-filter" and thus all needed data is contained. Now, the rewritten subscription "RXJ-trans" is executed on the peer where it has been registered, since we assume that this peer is capable of executing this query. If this would not be the case, it would be executed on the nearest peer being capable of evaluating it on the shortest path to SP_2, e. g., SP_1. Another possible query plan would be to directly use the original data stream at P_4 as data source for "RXJ-filter". This candidate will be rejected by comparing costs of these two query plans, since it would cause more network traffic than the former query plan, because the "photons" stream would be transmitted redundantly as shown in Figure 2.3. The XML representation of this new query plan for adding the subscription "RXJ", which is forwarded to the plan distribution components, is shown in Figure 2.8. It contains the computed set of queries (enclosed by the `source` tags), the peers at which they are executed (as values of the `atPeer` attributes), which data streams are used as inputs, and identifiers for the queries and data streams. Data streams being the results of queries are identified by means of the corresponding query identifiers. As explained before, data stream sharing is done in the filter query (apparent in `stream("vela")` and `<streamreference streamref="vela"/>`), by re-using the data stream generated by the subscription "Vela", which has been registered before, instead of working on the original "photons" data stream. Figure 2.9 shows a graphical representation of the network situation after the computed query plan has been integrated into the network. The symbols at the network connections denote groups of elements of

```
<plan atPeer="P_2" id="rxj_p2" xmlns="urn:streamglobe.in.tum.de/pdc"
  xmlns:pdc="urn:streamglobe.in.tum.de/pdc"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
  <add>
    <streamoperator id="result" name="display" xsi:type="builtInStreamoperator">
      <dependencies>
        <streamreference streamref="rxj-trans"/>
      </dependencies>
    </streamoperator>
    <streamoperator id="rxj-trans" name="query" xsi:type="queryStreamoperatorType">
      <dependencies>
        <streamreference streamref="rxj-filter"/>
      </dependencies>
      <source><![CDATA[
        <photons>
          { for $p in stream("rxj-filter")/photons/photon
            return
              <rxj>
                {$p/*}
              </rxj> }
        </photons>
      ]]></source>
    </streamoperator>
  </add>
  <plan atPeer="SP_2" id="rxj_sp2" xmlns="urn:streamglobe.in.tum.de/pdc"
    <add>
      <streamoperator id="rxj-filter" name="query" xsi:type="queryStreamoperatorType">
        <dependencies>
          <streamreference streamref="vela"/>
        </dependencies>
        <source><![CDATA[
          <photons>
            { for $p in stream("vela")/photons/photon
              where $p/en > 1.3 and
                $p/ra > 130.5 and $p/ra < 135.5 and
                $p/dec > -48.0 and $p/dec < -45.0
              return
                <photon>
                  {$p/ra} {$p/dec}
                  {$p/en}
                  {$p/det_time}
                </photon> }
          </photons>
        ]]></source>
      </streamoperator>
    </add>
  </plan>
</plan>
```

Figure 2.8: XML Representation of Query Plan

the original data stream. The diamond denotes the elements `pos_error`, `field_id`, `dx`, and `dy`, the circle `phc`, and the rectangle `ra`, `dec`, `en`, and `det_time`. Projections remove symbols, because corresponding elements are removed from the data stream. Selections

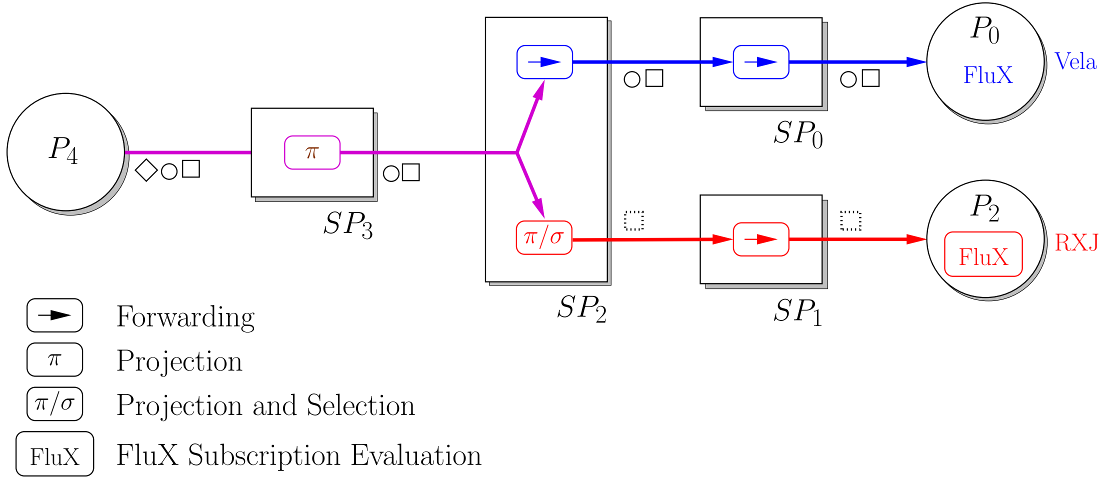


Figure 2.9: Query Plan of the Introductory Example

filter instances of elements not satisfying the selection predicate, which is visualized by dotted symbols.

Note that in this simple example the subscription has been split into only one filter query and one transformational query. For more complex subscriptions and network situations, this is normally not the case. In particular, if filter queries are merged, they might not only have to be executed at a suitable peer, but again be rewritten. For instance, suppose that a computed filter query contains a predicate, which is already contained in an existing filter query suitable for sharing the data stream with the new filter. In this case, we would not simply install the new filter query, because this predicate would then be evaluated twice. In fact, the new filter query would be rewritten such that this predicate is removed and the data stream of the existing filter would be used as input instead. So, in general, a single subscription will consist of a set of queries being evaluated in the network in a distributed fashion, depending on the current situation in the network.

Global optimization works similar to incremental optimization. Instead of only adding a single subscription into an existing network situation, all registered subscriptions are inserted simultaneously into an initially empty network. This allows for more powerful optimizations, but obviously increases the search space of the optimization problem and hence depends on efficient optimization algorithms. As already mentioned, we will not go into further details, since that problem is beyond the scope of this thesis.

2.3.3 Query Execution Basics

Let us now outline some basic concepts used for in-network query processing. As we have shown in the previous section, new subscriptions are optimized with respect to the current state of the network and as a result of the optimization process a query plan is generated. It contains a set of queries constituting the original subscription and the peers at which

each of these has to be executed. This query plan is forwarded to the plan distribution component (PDC) of the peer executing the top-level query or operator, respectively, which in turn forwards subplans to the corresponding PDCs on other peers, if necessary, and so on. On each peer where new queries have been installed, the corresponding PDC triggers the execution of its part of the query plan to be executed on this peer at the query execution service of this peer.

Query execution in StreamGlobe focuses on processing streaming data and therefore employs *push-based* evaluation strategies—in contrast to traditional query engines where data is normally "pulled" from subordinate operators, e. g., by using the iterator model. This renders query execution different from executing queries over traditional data, because mostly main-memory techniques are needed. For the evaluation of queries a query execution service is able to invoke *stream operators*, which have to implement the StreamGlobe interface for stream operators, process an input stream, and generate an output stream. As suggested by Figure 2.7, two kinds of stream operators exist in StreamGlobe: *user-defined stream operators* implemented as mobile code and *built-in stream operators*.

User-defined stream operators provide a maximum amount of flexibility in data stream processing. They are typically employed for highly customized and specialized tasks or whenever the semantics of a subscription cannot be conveniently written in XQuery. Users implement such operators as mobile code in Java. Hence, the query engine service of any peer chosen by the optimizer to run such an operator is able to load this mobile code from any given location, to instantiate it, and to execute it. For instance, we have implemented some typical astronomical operators used in our demonstration scenarios like converting Celestial into Cartesian coordinates, special distance computations, and the so called *cone search*, which selects data contained in a cone-shaped area of the sky. For a seamless integration into StreamGlobe, subscriptions based on user-defined stream operators are also specified in XQuery using the syntax for external functions. At the moment, such an XQuery is only used for specifying how the input parameters of the user-defined operator are extracted from the data stream and how to construct the output data stream. It is not yet possible to register more complex subscriptions based on user-defined operators, e. g., join queries, subqueries, and so on.

Built-in stream operators constitute the set of operations available as the core functionality of StreamGlobe. These are mostly simple operators for managing data streams, e. g., send-/receive operators for forwarding data streams between peers and a display operator for outputting a data stream.

To be able to manipulate data streams by means of the queries computed by the optimizer, StreamGlobe has to evaluate those XQueries on XML data streams. Therefore, it provides our own streaming XQuery engine *FluX* as a built-in stream operator. FluX is a novel optimization and evaluation technique for minimizing memory buffer consumption during the execution of XQueries on streaming data. FluX consists of an intermediate language extending the XQuery syntax by event-based processing instructions which enables conscious handling of main memory buffers and an optimization algorithm for rewriting XQueries in this intermediate language. FluX enables query evaluation on data streams with very low memory consumption or does not even need any buffering at all. Hence,

it provides for a scalable evaluation of XQueries generated to compute subscription rules. However, some subscription rules might possibly need unbounded buffering, e. g., subscriptions containing joins or special (holistic) aggregates. In such cases, unbounded buffering is precluded by requiring users to specify window constraints. These allow for a scalable execution on infinite data streams. To further provide a maximum amount of flexibility in processing XQueries, the FluX query engine is extensible by means of user-defined functions. Similar to user-defined operators discussed at the beginning of this section, these user-defined functions are implemented as mobile Java code, which is loaded from any given location and used by the query engine. The difference between user-defined functions of the query engine and user-defined operators is their granularity: User-defined operators completely work on the entire data streams on their own. In contrast, user-defined functions of the query engine, e. g., special aggregate functions, only work on (a set of) atomic values to compute a single result. Details on the FluX query language, its optimizations, and an experimental evaluation will be presented in Chapter 3.

2.4 Further Example Scenarios

Having discussed the principles of subscription optimization and evaluation, we will present some further example scenarios in this section. For that purpose, we stay in the astrophysical domain and continue the example we have started in Section 2.1.

Astrophysical Alerter Service

Assume, that the second group of astronomers wants to realize some kind of an “alerter service”. Such a service will be a common application scenario in the field of astronomy in the future to detect interesting events and to trigger some action, e. g., automatically directing currently idling robotic telescopes to a certain part of the sky to join the observation for being able to collect more observational data. In our case, an alert should be generated, whenever the average value of the energies of photons of the RXJ0852.0-4622 Supernova Remnant in the last 60 seconds exceeds 1.3 keV. To realize this, this group registers a subscription “RXJ-alert” at peer P_2 in our exemplary network depicted in Figure 2.1. This subscription can be written in XQuery as

```
<photons>
  { for $w in stream("photons")/photons/photon
      [ra > 130.5 and ra < 135.5 and dec > -48.0 and dec < -45.0]
      |det_time diff 60 step 15|
    let $a := avg($w/photon/en)
    where $a > 1.3
    return
      <avg_energy> {$a} </avg_energy> }
</photons>
```

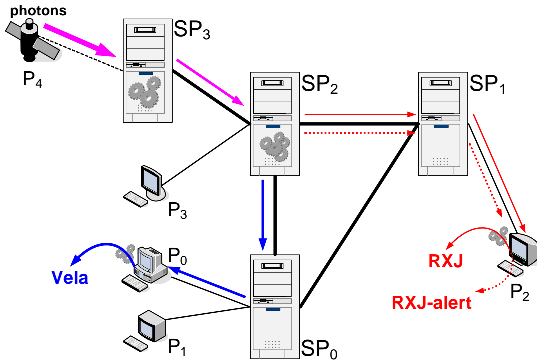


Figure 2.10: Integration of "RXJ-alert"

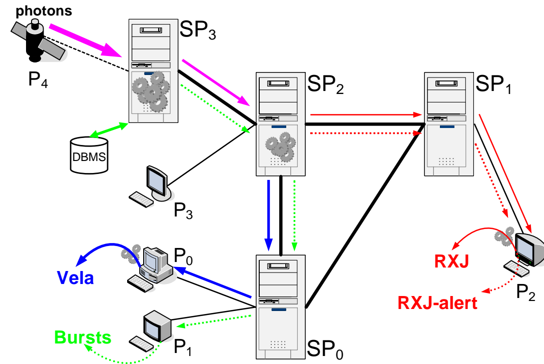


Figure 2.11: Integration of "Bursts"

The expression `|det_time diff 60 step 15|` is a part of our XQuery/XPath-extension to support window-based operators, which will be further discussed in Section 3.8.2. It specifies a time-based data window with respect to the values of the `det_time` element of the "photons" data stream. In detail, every 15 seconds an average value of the energies of the last 60 seconds is computed. The result is propagated to P_2 only if its value exceeds 1.3 keV. This subscription could be integrated into the network situation depicted in Figure 2.4 considering the optimization goals of the previous sections as follows. Because of the registered subscription "RXJ" P_2 already receives a part of the data needed for evaluating "RXJ-alert". In detail, all photons with `en` $\leq$ 1.3keV are filtered out by the selection predicate contained in "RXJ", but needed for "RXJ-alert". To also get those parts of the data stream, the optimizer could move the selection predicate `en` $\leq$ 1.3keV of the filter query for "RXJ" running on SP_2 to peer P_2 and execute "RXJ-alert" on P_2. This would yield an increased network traffic, because the benefit of early filtering the photon data stream for "RXJ" could not be exploited. The superior alternative is to execute "RXJ-alert" directly on SP_2. The data stream routed to SP_2 already contains all needed data and the additional network traffic of "RXJ-alert" is very small, since the selectivity of this subscription is very high and the aggregate compresses the data significantly. The selection of photons of the RXJ0852.0-4622 Supernova Remnant is identified as a common part of "RXJ" and "RXJ-alert" and only performed once at SP_2. The resulting network situation is depicted in Figure 2.10.

Combination of Streaming and Persistent Data

Another interesting and emerging application scenario in the field of astronomy is combining online observational data streams with persistent data of previous observations, e. g., stored in a DBMS. The goals of such an analysis are, e. g., to classify observed objects online using additional data of previous observations or to look for sudden changes ("bursts") in the energy of observed luminaries. To show how to realize such an application in StreamGlobe, we extend our example scenario such that super-peer SP_3 additionally hosts a DBMS in which all observational data of P_4 is persistently stored. Now, our first

group of astronomers shall be interested in detecting such bursts, i. e., sudden changes in the energy of online observed luminaries contained in the data stream "photons". Therefore, for each newly observed photon the difference of its current energy and that of previously observed photons in the same spot of the sky has to be computed. Since the measurement of the coordinates of photons is subject to small errors, we have to match the current photon with photons of previous observations, such that they most likely stem from the same luminary. Further, the time between the considered measurements should be approximately 90 minutes, which is the time the satellite needs for its orbit. To compute those pairs of measurements, i. e., photons, which best meet these requirements, we propose our novel *best-match join (BMJ) operator*, which will be explained in detail in Chapter 4. Therefore, astronomers of the first group are able to register the subscription "Bursts", which is phrased in XQuery as follows, at peer P_1.

```
<photons>
  { for $p1 in stream("photons")/photons/photon
      for $p2 in document("photons_db")/photons/photon
      where $p1 left outer bestmatch join $p2 on
        (fn:abs($p1/det_time - ($p2/det_time + 5400))) min 250,
        (fn:abs($p1/ra - $p2/ra)) min 0.1,
        (fn:abs($p1/dec - $p2/dec)) min 0.1
      return
        <rel_energy>
          {$p1/ra} {$p1/dec}
          {$p1/en - $p2/en}
        <rel_energy> }
</photons>
```

In this example, we utilize a left-outer best-match join (LOBMJ) to find best matching photons of previous observations for each newly observed photon. The `where`-clause of this subscription specifies the best-match join using our XQuery extensions for best-match joins. It contains the description of the join condition described above along with a specification of the maximum errors that are allowed in each comparison dimension for pairs being contained in the result. In detail, the measurement time must not differ more than 250 seconds from the given 90 minutes and the coordinates must not deviate more than 0.1 degree from each other. As explained in the previous sections, the optimal query plan for integrating the subscription "Bursts" into the existing network depends on the data volume of the input data stream "photons", the size of the computed data stream, and the current network situation. If the data volume of the result of this subscription is high compared to the data volume of the inputs, it is beneficial to execute "Bursts" as near as possible to peer P_1 to reduce the network costs. In our example, the optimizer is able to devise from statistics about the "photons" data stream and the persistent data, that such bursts are very seldom and hence propagation of the resulting data stream is neglectable compared to transferring the input data. Thus, it executes "Bursts" directly on super-peer SP_3, where all required data is already available. The resulting network situation is depicted in Figure 2.11.

In both examples, we have assumed that the peers chosen for executing the new subscriptions are capable of executing the computed queries and do not run into overload situations. If this would be the case, the peers at which (parts of) the subscriptions are executed would be chosen differently in order to prevent such overload situations. This might probably yield a sub-optimal network situation with respect to network traffic, but of course leads to a network being in a good state with respect to all optimization goals presented in Section 2.3.1.

2.5 Related Work

In the following, we present an overview of some work related to our StreamGlobe system. In particular, we deal with work in the fields of Peer-to-Peer data management, data stream management systems, multi-query optimization and execution, networking aspects, and Grid Computing. We will not present relevant work in the area of evaluating XPath/XQuery on data streams in this section, since this topic will be covered in detail in Section 3.10.

2.5.1 Peer-to-Peer Data Management

A number of relevant techniques and prototype systems have been developed in the Peer-to-Peer (P2P) context.

The most prominent P2P systems are file-sharing systems like Napster and Gnutella. Napster has been the first successful P2P system for exchanging files in a global scale. It stores all metadata on a central server, which is a vital drawback with respect to scalability of the network. To cope with this, Gnutella works without any centralized metadata service. Queries are propagated to neighbors and the results are "flowing" back to the client. The drawback of this approach is flooding of the network with queries and their results.

To efficiently support data retrieval various indexing techniques have been proposed, which are mostly based on distributed hash tables (DHTs). Some well known systems in this context are CAN [RFH+01], CHORD [SMK+01], Pastry [RD01], Tapestry [ZHS+04], and P-Grid [ACMD+03]. Hash-based techniques only support point queries, i. e., the retrieval of matching data objects (files), and are therefore not well suited for more complex applications such as distributed query processing that are targeted by StreamGlobe. However, some techniques can be adopted for StreamGlobe whenever point queries are needed, e. g., for retrieving metadata from other peers, to improve scalability and efficiency.

Beyond DHT-based indexing techniques various other topologies for P2P networks have been developed. In [YGM03] the concept of super-peer networks is introduced. These networks are meant to improve the scalability of P2P networks by using a super-peer backbone network. The super-peers usually are powerful servers. Less powerful, possibly mobile thin-peers can register and de-register themselves in the network via the super-peers. We employed the super-peer concept as an integral part of the StreamGlobe architecture because of its scalability and flexibility. In DHT-based networks the topology of the overlay

network, i. e., the neighborship relation of peers, is constructed with respect to the hash function and the content a peer serves. Another approach is the HyperCuP [SSDN02] topology using hypercubes as a network topology in P2P networks. It thereby achieves a logarithmic upper bound for the number of hops needed to get from one super-peer in the network to any other super-peer. Content-based construction of the overlay network might lead to the fact that neighbored peers are far from each other in the physical network. This renders such topologies not well suited for being used in StreamGlobe, which aims at optimizing network traffic. Attempts to construct an overlay network preserving the topology of the underlying physical network are proposed in [RHKS02].

DHT-based P2P networks are based on a global schema of the metadata describing the data objects. In contrast, schema-based P2P networks are able to deal with different schemata of metadata [BDK+03, ACMH03] or try to adopt queries to the different schemata at various peers, e. g., PIAZZA [TIM+03]. At the moment, StreamGlobe does not perform any schema mediation to answer subscriptions. However, this research area will be of importance in future work.

Other research efforts aim at building a peer database management system to enable transparent querying of a P2P system in the form of a traditional distributed DBMS. "Mutant Query Plans" [PMT03] implement distributed query processing at peers close to the data. PIER [HCH+05] is a distributed query engine based on structured overlay networks, which is intended to bring database query processing facilities to new, widely distributed environments. AmbientDB [FB04] aims at supporting non-trivial semantic multimedia retrieval queries. All these systems are common in that they share and enable querying of persistent data in P2P networks. Some optimization aspects are similar to those of StreamGlobe, e. g., pushing the execution of parts of queries into the network close to the data. In contrast to these systems, StreamGlobe aims at sharing continuous data streams and therefore has to consider additional optimization goals like optimizing network traffic in the P2P network.

2.5.2 Data Stream Management and Processing

With StreamGlobe being a system that handles and processes data streams, it is worthwhile to take a look at other recent approaches to building data stream management systems. The overview paper [BBD+02] presents general challenges and requirements for such data stream management systems.

STREAM [ABB+03, MWA+03, BW01] aims at constituting a comprehensive prototype data stream management system. It incorporates its own declarative query language CQL for continuous queries over data streams and relations. It handles streams by converting them into relations using special windowing operators and converting the query result back into a data stream if necessary. Up to now, STREAM only considers a centralized DSMS model, where all processing is performed on a single system.

Telegraph is a major project of the Berkeley Database Research group embracing various relevant sub-projects. TelegraphCQ [CCD+03, MSHR02] is a system to manage data streams and to adaptively process continuous queries over data streams. Currently, Tele-

graphCQ is a centralized DSMS, but a distributed implementation is planned for future work. PSoup [CF03] combines the processing of ad-hoc and continuous queries by treating data and queries symmetrically, allowing new queries to be applied to old data and new data to be applied to old queries.

Aurora [CÇC+02] is a new DBMS for monitoring applications and implements a centralized stream processor for dealing with streaming data. In [CBB+03] two complementary large-scale distributed stream processing systems, Aurora* and Medusa, are described. Aurora* is a distributed version of Aurora with nodes belonging to a common administrative domain. Medusa, on the other hand, supports the federated operation of several Aurora nodes across administrative boundaries. Aurora, Aurora*, and Medusa have been superseded by Borealis [AAB+05], which is a distributed multi-processor version of Aurora built upon the techniques of Aurora* and Medusa. Its current focus is on QoS management, load distribution, high availability, and fault tolerance in data stream processing.

Above mentioned systems aim at constituting a complete DSMS, but up to now—more or less—focus on special aspects of (adaptive) query processing, load balancing, or quality-of-service management. The major contribution of StreamGlobe compared to these systems is that it does not only efficiently locate and query data streams, but explicitly addresses the optimization of the data flow within the network by employing in-network query processing capabilities.

PIPES [KS04] aims at being a flexible and extensible infrastructure providing fundamental building blocks to implement a data stream management system. PIPES also contains separate frameworks that establish a basis for other essential runtime components, namely the scheduler, the memory manager, and the query optimizer.

Naturally, data streams play an important role in the area of sensor networks. The Cougar Sensor Database Project [YG02, YG03] tasks sensor networks through declarative queries. A query optimizer generates an efficient query plan for in-network query processing, which reduces resource usage and thus extends the lifetime of a sensor network. As a part of the Telegraph project, the Fjords [MF02] architecture aims at managing multiple queries over many sensors and shows how it can be used to limit sensor resource demands while maintaining high query throughput. These systems are designed to handle sensor networks and their special needs. StreamGlobe is well-suited to manage such sensor networks in terms of a general, distributed data stream system. It already natively provides optimizations needed in sensor networks, e. g., reducing network traffic to conserve energy, reuse of operators to minimize computing effort, and the execution of queries on devices with respect to their capabilities. Further special optimizations needed for sensor networks can easily be integrated into the subscription optimization process.

2.5.3 Multi-Query Optimization and Execution

With respect to query execution, work in the field of multi-query optimization is related to StreamGlobe. Since query optimization in StreamGlobe is not the main scope of this thesis, we will only briefly show the most important relevant work in this area.

Multi-query optimization has been addressed in [Sel88]. It pursues the goal of processing

multiple queries all at once instead of one query at a time. The main optimization potential lies in the fact that queries may share a considerable amount of common—or at least similar—parts of subscriptions that can be reused for more than one query. Obviously, StreamGlobe in general has to deal with a set of queries simultaneously, thus rendering multi-query optimization an applicable and suitable optimization approach.

StreamGlobe uses data stream sharing techniques to identify reusable existing data streams in the network that fit newly registered queries. This approach has similarly been applied in the world of persistent data where view materialization and view selection are used to improve the efficiency of query processing [LMSS95]. In [YKL97], further algorithms for solving the view materialization problem are devised. Materialized view selection and maintenance have also been examined using techniques of multi-query optimization [MRSR01]. The query containment problem in the context of XML queries, which is also relevant for multi-query optimization, has been addressed in [TH04].

Queries in StreamGlobe are usually continuous queries over data streams. A high level of scalability in continuous query processing intends NiagaraCQ [CDTW00] to achieve by grouping continuous queries according to similar structures. In StreamGlobe, we employ similar multi-query optimization approaches by extracting common parts of subscriptions to reduce network traffic and to enable efficient query evaluation.

Much work on multi-query execution has been done in the field of Selective Dissemination of Information or XML Message Brokering. XFilter [AF00] transforms XPath-Queries into finite automata and indexes their states to efficiently check what paths are matched. YFilter [DFFT02, DF03] targets on-the-fly matching of XML data to interest specifications written in a subset of XQuery, and transformation of the matching XML data based on recipient-specific requirements. It combines all path expressions into a single nondeterministic finite automaton where the common prefixes among path expressions are represented only once. When the automaton execution reaches an accepting state, a path match is output for all path expressions represented by that accepting state. ONYX [DRF04] constitutes a distributed XML Dissemination Service based on YFilter. XTrie [CFGR02] is an index structure based on tries, that supports the efficient filtering of streaming XML documents based on XPath expressions. In contrast to StreamGlobe, where subscriptions are “real” queries enabling transformation of input data, most of these systems treat subscriptions as boolean queries, i. e., it is only checked whether a query matches input data or not. Hence, these techniques cannot directly be applied to query execution in StreamGlobe. Nevertheless, some aspects of these approaches are also relevant for multi-query optimization and execution in StreamGlobe, e. g., for computing common filter queries for a set of subscriptions.

2.5.4 Networking Aspects

Multicast techniques route data towards receiving ends in a way that reduces network traffic by transmitting the same message or document only once for all recipients instead of multiple times, once for each recipient. Such a multicast technique is provided by the TCP/IP networking layer, but is mainly used in local area networks. Various works

have been done to extend TCP/IP routing capabilities to enable multicast over wide area networks, e. g. in [DC90]. Especially in the context of ad-hoc and sensor networks some work has been done, e. g., to efficiently enable multicast in evolving networks [HLR03].

It is important to point out that our work differs from these approaches in a major way. Multicast techniques as mentioned above work mostly on a network level, i. e., they prevent the redundant transmission of equal data packets of a data stream. This is sufficient for, e. g., video streams, since such streams always have the same content for each recipient. In the context of StreamGlobe this assumption does not hold, since an original data stream may have various more or less different instances. Hence, techniques working on the network level do not achieve satisfying improvements with respect to network traffic. Instead, StreamGlobe provides multicast techniques on an application or content-based level by means of data stream sharing. Instead of only reusing existing messages or documents, our system is able to perform in-network transformations on data streams. Therefore, it can dynamically create appropriate data streams for data stream sharing that fit the queries to be answered best while at the same time reducing network traffic.

2.5.5 Grid Computing and E-Science Applications

StreamGlobe builds on and extends the Open Grid Services Architecture and its reference implementation, the Globus Toolkit [Glo04] by adding data stream management and data stream processing capabilities to the Grid Computing domain.

A related approach, also building on Globus, is GATES [CRA04]. However, this alternative approach concentrates mainly on data stream analysis and quality-of-service aspects in data stream delivery, whereas we primarily focus on self-organization, distributed in-network query processing, and optimization.

Another system building on the Open Grid Services Architecture is OGSA-DAI (Open Grid Services Architecture Data Access and Integration) [OGS04]. As the name suggests, this project is concerned with constructing a middleware to enable the access and integration of data from distributed data sources via the Grid. It also contains a distributed query processor called OGSA-DQP. In contrast to StreamGlobe, OGSA-DAI has no special focus on data streams.

Recent efforts in applying database and Grid Computing techniques to e-science applications in data-intensive fields, e. g., astronomy, high-energy physics, or genetics, have gained much attention. In [NSGS+05] it is demonstrated how the astronomical problem of finding galaxy clusters can be improved by magnitudes employing database techniques. GridDB [LF04] is a software overlay that provides data-centric services for scientific Grid Computing. These examples prove the demand for efficient management and processing systems for huge data volumes in future e-science applications, which StreamGlobe targets as one of its main application scenario.

2.6 Discussion

In this chapter, we have described the basic architecture and goals of our StreamGlobe data stream management system. StreamGlobe is focused on meeting the challenges that arise in processing data streams in an (ad-hoc) P2P network scenario. It differs from other data stream systems in not only efficiently locating and querying data streams, but also optimizing the data flow in the network using expressive in-network query processing techniques. This is basically achieved by pushing operators for query processing into the network and data stream sharing. Continuous re-optimization leads to an adaptive and self-optimizing system which enables users to carry out powerful information processing and retrieval tasks. StreamGlobe builds on and extends the Globus Toolkit, a reference implementation of the Open Grid Services Architecture for Grid Computing, and serves as a research platform for our future work.

Future research will cover further topics in query processing on streaming data, optimization methods for distributed data stream processing, load balancing, and quality-of-service aspects [BKK03] in a distributed data stream management system. In detail, this will comprise improving the optimization component by taking into account reorganization issues to keep the system effective as well as synchronization aspects, e. g. for distributed join processing on various streaming inputs. We will investigate more sophisticated algorithms for the—in general—NP-hard problem of predicate comparisons in the context of selecting data streams being suitable for performing data stream sharing. Furthermore, we will continue to examine routing approaches and well-suited network topologies for our hierarchical network organization. Another interesting aspect will be support for content-based query subscriptions.

Chapter 3

The *FluX* Streaming XQuery Processor

In this chapter our streaming XQuery processor *FluX*, which is employed for subscription evaluation in StreamGlobe, is presented. We introduce an extension of the XQuery language that supports event-based query processing and the conscious handling of main memory buffers. Purely event-based queries of this language can be executed on streaming XML data in a very direct way. We then develop an algorithm that allows to efficiently rewrite XQueries into the event-based FluX language. This algorithm uses order constraints from a DTD to schedule event-handlers and to thus minimize the amount of buffering required for evaluating a query. Further, various technical aspects of query optimization and query evaluation within our framework are discussed. This is complemented with an experimental evaluation of our approach.

This work evolved from a collaboration with Prof. Dr. C. Koch (TU Wien/Universität des Saarlandes) and Prof. Dr. N. Schweikardt (HU Berlin). Parts of this chapter have been presented [KSSS04c] at the *International Conference on Very Large Databases 2004* in conjunction with a demonstration of our prototype of the FluX query engine [KSSS04a]. An extended version of [KSSS04c] has been published in [KSSS04b].

This chapter is structured as follows. At first, we motivate our approach for processing XQueries on streaming data. In Sections 3.2 and 3.3 we start with basics on DTDs, regular languages, schema constraints, and checking of the constraints. Section 3.4 defines the query language considered in this chapter. Section 3.5 presents our algorithm for translating XQuery into FluX. In Section 3.6 we show additional algebraic optimizations which exploit DTD knowledge and discuss implementation aspects of our prototype system in Section 3.7. We show principles for extending the core FluX query engine in Section 3.8 considering aggregations and data windows as examples. In Section 3.9 we present our experiments conducted using our prototype implementation. We conclude with related work in Section 3.10 and a discussion in Section 3.11.

3.1 Motivation

XML is the preeminent data exchange format not only in our StreamGlobe system, but generally on the Internet. Stream processing naturally bears relevance in the data exchange context (e. g., in e-commerce). An increasingly important data management scenario is the processing of XQueries on streams of exchanged XML data. While the weaknesses of XML as a semistructured data model have been observed time and again (cf. e. g. [ABS00]), XQuery on XML streams can be seen as the prototypical instance of the problem of queries on *structured* (vs. flat tuple) *data streams*.

Query engines for processing streams are naturally main-memory-based. Conversely, in some efforts towards developing main-memory XQuery engines whose original emphasis was *not* on stream processing (e. g., BEA's XQRL [FHK+03]), it was observed that it is worthwhile to build such systems using stream processing operators. The often excessive need for buffers in current main memory query engines causes a scalability issue that has been identified as a significant research challenge [ABB+02]. While the efficient evaluation of XPath queries on streams has been worked on extensively in the past (here, state-of-the-art techniques use very little main memory), not much work has been done on efficiently processing XQuery on streams. The nature of XQuery, as a *data-transformation* query language entirely different from *node-selecting* XPath, requires new techniques for dealing with (and reducing) main memory buffers. State-of-the-art XQuery engines consume main memory in large multiples of the actual size of input XML documents [MS03].

An important goal is thus to devise a well-principled machinery for processing XQuery that is parsimonious with resources and allows to minimize the amount of buffering. Such machinery needs to be based on intermediate representations of queries that are syntactically close to XQuery and has to allow for an algebraic approach to query optimization, with buffering as an optimization target. This is necessary to allow for both extensibility and the leverage of a large body of related earlier work done by the database research community. However, to our knowledge, no principled work exists on query optimization in the framework of XQuery (rather than automata) for *structured data streams* (such as XML, but unlike flat tuple streams) which honors the special features of stream processing. Moreover, no framework for optimizing queries on structured data streams exists that captures the spirit of stream processing and allows for query optimization using schema information. However, there are XQuery algebras meant for conventional query processing, and there is some work on applying them in the streaming context[1].

In this chapter, we attempt to improve on this situation. We introduce the *FluX* query language, which extends XQuery by a new construct for event-based query processing called `process-stream`. FluX motivates a very direct mode of query evaluation on data streams (similar to query evaluation in XQRL), and provides a strong intuition for what main memory buffers are needed in which queries. This allows for a strongly "buffer-conscious" mode of query optimization. The main focus of this section is on automatically rewriting XQueries into event-based FluX queries and at the same time optimizing (reducing) the

[1]For details, see Section 3.10.

use of buffers using schema information from a DTD.

To motivate our approach, we leave the astrophysical domain used in the previous chapter and utilize a bibliography domain known from the XML Query Use Cases [W3C05a]. Consider the following XQuery Q (XMP-Q3) as an example:

```
<results>
  { for $b in $ROOT/bib/book
    return
      <result>
        {$b/title}
        {$b/author}
      </result> }
</results>
```

For each book in the bibliography, this query lists its title(s) and authors, grouped inside a `result` element. Note that the XQuery language requires that, within each book, in the result all titles are output before all authors.

Consider the DTD

```
<!ELEMENT bib    (book)*>
<!ELEMENT book   (title|author)*>
<!ELEMENT title  (#PCDATA)>
<!ELEMENT author (#PCDATA)>
```

specifying that each `book` node may have several `title` and several `author` children[2]. A priori, no order among these items is inferable from the given DTD. To implement this query, we may output the `title` children inside a `book` node as soon as they arrive on the stream. However, the output of the `author` children needs to be delayed (using a memory buffer) until we reach the closing tag of the `book` node (at that time, no further `title` nodes may be encountered). Then, we may flush the buffer of `author` nodes to the output, empty it, and later refill it with the `author` nodes from the next `book` node.

We thus only need to buffer the `author` children of one `book` node at a time, but not the titles. Current main memory query engines do not exploit this fact, and rather buffer either the entire book nodes or, as an optimization [MS03], only all `title` and all `author` nodes of a `book` node. Previous frameworks for evaluating or optimizing XQuery do not provide any means of making this seeming subtlety explicit and reasoning about it.

The `process-stream` construct of FluX allows to express precisely the mode of query

[2]For the sake of readability, we will omit terminals of the DTD similar to `<!ELEMENT title (#PCDATA)>` in the remainder.

execution just described. XQuery Q is then phrased as a FluX query[3] as follows:

```
<results>
{ process-stream $ROOT:
  on bib as $bib return
    { process-stream $bib:
      on book as $book return
        <result>
        { process-stream $book:
          on title as $t return {$t};
          on-first past(title, author) return
            { for $a in $book/author
              return {$a} }; }
        </result>; }; }
</results>
```

A "`process-stream $x`"-expression consists of a number of *event-handlers* which process the children of the XML tree node bound by variable `$x` from left to right (on the stream, from start to end). An "`on a ... return` α"-handler fires on each child labeled "`a`" visited during such a traversal, executing the associated query expression α. In the "`process-stream $book`"-expression above, the "`on-first past(title, author)`"-handler fires exactly once as soon as the DTD implies for the first time that no further `author` *or* `title` node can be encountered among the children of the node currently bound to `$book`. As observed above, in the given, very weak DTD, this is the case only as soon as the last child of `$book` has been seen. In the query associated with the "`on-first past(title, author)`"-handler, we may freely use paths of the form "`$book/author`" or "`$book/title`", because such paths cannot be encountered anymore and we may assume that the query engine has already buffered all matches of these paths for us. It is a feasible task for the query engine to buffer only those paths that the query actually employs (see also [MS03]).

We call a query *safe* for a given DTD if, informally, it is guaranteed that XQuery subexpressions (such as the `for`-loop in the query above) do not refer to paths that may still be encountered in the stream. The above FluX query is safe: The `for`-expression employs the "`$book/author`" path, but is part of an "on-first"-handler that cannot fire before all `author` nodes relative to `$book` have been seen. If the path "`$book/author`" was replaced by, e. g., "`$book/price`" and the DTD production for `book` were

```
<!ELEMENT book   ((title|author)*, price)>
```

then the FluX query above would not be safe. In that case, on the firing of "`on-first past(title, author)`", the buffer for "`$book/price`" items would still be empty and the query result would be incorrect.

[3]Here, we employ a slight generalization of the FluX syntax as defined later on—expressions of the form $\langle t\rangle\alpha\langle/t\rangle$, where α is a "`process-stream`"-expression—to improve readability.

Query Q can be processed more efficiently with the schema used in the XML Query Use Cases, which is

```
<!ELEMENT bib  (book)*>
<!ELEMENT book (title, (author+|editor+), publisher, price)>
```

Here, no buffering is required to execute our query, because the DTD asserts that for each book, the title occurs strictly before the authors. We denote this situation as $\mathit{Ord}_{\texttt{book}}(\texttt{title}, \texttt{author})$, called an *order constraint*. Hence, we may phrase our query in FluX so as to directly copy titles and authors to the output as they arrive on the input stream as follows:

```
<results>
{ process-stream $ROOT:
  on bib as $bib return
    { process-stream $bib:
      on book as $book return
        <result>
        { process-stream $book:
          on title  as $t return {$t};
          on author as $a return {$a}; }
        </result> }; }
</results>
```

In this case, no data items need to be buffered during the execution of this FluX query. Hence, the evaluation of this query using our optimizations enables for scalable execution of XQueries on data streams.

This is, to the best of our knowledge, the first work on optimizing XQuery using schema constraints derived from DTDs[4]. A main strength of the approach taken in this work is its extensibility. Even though we restrict our discussion to a (powerful) *fragment* of XQuery, our results can be generalized to even larger fragments. Some principles for extending the XQuery fragment will be shown in Section 3.8.

3.2 Preliminaries

In this section, we introduce our data model, some basics about DTDs, and define schema constraints needed for the remainder of this chapter.

For simplicity of exposition, we consider the fragment of XML without attributes as our data model. Note that this is no substantial restriction, since attributes can be handled in the same way as subelements. Moreover, we focus on *valid* documents, i. e., documents conforming to a given DTD. The input XML stream is processed by a SAX parser generating corresponding events upon reading opening tags, character data, and closing tags.

[4]To simplify presentation we restrict ourselves to DTDs, but the required information could also be derived from XML Schemata [W3C04b].

Let Σ be a set of symbols (or tag names). A *document type definition* (DTD) is an extended context free grammar defined for a set of element names Σ and special symbols `PCDATA` and `EMPTY`, with productions $p \in P$ of the form

$$p \ \texttt{::=}\ \texttt{<!ELEMENT}\ e^p \ \ \rho^p\texttt{>}$$

where $e^p \in \Sigma$ and ρ^p is either `EMPTY` or a *one-unambiguous* regular expression over $\Sigma \cup \{\texttt{PCDATA}\}$. Furthermore, DTDs are *local* tree grammars [MLM01], i. e. without competing nonterminals to the left-hand sides of productions, so each production in a DTD is unambiguously identified by a tag name in Σ. By $L(\rho)$ we denote the language defined by ρ, i. e., the set of words over $symb(\rho)$ that are recognizable by ρ.

Definition 3.2.1 (Order Constraint) *Let ρ be a regular expression and let $symb(\rho)$ be the set of atomic symbols that occur in ρ. Given a word w, let w_i denote its i-th symbol. We define a binary relation $Ord_\rho \subseteq symb(\rho) \times symb(\rho)$ such that for $a, b \in symb(\rho)$,*

$$Ord_\rho(a, b) \ :\Leftrightarrow\ \nexists w \in L(\rho) : w_i = b \wedge w_j = a \wedge i < j.$$

Then, a constraint of the form $Ord_\rho(a, b)$ is defined as an order constraint.

That is, $Ord_\rho(a, b)$ holds if there is no word in $L(\rho)$ in which a symbol a is preceded by a symbol b. In other words, all a symbols occur before all b symbols.

Example 3.2.2 *Let*

$$\rho = (a^*.b.c^*.(d|e^*).a^*)$$

Then, $Ord_\rho(b, c)$, $Ord_\rho(c, d)$, and $Ord_\rho(c, e)$, but $\neg Ord_\rho(a, c)$. Ord_ρ is transitive, so we also have e. g. $Ord_\rho(b, d)$. □

Besides order constraints, we are able to infer constraints regarding the cardinality of elements from a DTD.

Definition 3.2.3 (Cardinality Constraints) *Let ρ be a regular expression. We define* cardinality constraints *as sets $||^0_\rho$, $||^{\leq 1}_\rho$, $||^{=1}_\rho \subseteq \Sigma$ such that for $a \in \Sigma$,*

$$\begin{aligned} a \in ||^0_\rho \ &:\Leftrightarrow\ a \notin symb(\rho), \\ a \in ||^{\leq 1}_\rho \ &:\Leftrightarrow\ \forall w \in L(\rho) :\ |\{i \mid w_i = a\}| \leq 1, \\ a \in ||^{=1}_\rho \ &:\Leftrightarrow\ \forall w \in L(\rho) :\ |\{i \mid w_i = a\}| = 1. \end{aligned}$$

That is, for a symbol $a \in \Sigma$, the order constraint $a \in ||^0_\rho$ ($a \in ||^{\leq 1}_\rho$ or $a \in ||^{=1}_\rho$, respectively) holds, if the symbol a does not occur (occurs at most once or occurs exactly once, respectively) in any word of $L(\rho)$.

Example 3.2.4 *As before, let*

$$\rho = (a^*.b.c^*.(d|e^*).a^*)$$

Then, $||^{\leq 1}_\rho = \{b, d\}$ and $||^{=1}_\rho = \{b\}$. □

3.3 Efficient Checking of Schema Constraints

In this section, we present how to efficiently compute order constraints and cardinality constraints derived from DTDs.

First, we introduce additional notation. Let ρ be the regular expression contained in a single production of a DTD as outlined in the previous section. By a *marking* of a regular expression ρ we denote a regular expression ρ' such that each occurrence of an atomic symbol in ρ is replaced by the symbol with its position among the atomic symbols of ρ added as a subscript. That is, a symbol $a \in symb(\rho)$ at the i-th position of ρ is replaced by a_i. The reverse of a marking (indicated by #) is obtained by dropping the subscripts.

Example 3.3.1 *Consider again the regular expression*

$$\rho = (a^*.b.c^*.(d|e^*).a^*)$$

The marking ρ' of ρ is

$$\rho' = (a_1^*.b_2.c_3^*.(d_4|e_5^*).a_6^*),$$

and, e. g., $a_1^\# = a_6^\# = a$. □

As stated before, all regular expressions in a DTD are one-unambiguous [BKW98]. Intuitively, a one-unambiguous regular expression ρ allows for deterministic matching of a word $w \in L(\rho)$ using only a one-token lookahead.

For each one-unambiguous regular expression, an equivalent deterministic finite automaton called the Glushkov automaton can be constructed. Glushkov automata have the characteristic properties that

1. each state in a Glushkov automaton (apart from the initial state) corresponds to a symbol in the marked regular expression, and
2. each transition $\delta(q, a) = p$ into a state p takes place under input symbol $a = p^\#$.

We refer to [BKW98] for a formal definition of one-unambiguous regular expressions, of Glushkov automata, and their construction. Glushkov automata for one-unambiguous regular expressions can be constructed efficiently in quadratic time [BKW98]. In the context of document grammars for SGML (and thus, XML) documents Glushkov automata can even be constructed in linear time [BK93].

As mentioned in the introduction, we have to determine when certain elements cannot be encountered anymore, which might only be the case as soon as we see the closing tag of the current production. For instance, in the first DTD of the introductory example, we are only able to ensure that no further titles or authors can be encountered upon seeing the closing tag `</book>` of the current `book` element. To be able to handle such situations, we introduce the symbol "$\odot$" which corresponds to the closing tag of the current production ρ. With this, we define $\overline{\rho} := \rho.\odot$ by appending "$\odot$" to a given production ρ. Intuitively, the symbol "$\odot$" is used as some kind of "end marker" in ρ.

Let $\mathcal{G}(\overline{\rho}) = (Q,\ symb(\overline{\rho}'),\ \delta,\ q_0,\ F)$ be a Glushkov automaton for $\overline{\rho}$ with $Q = \{q_0, \dots, q_n\}$ and $F \subseteq Q$. Further, let $\delta^*(q,\ a_1 \dots a_n) := \delta(\dots\delta(\delta(q, a_1), a_2), \dots, a_n)$. Therewith, let the reachability relation Δ be

$$\Delta := \{\langle q_i, q_j\rangle \mid \exists u \in symb(\overline{\rho}')^* : \delta^*(q_i,\ u) = q_j\}.$$

Obviously, Δ can be computed in time $O(|Q|^2)$ by simply, for each $q \in Q$, computing the reachable states in the transition graph of $\mathcal{G}$ (for which there is a well-known linear time algorithm [Tar72]).

Definition 3.3.2 (Past Set) *Let ρ be the regular expression for a production of the DTD, $\mathcal{G}(\overline{\rho})$ its Glushkov automaton, and $a \in symb(\overline{\rho})$. The relation $Past_\rho \subseteq Q \times \Sigma$ is defined as*

$$Past_\rho(q_i, a) \Leftrightarrow \nexists q_j : q_j^\# = a \wedge \langle q_i, q_j\rangle \in \Delta.$$

Intuitively, $Past_\rho(q_i, a)$ means that on reaching state q_i, we are past all occurrences of "a". That is, we may not encounter "a" anymore until we reach the end of the word, otherwise it is not in $L(\rho)$. Obviously, this relation can be obtained from Δ in time $O(|Q|^2)$. Note that $^\#$ is functional and imposes a partition of Q.

Now, we can define order constraints as

$$Ord_\rho(a, b) \ \Leftrightarrow\ \forall q : (q^\# = b) \rightarrow Past_\rho(q, a).$$

It is easy to see that the relation Ord_ρ can be computed in time $O(|Q| \cdot |symb(\rho)|)$.

Beyond computing order constraints, for the evaluation of FluX queries we have to generate events, called *"on-first-past"-events*, which indicate that no atomic symbol of a given set $S \subseteq \Sigma$ can be encountered anymore in the stream of the current production. To generate "on-first-past"-events, we validate each token read from the input stream by simulating a transition of the Glushkov automaton associated with the current DTD production. We can pre-compute a table

$$PastTable_{\rho,S}(q) \Leftrightarrow \forall a \in S : Past_\rho(q, a)$$

which denotes whether symbols of the set S in the current production ρ cannot be encountered in the stream anymore if the Glushkov automaton is currently in the state q. Since we are only interested in the first occurrence of this situation, we therefore additionally define a Boolean function $\textit{first-past}_{\rho,S}$. For each transition of the Glushkov automaton, we may compute $\textit{first-past}_{\rho,S}$ by a constant-time lookup in *PastTable*. More precisely, we compute $\textit{first-past}_{\rho,S}(u_1 \dots u_i)$ as follows. Initially,

$$\textit{first-past}_{\rho,S}(\epsilon) := PastTable_{\rho,S}(q_0).$$

On making the transition $\delta(q, u_i) = q'$ where $q = \delta^*(q_0, u_1 \dots u_{i-1})$,

$$\textit{first-past}_{\rho,S}(u_1 \dots u_i) := PastTable_{\rho,S}(\delta(q, u_i)) \ \wedge\ \neg PastTable_{\rho,S}(q).$$

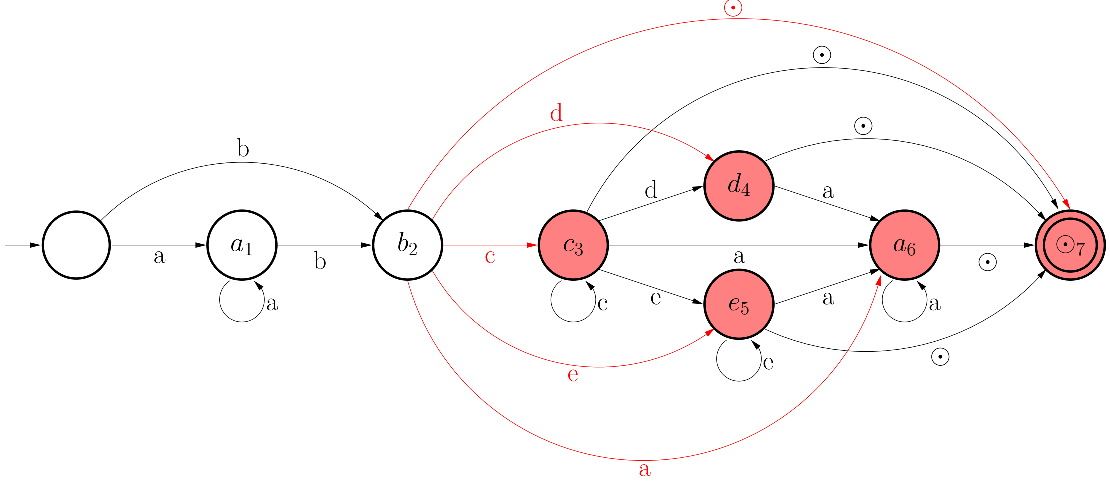


Figure 3.1: Glushkov Automaton for $\rho = (a^*.b.c^*.(d|e^*).a^*)$ and $S = \{b\}$

Intuitively, when processing a word $uw \in L(\rho)$ from left to right, if *first-past*$_{\rho,S}(u)$ holds, then the reading of the last symbol of u is the earliest possible time at which we know that none of the symbols in S can be seen anymore until the end of the word uw. Thus the SAX parser generates "on-first-past"-events (which fire when *first-past*$_{\rho,S}$ becomes true) in addition to traditional SAX events with very little overhead, namely one validating DFA transition and one constant-time lookup per input token read.

Example 3.3.3 *We continue with the regular expression ρ and its marking ρ' we have started in Example 3.3.1. Figure 3.1 shows the Glushkov automaton $\mathcal{G}(\overline{\rho})$ for ρ constructed as described above. Let $S = \{b\}$. Then, $Past_\rho(q, b)$ holds for the states $q \in \{c_3, d_4, e_5, a_6, \odot_7\}$, which are marked red in the figure. first-past$_{\rho,S}$ is true for all transitions of $\mathcal{G}(\overline{\rho})$ which start in a state not in $Past_\rho$ and end in a state contained in $Past_\rho$. Again, these transitions are marked red. Whenever such a transition is simulated, we fire an "on-first-past"-event indicating that no "b" can be seen while processing the production ρ anymore.* □

Note that instead of performing a lookup in *PastTable* for each transition to determine whether to fire an "on-first-past"-event or not, we may annotate the events at the corresponding transitions of the automata. Also note that on making a single transition we might have to fire more than one "on-first-past"-event (for different sets S of symbols). In this case, all "on-first-past"-events are thrown simultaneously.

Cardinality constraints can be computed efficiently as follows.

Definition 3.3.4 (Ahead Set) *Let ρ be the regular expression for a production of the DTD, $\mathcal{G}(\overline{\rho})$ the Glushkov automaton of ρ, and $a \in symb(\overline{\rho})$. Analogously to $Past_\rho$, we define the relation $Ahead_\rho \subseteq Q \times \Sigma$ as*

$$Ahead_\rho(q_j, a) \Leftrightarrow \nexists q_i : q_i^\# = a \wedge \langle q_i, q_j \rangle \in \Delta.$$

That is, $Ahead_\rho(q, a)$ holds iff for no $u \in \Sigma$ such that $\delta^*(q_0, u) = q$, there is an i with $u_i = a$.

We compute cardinality constraints as

$$\begin{array}{rcl} a \in ||_\rho^{\leq 1} & \Leftrightarrow & \exists q_i, q_j : Ahead_\rho(q_i, a) \wedge (\delta(q_i, a) = q_j \vee q_i = q_j) \wedge Past_\rho(q_j, a) \\ a \in ||_\rho^{=1} & \Leftrightarrow & L(\rho) \subseteq L(\text{precisely one } a) \\ & \Leftrightarrow & L(\rho) \cap L(\text{not precisely one } a) = \emptyset \\ & \Leftrightarrow & L(\rho) \cap L((\Sigma \backslash \{a\})^* \mid \Sigma^*.a.\Sigma^*.a.\Sigma^*) = \emptyset. \end{array}$$

The relation $||_\rho^{\leq 1}$ can be be computed in time $O(|Q| \cdot |symb(\rho)|)$, where we use the fact that δ is functional. The relation $||_\rho^{=1}$ can be computed in time $O(|\mathcal{G}| \cdot |\Sigma|)$. The "$\cap$" construction is that of a product automaton of $\mathcal{G}$ with a *constant* automaton, and the emptiness test is feasible in time linear in the size of the automaton produced.

3.4 Query Language

In Section 3.4.1 we specify the XQuery fragment supported by FluX. Based on this fragment, Section 3.4.2 defines the syntax and semantics of the FluX query language, and Section 3.4.3 singles out the safe FluX queries.

Before defining the XQuery fragment we support and our FluX language, we need some more notation. We write $\$x$, $\$y$, $\$z$, ... to denote variables that range over XML trees. In the following, we overload the meaning of a variable $\$x$ bound to an XML tree whose root is labeled a, by writing $\$x$ when we actually mean the DTD production unambiguously identified by the element a. For example, if the DTD contains the rule `<!ELEMENT` a ρ^a`>` for a regular expression ρ^a, we write $Ord_{\$x}(c, d)$ instead of $Ord_{\rho^a}(c, d)$, and we write $symb(\$x)$ instead of $symb(\rho^a)$.

Definition 3.4.1 (Fixed Path) *A* fixed path *is a sequence $a_1/\ldots/a_n$, where the a_i are symbols from the DTD and $n \geq 1$.*

XPath expressions such as $a/*/b$, $a//b$, or $a[\chi]$ with χ being some conditional expression are no fixed paths and not considered in this work.

Definition 3.4.2 (Value Expression) *A* value expression *is of the form*

- s
- $\$x/\pi$
- v *ValOp* v'

with s being a string, π being some fixed path, v and v' being value expressions, and $ValOp \in \{+, -, *, \texttt{div}, \texttt{idiv}, \texttt{mod}\}$.

Definition 3.4.3 (Atomic Condition) *An* atomic condition *is of the form*

- `fn:true()`
- `fn:false()`
- v *RelOp* v',
- `fn:exists`($\$x/\pi$)
- `fn:empty`($\$x/\pi$)

with v *and* v' *being value expressions,* π *being a fixed path, and RelOp* $\in \{=, !=, <, \leq, >, \geq\}$.

Definition 3.4.4 (Condition) *A* condition *is a Boolean combination using "*`and`*", "*`or`*", and "*`fn:not`*" of atomic conditions.*

Note that for specifying (Boolean) functions, e. g., "exists" or "not", and Boolean values, i. e., "true" and "false", we adhere to the syntax as defined in [W3C05d].

3.4.1 An XQuery Fragment: XQuery$^-$

With the above definitions we are able to define the XQuery$^-$ fragment of XQuery that we support with our FluX query engine.

Definition 3.4.5 (XQuery$^-$) *The XQuery fragment XQuery$^-$ is the smallest set consisting of expressions*

- ϵ *(the empty query)*
- s *(output of a fixed string)*
- α β *(sequence)*
- { `for` $\$x$ `in` $\$y/\pi$ `return` α } *(for-loop)*
- { `for` $\$x$ `in` $\$y/\pi$ `where` χ `return` α } *(conditional for-loop)*
- { $\$x/\pi$ } *(output of subtrees reachable from node $\$x$ through path π)*
- { $\$x$ } *(output of subtree of node $\$x$)*
- { `if` (χ) `then` α } *(conditional)*

where π *is a fixed path,* s *a fixed string,* χ *a condition, and* α*,* β *are XQuery$^-$ expressions.*

Indeed, XQuery$^-$ is very similar to (a fragment of) standard XQuery [W3C05c], but slightly differs in semantical and syntactical aspects.

The first is how we treat fixed strings inside queries. For example, the string "`<hello>`" is valid in XQuery$^-$, but not in standard XQuery. The query

```
<result> {/bib/book} </result>
```

is understood in standard XQuery as a `result` node, denoted as an *element constructor*, with an embedded query to produce its children. In this work, the same query is read as a sequence of three queries which write the string "`<result>`", the "`/bib/book`" subtrees,

and finally the string "</result>" to the output. This, however, is only a subtlety which, on the one hand, is very convenient for obtaining our main results in Section 3.5 and which, on the other hand, as the following Proposition 3.4.7 shows, does not cause any problems. The alternative semantics of XQuery$^-$ is the basis of optimizations used *internally* by the query engine. Users formulate input queries in standard XQuery and may assume the usual semantics.

Apart from this small semantical difference, there are also some syntactical differences. In XQuery, an expression to be executed has to be put into curly braces if it is enclosed by an element constructor. Otherwise, it is treated as a string constant. Further, a sequence of expressions $\alpha_1, \ldots, \alpha_n$ has to be written as "$(\alpha_1, \ldots, \alpha_n)$" if it is not enclosed in an element constructor, otherwise it must be phrased as "$\{\alpha_1\} \ldots \{\alpha_n\}$". To simplify notation, we always use the standard XQuery syntax in element constructors, i. e., we enclose every expression by curly braces.

Example 3.4.6 *The following standard XQuery*

```
for $x in /bib/book
return
  ($x/author, $x/title)
```

is phrased in XQuery$^-$ as

```
{ for $x in /bib/book
  return
    {$x/author}
    {$x/title} }
```

□

Figure 3.2 shows the grammar of XQuery$^-$ in an EBNF similar to that used in [W3C05b] for standard XQuery. Non-terminals on the left-hand side printed in italics denote productions of XQuery$^-$. Non-terminals on the right-hand side printed in bold refer to the original productions of the standard XQuery grammar [W3C05b].

Let $[\![Q]\!]_{XQuery^-}(D)$ (and $[\![Q]\!]_{XQuery}(D)$, respectively) denote the XML document stream produced by evaluating query Q on document D under our XQuery$^-$ semantics (and under the standard XQuery semantics [W3C05c], respectively).

Proposition 3.4.7 *Let Q be an XQuery that parses as an XQuery$^-$ query. Then, for any input document D,*

$$[\![Q]\!]_{XQuery^-}(D) \;=\; [\![Q]\!]_{XQuery}(D).$$

FixedPath ::= **VarRef** (“/” **QName**)*
AtomicClause ::= (“`fn:true()`” | “`fn:false()`”)
ValExpr ::= (**Literal**
| *FixedPath*
| *ValExpr* (“+” | “-”) *ValExpr*
| *ValExpr* (“*” | “div” | “idiv” | “mod”) *ValExpr*)
AtomicCondition ::= (*ValExpr* **GeneralComp** *ValExpr*
| *AtomicClause*
| (“`fn:empty`” | “`fn:exists`”) “(” *FixedPath* “)”
| “`fn:not(`” *AtomicCondition* “)”
| *AtomicCondition* “and” *AtomicCondition*
| *AtomicCondition* “or” *AtomicCondition*)
Expression ::= (“()”
| **Literal**
| *Expression* (*Expression*)+
| “{” “for” **VarRef** “in” *FixedPath* (“where” *AtomicCondition*)? “return” *Expression* “}”
| “{” *FixedPath* “}”
| “{” **VarRef** “}”
| “{” “if” “(” *AtomicCondition* “)” “then” *Expression* “}”)

Figure 3.2: XQuery$^-$ Grammar

3.4.2 Syntax and Semantics of FluX

To be able to define the syntax and semantics of our query language FluX, we need some more notation and a further classification of XQuery$^-$expressions.

Definition 3.4.8 (Parent Variable) *Let α be an XQuery$^-$expression. Then, we define parentVar(α) as*

- *the nearest variable $\$x$ bound by a surrounding* `for`*-expression, or*
- `$ROOT` *if no surrounding* `for`*-expression exists.*

With this, we are able to define the class of *simple conditions* and *simple expressions*.

Definition 3.4.9 (Simple Condition) *Let χ be a condition as defined in Definition 3.4.4. Then, χ is denoted as a* simple condition *if*

- *χ is an atomic condition of the form “s RelOp s'” or “$\$x/\pi$ RelOp s”, where s, s' are strings (constants), $\$x$ is some variable, and π is a fixed path, or*
- *χ is a Boolean combination (using “*`and`*”, “*`or`*”, and “*`fn:not`*”) of simple conditions.*

That is, a condition is simple if it does not contain any atomic condition performing a join, e. g., “$\$x/\pi$ *RelOp* $\$y/\pi'$”, and thus it can be checked statically or directly on-the-fly as we see the corresponding elements on the data stream.

Definition 3.4.10 (Simple Expressions) *Let* $\overline{\alpha}$ *be an XQuery*$^-$ *expression of the form* $\overline{\alpha} := \alpha\ \beta\ \gamma$. *Further, let* $\$x := parentVar(\overline{\alpha})$. $\overline{\alpha}$ *is* simple *if*

- α *and* γ *are possibly empty sequences of expressions "s" or of expressions of the form "*`{if (`χ`) then `s`}`*", where s is outputting a fixed string and* χ *is a simple condition, and*
- β *is either empty, "*$\{\$x\}$*", or "*`{if (`$\chi$`) then {`$\$x$`}}`*" for some simple condition* χ*, and*
- *if* β *is not empty, then no condition that occurs in* $\alpha\,\beta$ *contains the variable* $\$x$.

Intuitively, an expression is simple, if it can be evaluated on the current stream (defined by $parentVar(\overline{\alpha})$) without any additional buffering of the input stream.

Example 3.4.11 *The following expression* α

```
<a>{$x}</a> {if ($x/b=5) then <b>5</b>}
```

with $parentVar(\alpha) := \$x$ *is simple, since the* `if`*-condition can be checked during outputting the stream and the* `then`*-clause may be appropriately executed afterwards. In contrast,*

```
{$x} {$y}
```

is not simple, because at least the contents of one variable have to be buffered, depending on the current stream (or, parentVar). □

Using this definition of simple expressions, we now define the syntax of our query language *FluX*.

Definition 3.4.12 (FluX) *The class of FluX expressions is the smallest set of expressions that are either* simple *or of the form*

$$s\ \{\ \texttt{process-stream}\ \$y\texttt{:}\ \zeta\ \}\ s'$$

where s and s′ are possibly empty strings, $\$y$ *is a variable, and* ζ *is a list (where entries are separated by semicolons ";") of one or more* event-handlers*. Each event-handler is of one of the following two types:*

1. *(so-called "*`on-first`*"-handler)*
   `on-first past(`S`) return` α
   where $S \subseteq symb(\$y)$ *and* α *is an XQuery*$^-$ *expression*
2. *(so-called* `on`*-handler)*
   `on` a `as` $\$x$ `return` Q
   where $\$x$ *is a variable, a is an element name in* $symb(\$y)$*, and Q is a FluX expression.*

We will use "ps" as a shortcut for "process-stream", "on-first past(*)" as an abbreviation for "on-first past($symb(\$y)$)", and furthermore "on-first past()" in place of "on-first past($\emptyset$)".

Some examples of FluX expressions, as well as an informal description of the FluX semantics, were already given in Section 3.1; further examples can be found in Section 3.5.3.

In general, we evaluate an expression

$$\{ \texttt{process-stream } \$y\texttt{: } \zeta \}$$

as follows: An event-handling statement considers the children of the node currently bound by variable $\$y$ as a list (or stream) of nodes and processes this list one node at a time. On processing a node v with children $t_1, \ldots, t_n$, with the labels of t_i denoted as $label(t_i)$, we proceed as follows. For each i from 0 to $n{+}1$ (i. e., $n{+}2$ times), we scan the list of event-handlers $\zeta = \zeta_1; \ldots; \zeta_m$ once from the beginning to the end. In doing so, we test for each event-handler ζ_j whether its event condition is satisfied, in which case the event-handler ζ_j "fires" and the corresponding query expression is executed:

- A handler "on a as $\$x$ return Q" fires if $1 \leq i \leq n$ and $label(t_i) = a$.
- A handler "on-first past(S) return α" fires if $0 \leq i \leq n$ and

  $$\textit{first-past}_{\$y,S}(label(t_1) \ldots label(t_i))$$

  is true (i. e., for the first time while processing the children of $\$y$, no symbol of S can be encountered anymore) or if $i = n{+}1$ and this event-handler has not fired in any of the previous $(n{+}1)$ scans.

In summary, it is well possible that several events fire for a single node, in which case they are processed in the order in which the handlers occur in ζ. During the run on $t_1, \ldots, t_n$, each on-handler may fire zero up to several times, while each "on-first"-handler is executed exactly once.

Definition 3.4.13 (Free Variables) *For a FluX or XQuery$^-$ expression Q, let $free(Q)$ be the set of all* free variables *in Q, defined analogously to the free variables of a formula in first-order logic as follows.*

$$\begin{array}{rcl}
free(\epsilon) = free(s) & := & \emptyset \\
free(\{\$x/\pi\}) = free(\{\$x\}) & := & \{\$x\} \\
free(\alpha\ \beta) & := & free(\alpha) \cup free(\beta) \\
free(\{\texttt{if } (\chi) \texttt{ then } \alpha\}) & := & free(\alpha) \cup \{\$x \mid \$x \textit{ appears in } \chi\} \\
free(\{\texttt{for } \$x \texttt{ in } \$r/\pi \texttt{ return } \alpha\}) & := & (free(\alpha) \backslash \{\$x\}) \cup \{\$r\} \\
free(\{\texttt{for } \$x \texttt{ in } \$r/\pi \texttt{ where } \chi \texttt{ return } \alpha\}) & := & free(\{\texttt{for } \$x \texttt{ in } \$r/\pi \texttt{ return} \\
 & & \quad \{\texttt{if } \chi \texttt{ then } \alpha \ \}\}) \\
free(\{\texttt{process-stream } \$r\texttt{: } \ \zeta\}) & := & \{\$r\} \ \cup \\
 & & \bigcup\{free(\alpha) \mid \text{"}\texttt{on-first past}(X) \texttt{ return } \alpha\text{"} \in \zeta\} \ \cup \\
 & & \bigcup\{free(\beta) \backslash \{\$x\} \mid \text{"}\texttt{on } a \texttt{ as } \$x \texttt{ return } \beta\text{"} \in \zeta\}
\end{array}$$

Note that expressions of the form "`{for` $\$x$ `in` $\$y/a$ `return` α`}`" and event-handlers of the form "`on` a `as` $\$x$ `return` Q" *bind* the variable $\$x$, i. e., remove it from the free variables of the superexpressions.

Definition 3.4.14 (FluX Query) *A* FluX query *is a FluX expression in which all free variables except for the special variable* `$ROOT` *corresponding to (the root of) the document are bound. That is, for a* query Q *in FluX (respectively,* α *in XQuery*$^-$*) we require that* $free(Q) \subseteq \{\$ROOT\}$ *(respectively,* $free(\alpha) \subseteq \{\$ROOT\}$*).*

As the following example shows, every XQuery$^-$ query can be transformed into a FluX query in a straightforward way.

Example 3.4.15 *Every XQuery*$^-$ *query* α *is equivalent to the FluX query*

```
{ ps $ROOT:
  on-first past(*) return α }
```

In Section 3.5 we will show how, depending on a given DTD, this FluX query can be transformed into an equivalent FluX query that can be evaluated more efficiently. □

In the remainder of this chapter we need the size of a query to be able to estimate the complexity of our algorithms. By the *size* of an expression Q (respectively, a condition χ), denoted $|Q|$ (respectively, $|\chi|$), we refer to the size of its string representation.

3.4.3 Safe Queries

We next define the notion of *safety* for FluX queries. Informally, a query is called *safe* for a given DTD if it is guaranteed that XQuery$^-$ subexpressions do not refer to paths that might still be encountered in an input stream compliant with the given DTD.

For a precise definition we need the following notation. By the *condition paths* in α, we refer to the set of paths $\$x/\pi$ in a condition χ that occurs in α. For FluX or XQuery$^-$ expressions α and β we write $\alpha \preceq \beta$ (respectively, $\alpha \prec \beta$) to denote that α is a subexpression (respectively, proper subexpression) of β. An XQuery$^-$ subexpression α of a FluX expression Q is called *maximal* if there is no XQuery$^-$ expression β with $\alpha \prec \beta \preceq Q$. Note that a FluX query may contain several such maximal expressions.

Example 3.4.16 *The maximal XQuery*$^-$ *subexpressions of the first FluX query from Section 3.1 are "*`{$t}`*" and "*`{for $a in $book/author return {$a}}`*".* □

We characterize the set of symbols needed for the evaluation of an XQuery$^-$expression with respect to a given variable (or, production of the DTD) as *dependencies*.

Definition 3.4.17 (Dependencies) *The set of dependencies with respect to a variable* $\$y$ *of an XQuery*$^-$ *expression is defined as*

$$
\begin{array}{rcl}
dep(\$y;\ a/\pi) & := & \{a\} \\
dep(\$y;\ \{\$y/\pi\}) = dep(\$y;\ \$y/\pi) & := & dep(\$y;\ \pi) \\
dep(\$y;\ f(\alpha)) & := & dep(\$y;\ \alpha) \\
dep(\$y;\ \alpha_1\ op\ \alpha_2) & := & dep(\$y;\ \alpha_1) \cup dep(\$y;\ \alpha_2) \\
dep(\$y;\ \{\texttt{for}\ \$x\ \texttt{in}\ \$y/\pi\ \texttt{return}\ \alpha\}) & := & dep(\$y,\ \pi) \cup dep(\$y;\ \alpha) \\
dep(\$y;\ \{\texttt{for}\ \$x\ \texttt{in}\ \$y/\pi\ \texttt{where}\ \chi\ \texttt{return}\ \alpha\}) & := & dep(\$y,\ \pi) \cup dep(\$y;\ \alpha) \\
 & & \cup\ dep(\$y;\ \chi) \\
dep(\$y;\ \{\texttt{if}\ (\chi)\ \texttt{then}\ \alpha\}) & := & dep(\$y;\ \chi) \cup dep(\$y;\ \alpha) \\
dep(\$y;\ \alpha_1\ \alpha_2) & := & dep(\$y;\ \alpha_1) \cup dep(\$y;\ \alpha_2) \\
dep(\$y;\ \cdot) & := & \emptyset
\end{array}
$$

with a being a symbol from the DTD, π a non-empty path, $\alpha, \alpha_1, \alpha_2$ XQuery⁻ expressions, χ a condition, $f \in \{$`fn:not`*, *`fn:empty`*, *`fn:exists`*$\}$, $op \in ValOp \cup RelOp \cup \{$*`and`*, *`or`*$\}$, and $dep(\$y; \cdot)$ matching all other expressions not being explicitly defined.*

With this, we are able to precisely define the notion of *safe queries*, which we have already informally introduced in Section 3.1 and at the beginning of this section.

Definition 3.4.18 (Safe Queries) *A FluX query Q is called* safe *with respect to a given DTD if and only if for each subexpression "*`{ps $y: ζ }`*" of Q, the following two conditions are satisfied:*

1. *For each handler "*`on-first past(S) return α`*" in the list ζ, the following is true:*
   - *$\forall\ b \in dep(\$y, \alpha)$ we have:*
   $$(b \in S) \vee (\exists a \in S : Ord_{\$y}(b, a))$$
   - *$\forall\ \$z \in free(\alpha)$ such that "$\{\$z\}$" $\preceq \alpha$ or "$\{\$z/\pi\}$" $\preceq \alpha$ (for some π) we have:*
   $$(\$z = \$y) \wedge (\forall\ b \in symb(\$y) : (b \in S) \vee (\exists a \in S : Ord_{\$y}(b, a))).$$
2. *For each handler "*`on a as $x return` $\tilde{Q}$*" in the list of handlers ζ, and for each maximal XQuery⁻ subexpression α of $\tilde{Q}$, the following is true:*
   - *$\forall\ b \in dep(\$y, \alpha)$ we have:*
   $$Ord_{\$y}(b, a)$$
   - *if $\alpha = \tilde{Q}$ (note that according to Definition 3.4.12 α must then be* simple*), then for all $\$u$ such that "$\{\$u\}$" $\preceq \alpha$ we have:*
   $$\$u = \$x.$$

This notion of *safety* is sufficient to ensure that main memory buffers are fully populated when they are accessed by a query, i. e., that a FluX query can be evaluated in a straightforward way on input streams compliant with the given DTD.

Examples of safe and un-safe FluX queries with respect to a given DTD have been shown in Section 3.1. Further examples of safe FluX queries will be presented in Section 3.5. To be precise, all FluX queries occurring in the remainder of this chapter are safe.

$$\frac{\texttt{\{ for \$}x\texttt{ in \$}y/\pi\texttt{ where }\chi\texttt{ return }\beta\texttt{ \}}}{\texttt{\{ for \$}x\texttt{ in \$}y/\pi\texttt{ return \{ if (}\chi\texttt{) then }\beta\texttt{ \} \}}} \quad \textit{[Elim-Where]}$$

$$\frac{\texttt{\{ \$}y/\pi\texttt{ \}}}{\texttt{\{ for \$}x\texttt{ in \$}y/\pi\texttt{ return \{\$}x\texttt{\} \}}} \quad \textit{[Only-For]}$$

$$\frac{\texttt{\{ for \$}x\texttt{ in \$}y/a/\pi\texttt{ return }\beta\texttt{ \}}}{\begin{array}{l}\texttt{\{ for \$}x_0\texttt{ in \$}y/a\texttt{ return}\\ \quad\texttt{\{ for \$}x\texttt{ in \$}x_0/\pi\texttt{ return }\beta\texttt{ \} \}}\end{array}} \quad \textit{[Single-Step-For]}$$

$$\frac{\texttt{\{ if (}\chi\texttt{) then \{ for \$}x\texttt{ in \$}y/\pi\texttt{ return }\alpha\texttt{ \} \}}}{\texttt{\{ for \$}x\texttt{ in \$}y/\pi\texttt{ return \{ if (}\chi\texttt{) then }\alpha\texttt{ \} \}}} \quad \textit{[Pushdown-If-1]}$$

$$\frac{\texttt{\{ if (}\chi\texttt{) then }\alpha\ \beta\texttt{ \}}}{\texttt{\{ if (}\chi\texttt{) then }\alpha\texttt{ \} \{ if (}\chi\texttt{) then }\beta\texttt{ \}}} \quad \textit{[Pushdown-If-2]}$$

$$\frac{\texttt{\{ if (}\chi\texttt{) then \{ if (}\psi\texttt{) then }\alpha\texttt{ \} \}}}{\texttt{\{ if (}\chi\texttt{ and }\psi\texttt{) then }\alpha\texttt{ \}}} \quad \textit{[Merge-If]}$$

Figure 3.3: Normal Form Rewrite Rules

3.5 Translating XQuery$^-$ into FluX

In this section, we address the problem of rewriting a query of our XQuery$^-$ fragment into an equivalent FluX query that employs as little buffering as possible. This rewriting proceeds in two steps: First, we transform the given XQuery$^-$ query into an equivalent query in XQuery$^-$ *normal form*, which will be presented in Section 3.5.1. In Section 3.5.2, we present our algorithm for rewriting this normalized query, depending on a given DTD, into an equivalent safe FluX query. The FluX extensions manage the event-based, streaming execution of the query. All subqueries exclusively working on buffered data are XQuery$^-$ expressions. Some examples of this transformation are given in Section 3.5.3.

3.5.1 A Normal Form for XQuery$^-$

An XQuery$^-$expression is transformed into *normal form* by rewriting (subexpressions of) it using the rules in Figure 3.3 until no further changes are possible. In this work, rewrite rules are displayed in the form

$$\frac{\text{expression } \alpha}{\text{rewritten expression } \alpha'} \quad \text{(condition } c\text{)}$$

where query expression α is rewritten to α', iff condition c holds. All rewrite rules are equivalences, but our rewrite strategy is always downward. I. e., it is never necessary to employ a rule to go from an expression α' to an expression α. Note that none of the normal

form rewrite rules shown in Figure 3.3 has a condition, i. e., they are always applied on a matching expression. Conditional rewriting rules will be used in Section 3.6.

For an XQuery$^-$ expression in normal form the following three properties hold:

1. All paths except those inside conditionals are simple (single-)step paths of the form $\$x/a$.

2. An expression in normal form does not contain any *conditional* `for`-loops, as the normalization process pushes conditionals inside the innermost `for`-loops.

3. For each subexpression of the form "`{if` χ `then` α`}`", α is either a fixed string or of the form "`{`$\$x$`}`" for some variable $\$x$.

In particular, the normalization process leaves paths inside conditionals unchanged, while all other fixed paths $\$y/a/\pi$, where π is a path of nonzero length, are decomposed into nested `for`-loops.

It can be shown that for an XQuery Q the rule applications of Figure 3.3 terminate with a unique result, the so-called *normalization* of Q. Obviously, the normalization of an XQuery Q is equivalent to Q. Further, the rewriting can be implemented in such a way that it terminates after $O(|Q|)$ rule applications [KSSS04c].

The following example shows the normalization process on behalf of a sample query taken from the XQuery Use Cases [W3C05a].

Example 3.5.1 ([W3C05a], XMP-Q1) *Consider the following XQuery Q_1 for books published by Addison-Wesley after 1991, including their year and title.*

```
<bib>
{ for $b in $ROOT/bib/book
  where $b/publisher = "Addison-Wesley" and $b/year > 1991
  return
    <book>
      {$b/year}
      {$b/title}
    </book> }
</bib>
```

In the following, we abbreviate the condition

```
$b/publisher = "Addison-Wesley" and $b/year > 1991
```

by χ.

Then, Q_1 has the following normalization, denoted as Q'_1:

```
<bib>
{ for $bib in $ROOT/bib return
  { for $b in $bib/book return
    { if (χ) then <book> }
    { for $year in $b/year return
      { if (χ) then {$year} } }
    { for $title in $b/title return
      { if (χ) then {$title} } }
    { if (χ) then </book> } } }
</bib>
```

□

3.5.2 Rewriting Normalized XQuery$^-$ into FluX

To formulate our main rewrite algorithm for transforming normalized XQuery$^-$ queries into equivalent, safe FluX queries, (Function "rewrite"), we need some further notation.

Definition 3.5.2 (HSymb) *Let Σ be the set of tag names occurring in the given DTD and $\perp$ denote an empty list of event handlers. For a list ζ of event handlers, we inductively define the set $hsymb(\zeta)$ of handler symbols for which an* `on`*-handler or an "*`on-first`*"-handler exists in ζ:*

$$\begin{aligned} hsymb(\perp) &:= \emptyset \\ hsymb(\zeta;\ \texttt{on}\ a\ \texttt{as}\ \$x\ \texttt{return}\ \alpha) &:= hsymb(\zeta) \cup \{a\} \\ hsymb(\zeta;\ \texttt{on-first past}(S)\ \texttt{return}\ \alpha) &:= hsymb(\zeta) \cup S \end{aligned}$$

Our algorithm for recursively rewriting normalized XQuery$^-$ expressions into FluX is shown in Function "rewrite". Note that this algorithm uses order constraints and hence depends on the underlying DTD. If no DTD is given, we assume a very generic DTD which does not impose any order constraints. Given a query Q, we obtain the corresponding FluX query as "rewrite(`$ROOT`, $\perp$, Q)". Note that we need to distinguish the situation that no set of symbols is given (as the second parameter of rewrite, H) from the situation that rewrite is called with an empty set. We do this by using "$\perp$" for the situation that no set is given (in the sense of a `null`-pointer in programming languages), and depicting the empty set by "$\emptyset$". Some example runs of this algorithm are given in Section 3.5.3 below. The goals in the design of the algorithm were to produce a FluX query which

1. is safe with respect to the given DTD,
2. is equivalent to the input XQuery (on all XML documents compliant with the given DTD), and
3. minimizes the amount of buffering needed for evaluating the query.

Function rewrite(*Variable* $\$x$, *Set*$\langle\Sigma\rangle$ H, *XQuery*$^-$ β) **returns** *FluXQuery*

```
1  begin
2      if {$x} ⪯ β then
3          if β is simple then
4              return β ;
5          else
6              return {ps $x:  on-first past(∗) return β} ;
7          endif
8      else
9          if β is simple and H = ⊥ then
10             return β ;
11         else if β = β1 β2 then
12             if H = ⊥ then H := ∅ ;
13             β'1 := rewrite ($x, H, β1) ;
14             match ζ1 such that β'1 = {ps $x:  ζ1 } ;
15             β'2 := rewrite ($x, H ∪ hsymb(ζ1), β2) ;
16             match ζ2 such that β'2 = {ps $x:  ζ2 } ;
17             return {ps $x:  ζ1; ζ2 }
18         else if β is of the form {for $y in $z/a return α} then
19             X := {b ∈ dep($x, α) ∪ H | ¬Ord_$x(b, a)} ;
20             if $z ≠ $x then
21                 return {ps $x:  on-first past(X) return β};
22             else if X ≠ ∅ then
23                 return {ps $x:  on-first past(X ∪ {a}) return β} ;
24             else
25                 α' := rewrite ($y, ⊥, α) ;
26                 return {ps $x:  on a as $y return α'} ;
27             endif
28         else
29             return {ps $x:  on-first past(dep($x, β) ∪ H) return β} ;
30         endif
31     endif
32 end
```

To meet goals (1) and (2), e. g., the particular order of the **if**-statements in the algorithm (lines 2, 9, 11, 18) is crucial. Also, a set H of handler symbols must be passed on in recursive calls of the algorithm, because otherwise the resulting FluX query would not be safe. One important construct for meeting goal (3) is the case distinction in lines 20–27, where an `on`-handler is created provided that this is *safe*, and an "`on-first`"-handler is created otherwise.

Given a DTD D and a normalized XQuery$^-$ query Q, "rewrite($ROOT, $\perp$, Q)" runs in time $O(|D|^3 + |Q|^2)$ and produces a safe FluX query that is equivalent to Q on all XML documents compliant to the given DTD. Although our algorithm performs only a single traversal of the query tree, the runtime of $O(|Q|^2)$ is due to the need to compute *dependencies* in subtrees of the expression being currently rewritten. In addition, $O(|D|^3)$ is needed to compute order constraints imposed by the given DTD as shown in Section 3.3.

Note that the resulting FluX query is also in normal form in the sense that every XQuery$^-$ subexpression is in normal form.

3.5.3 Examples

We now discuss the effect of our rewrite algorithm on sample queries from the XQuery Use Cases [W3C05a][5]. In Section 3.1 we have already shown the query XMP-Q3 and the resulting FluX queries obtained by our rewrite algorithm using DTDs with and without order constraints, respectively, between titles and authors. In the optimal case, i. e., with the order constraint that all titles arrive before authors, this query can be evaluated without any buffering.

The following query will always require some data to be buffered. However, the amount of data that must be buffered depends on the schema.

Example 3.5.3 ([W3C05a], XMP-Q2) *Let us consider the XQuery XMP-Q2, in the following denoted as Q_2, from the XQuery Use Cases [W3C05a], which creates a flat list of all the title–author pairs, with each pair enclosed in a* `result` *element.*

```
<results>
{ for $b in $ROOT/bib/book return
  { for $t in $b/title return
    { for $a in $b/author return
        <result>
        {$t}
        {$a}
        </result> } } }
</results>
```

Normalizing this query yields the following query Q_2' (which is very similar to the original XQuery Q_2):

[5] We rewrite the queries to work without attributes as shown in Appendix C.1.

```
1   <results>
2   { for $bib in $ROOT/bib
3     { for $b in $bib/book return
4       { for $t in $b/title return
5         { for $a in $b/author return
6             <result>
7             {$t}
8             {$a}
9             </result> } } } }
10  </results>
```

When given a DTD that does not impose any order constraints on `title` *and* `author` *(only the* `book` *element is of relevance in this query), e. g., the first DTD from Section 3.1, then "*rewrite*($\$ROOT$, $\perp$, Q'_2)" proceeds as follows: First, Q'_2 is decomposed into two subexpressions β_1, consisting of line 1, and β_2, consisting of lines 2–10. Then, the rewrite algorithm is recursively called for β_1 and for β_2. As $H = \emptyset$, the call for β_1 produces the result:*

```
{ ps $ROOT:
  on-first past() return <results>; }
```

*The call for β_2 decomposes β_2 into two subexpressions β_{21}, consisting of lines 2–9, and β_{22}, consisting of line 10 of Q'_2. The recursive call "*rewrite*($\$ROOT$, $\emptyset$, β_{21})" then executes lines 24–27 of the rewrite algorithm, because β_{21} is a* `for`*-loop with parent variable $\$ROOT$ and associated set $X = X_{\beta_{21}} = \emptyset$. That is, the result*

```
{ ps $ROOT:
  on bib as $bib return α1; }
```
(with $\overline{\alpha_1}$)

*is produced, where $\overline{\alpha_1}$ is the result produced by the recursive function call "*rewrite*($\$bib$, $\perp$, α_1)", for the subquery α_1 of Q'_2 in lines 3–9. This recursive call for α_1 again executes lines 24–27 of the algorithm, producing the expression $\overline{\alpha_1}$ =*

```
{ ps $bib:
  on book as $b return α2; }
```
(with $\overline{\alpha_2}$)

*where $\overline{\alpha_2}$ is the result of "*rewrite*($\$b$, $\perp$, α_2)" for the subquery α_2 of Q'_2 in lines 4–9. As α_2 is a* `for`*-loop with parent variable $\$b$ and associated set $X = X_{\alpha_2} = \{$*`author`*$\}$, in this call line 23 of the algorithm is executed, producing the expression $\overline{\alpha_2}$ =*

```
{ ps $b:
  on-first past(author, title) return α2; }
```

*All in all, "*rewrite(`$ROOT`, $\perp$, Q_2')*" returns the following FluX query F_2:*

```
1  { ps $ROOT:
2    on-first past() return <results>;
3    on bib as $bib return
4      { ps $bib:
5        on book as $b return
6          { ps $b:
7            on-first past(author, title) return
8              { for $t in $b/title return
9                { for $a in $b/author return
10                   <result>
11                   {$t}
12                   {$a}
13                   </result> } }; }; };
14   on-first past(bib) return </results>; }
```

We will refer to the "`{ps $b ··· }`*"-expression in lines 6–13 of F_2 as $\overline{\alpha_2}$. When evaluating the query F_2 on an XML document, the XQuery inside $\overline{\alpha_2}$ will be evaluated once* all `author` *and* all `title` *nodes have been encountered and buffered.*

Let us now consider the case where we are given a DTD with the production

```
<!ELEMENT book (author*, title*)>
```

where the order constraint Ord_{book}(`author`, `title`*) is met. While running "*rewrite(*$ROOT*, $\perp$, Q_2')*" we now encounter the situation where $X = X_{\alpha_2} = \emptyset$ (rather than \{*`author`*\}, as with the previous DTD). Therefore, when processing the recursive call "*rewrite(*$b*, $\perp$, α_2)*", now lines 24–27 of the algorithm are executed, eventually producing the following result $\widehat{\alpha_2}$ =*

```
{ ps $b:
  on title as $t return
    { ps $t:
      on-first past(*) return
        { for $a in $b/author return
            <result>
            {$t}
            {$a}
            </result> }; }; }
```

*Now, "*rewrite(*$ROOT*, $\perp$, Q_2')*" yields query F_2' differing from F_2 in the lines 6–13, which must be replaced by the above expression $\widehat{\alpha_2}$.*

When evaluating F_2' on an XML document compliant with the second DTD, all `author` *nodes arrive before* `title` *nodes and are buffered. Encountering a* `title` *node in the input stream invokes the following actions: The value of that particular node is buffered, i. e., "*`on-first past(*)`*" delays the execution until the complete* `title` *node has been seen.*

Then, we iterate over the buffer containing all collected `author` *nodes, each time writing the buffered* `title` *and the current* `author` *to the output. In contrast to the worst-case scenario above, we only buffer one title at a time in addition to the list of all authors. If there is more than one title, this strategy is clearly preferable.* □

We next demonstrate that conditional `for`-loops are optimized correspondingly in the following example.

Example 3.5.4 ([W3C05a], XMP-Q1) *Let us consider the query Q_1 and its normalization Q'_1 from Example 3.5.1. Given a DTD that does not impose any order constraints, e. g., the DTD:*

```
<!ELEMENT bib   (book)*>
<!ELEMENT book  (title|publisher|year)*>
```

*Then, the function call "*rewrite*($\$ROOT$, $\perp$, Q'_1)" rewrites Q'_1 into the following FluX query F_1 (with the condition abbreviated by χ):*

```
1  { ps $ROOT:
2    on-first past() return <bib>;
3    on bib as $bib return
4      { ps $bib:
5        on book as $b return
6          { ps $b:
7            on-first past(publisher, year) return
8              { if (χ) then <book> }
9            on-first past(publisher, year) return
10             { for $year in $b/year return
11               { if (χ) then {$year} } };
12           on-first past(publisher, year, title) return
13             { for $title in $b/title return
14               { if (χ) then {$title} } };
15           on-first past(publisher, year, title) return
16             { if (χ) then </book> } } };
17   on-first past(bib) return </bib> }
```

The "`on-first`*"-handler in lines 12–14 delays query execution until all* `title` *nodes have been buffered and all* `publisher` *and* `year` *nodes have been seen. The condition χ is evaluated on the fly and does not require further buffers except one bit storing its current state.*

Consider a different DTD which ensures that both order constraints Ord_{book}(`year`, `title`) *and Ord_{book}*(`publisher`, `title`) *hold, e. g.*

```
<!ELEMENT bib   (book)*>
<!ELEMENT book  (publisher*, year*, title*)>
```

Then, the `title` *nodes can be processed in a streaming fashion. The following query F'_1, which differs from the above query in lines 12–14, is produced by "*rewrite*($\$ROOT$, $\perp$,Q'_1)" using this new DTD:*

```
1  { ps $ROOT:
2    on-first past() return <bib>;
3    on bib as $bib return
4      { ps $bib:
5        on book as $b return
6          { ps $b:
7            on-first past(publisher, year) return
8              { if (χ) then <book> }
9            on-first past(publisher, year) return
10             { for $year in $b/year return
11               { if (χ) then {$year} } };
12           on title as $title return
13             { if (χ) then {$title} };
14           on-first past(publisher, year, title) return
15             { if (χ) then </book> } } };
16   on-first past(bib) return </bib> }
```

Consequently, titles will not be buffered at all during evaluation of this query. □

Note that more than two *subsequent* "on-first"-handlers can be merged into a single handler by concatenating their subqueries if they work on equal past-sets. For instance, in query F_1' of the above example the two handlers in lines 7 and 9 can be merged into a single handler by appending the subquery of lines 10–11 to the subquery in line 8. Non-subsequent handlers *must not* be merged, because in this case, the execution order of statements given by the XQuery might be violated depending on when the events are fired.

Our rewrite algorithm is capable of optimizing joins over two or more join predicates, as is demonstrated in the following example which is not part of the XQuery Use Cases.

Example 3.5.5 *We remain in the bibliography domain and consider documents compliant with the DTD:*

```
<!ELEMENT bib     (book|article)*>
<!ELEMENT book    (title, (author+|editor+), publisher)>
<!ELEMENT article (title, author+, journal)>
```

The following XQuery Q_3 retrieves those authors of articles which are co-authored by people who have also edited books:

```
<results>
  { for $bib in $ROOT/bib return
      for $article in $bib/article return
        for $book in $bib/book
        where $article/author = $book/editor
        return
          <result>
            {$article/author}
          </result> }
</results>
```

For the remainder of this example, we abbreviate the join-condition comparing the authors of articles with the editors of books by χ. Normalization yields the following XQuery⁻ query Q'_3:

```
1  <results>
2  { for $bib in $ROOT/bib return
3    { for $article in $bib/article return
4      { for $book in $bib/book return
5        { if (χ) then <result> }
6        { for $author in $article/author return
7          { if (χ) then {$author} } }
8        { if (χ) then </result> } } } }
9  </results>
```

When executing "rewrite($\$ROOT$, $\perp$, Q'_3)" with the DTD given above, the recursive call "rewrite($\$bib$, $\perp$, β)" is eventually invoked for the subexpression β of Q'_3 in lines 3–8. As β is a `for`*-loop with parent variable $\$bib$ and associated set $X = X_\beta = \{$*`book`*$\} \neq \emptyset$, line 23 of the algorithm is executed, returning an expression of the form*

```
{ ps $bib: on-first past(book, article) ··· }.
```

That is, as no order constraint between `article` *and* `book` *holds, an "*`on-first`*"-handler ensures that all articles and books will be buffered. Altogether, "rewrite($\$ROOT$, $\perp$, Q'_3)" produces the following FluX query F_3:*

```
1  { ps $ROOT:
2    on-first past() return <results>;
3    on bib as $bib return
4      { ps $bib:
5        on-first past(book, article) return
6          { for $article in $bib/article return
7            { for $book in $bib/book return
8              { if (χ) then <result> }
9              { for $author in $article/author return
10               { if (χ) then {$author} } }
11             { if (χ) then </result> } } } }
12   on-first past(bib) return </results> }
```

When given a different DTD which imposes an order on books and articles, e. g., by the following production

```
<!ELEMENT bib     (book*, article*)>
```

we can evaluate Q'_3 by buffering only `book` *nodes but processing* `article` *nodes in a streaming fashion. Indeed, when executing "rewrite($\$ROOT$, $\perp$, Q'_3)" with this new DTD, we eventually encounter the situation where the set $X = X_\beta = \emptyset$, and therefore, lines 24–27 (rather than line 23, as with the previous DTD) are executed. Altogether, the following FluX query F'_3 is produced now, which differs from the above query F_3 in the subexpression in line 5:*

```
1  { ps $ROOT:
2    on-first past() return <results>;
3    on bib as $bib return
4      { ps $bib:
5        on article as $article return
6          { for $book in $bib/book return
7            { if (χ) then <result> }
8            { for $author in $article/author return
9              { if (χ) then {$author} } }
10           { if (χ) then </result> } } }
11   on-first past(bib) return </results> }
```

As all `book` *nodes will have arrived before an* `article` *node can be encountered, data from books is available in buffers once the first* `article` *node is being read. When processing the children of an* `article` *node, we first buffer all* `author` *nodes before the query can be evaluated for the current article. During the evaluation of F_3', we therefore only buffer the authors of a single article in addition to the data already stored on books, whereas the evaluation of F_3 requires the authors of* all *articles to be buffered.* □

3.6 Algebraic Optimization of XQuery$^-$

In this section, we present an effective and efficient rewrite system for algebraically optimizing XQueries exploiting schema knowledge from the DTD. All rewrite rules are, again, equivalences that are relevant beyond the optimization of XQuery in main memory engines, even though emphasis is put on reducing memory consumption rather than running time. In practice, for large data volumes, these two goals are of course very highly correlated. Algebraic optimization of XQueries is done in a preprocessing phase before rewriting them into FluX. These optimizations are optional in the sense that they provide no prerequisite for the application of our main rewrite algorithm introduced in Section 3.5.2. However, the algebraic optimization steps introduced here improve the efficiency of the resulting FluX queries, as we will show in Section 3.6.2.

3.6.1 Rewrite Rules for Algebraic Optimization

In this section, we assume that all XQuery$^-$ *conditions* are in *negation normal form*, i. e., negation only occurs directly in front of *atomic* conditions. Of course, conditions can be efficiently transformed into negation normal form by using De Morgan's laws. To formulate the rewriting rules for algebraic optimization (Figure 3.5), we first need to define some further notation.

For each condition χ in negation normal form we define a language $L_{req}(\chi)$ in such a way that the following is true: If the sequence of children of the node bound by variable $\$x$ does *not* belong to $L_{req}(\chi)$, then condition χ cannot be satisfied at this node.

$$\frac{\alpha}{\epsilon} \quad \big(\exists\, \$x \text{ such that } L(\$x) \cap L_{req\uparrow Q}(\$x,\ \alpha) = \emptyset\big) \qquad \textit{[DTD-Impl]}$$

$$\frac{\texttt{\{ if (}\chi\texttt{) then } \beta \texttt{ \}}}{\texttt{\{ if (}simpl_{DTD}(\chi)\texttt{) then } \beta \texttt{ \}}} \qquad \textit{[Simplify-If]}$$

$$\frac{\texttt{\{ if (fn:true()) then } \beta \texttt{ \}}}{\beta} \qquad \textit{[Sat-If]}$$

$$\frac{\texttt{\{ if (fn:false()) then } \beta \texttt{ \}}}{\epsilon} \qquad \textit{[Unsat-If]}$$

$$\frac{\texttt{\{ for } \$x \texttt{ in } \$y/a \texttt{ return } \epsilon \texttt{ \}}}{\epsilon} \qquad \textit{[Silent-For]}$$

$$\frac{\texttt{\{ for } \$x \texttt{ in } \$z/a \texttt{ return } \alpha \texttt{ \} \{ for } \$y \texttt{ in } \$z/a \texttt{ return } \beta \texttt{ \}}}{\texttt{\{ for } \$x \texttt{ in } \$z/a \texttt{ return } \alpha\ \ \beta[\$y \to \$x] \texttt{ \}}} \quad \Big(a \in ||^{\leq 1}_{\$z}\Big) \qquad \textit{[Merge1]}$$

$$\frac{\begin{array}{l}\texttt{\{ for } \$x \texttt{ in } \$z/a \texttt{ return } \alpha \texttt{ \}} \\ \beta_1 \ldots \beta_n \\ \texttt{\{ for } \$y \texttt{ in } \$z/a \texttt{ return } \gamma \texttt{ \}}\end{array}}{\texttt{\{ for } \$x \texttt{ in } \$z/a \texttt{ return } \alpha\ \ \beta_1 \ldots \beta_n\ \ \gamma[\$y \to \$x] \texttt{ \}}} \quad \left(\begin{array}{l} a \in ||^{=1}_{\$z}, \\ \beta_i \text{ is either of the form} \\ s_i \text{ or } \texttt{\{ if (}\chi_i\texttt{) then } s_i \texttt{ \}} \end{array}\right) \qquad \textit{[Merge2]}$$

$$\frac{\texttt{\{ for } \$x \texttt{ in } \$z/a \texttt{ return } \alpha \texttt{ \}}}{\texttt{\{ for } \$x \texttt{ in } \$z/a \texttt{ return } \alpha' \texttt{ \}}} \quad \left(\begin{array}{l} a \in ||^{\leq 1}_{\$z},\ \alpha' \text{ is obtained from } \alpha \text{ by replacing} \\ \text{each condition path } \$z/a/\pi\ (\pi \neq \epsilon), \\ \text{by } \$x/\pi \text{ and each subexpression of the form} \\ \texttt{\{ for } \$y \texttt{ in } \$z/a \texttt{ return } \beta \texttt{ \}} \text{ by } \beta[\$y \to \$x] \end{array}\right) \qquad \textit{[Simplify]}$$

Figure 3.5: Algebraic Optimization Rewrite Rules

Definition 3.6.1 (Condition Requirements) *We define the* requirement of a condition χ*, denoted as* $L_{req}(\chi)$*, as follows:*

$$\begin{array}{rcl}
L_{req}(\$x,\ \texttt{fn:exists(}\$x/a/\pi\texttt{)}) & := & L(\Sigma^*.a.\Sigma^*) \\
L_{req}(\$x,\ \texttt{fn:empty(}\$x/a/\pi\texttt{)}) & := & L((\Sigma \setminus \{a\})^*) \\
L_{req}(\$x,\ \$x/a/\pi\ op\ c) & := & L(\Sigma^*.a.\Sigma^*) \\
L_{req}(\$x,\ \$x/a/\pi\ op\ \$y/b/\pi') & := & L(\Sigma^*.a.\Sigma^*) \\
L_{req}(\$x,\ \$x/a/\pi\ op\ \$x/b/\pi') & := & L(\Sigma^*.a.\Sigma^*) \cap L(\Sigma^*.b.\Sigma^*)
\end{array}$$

Here, $op \in \mathit{ValOp} \cup \mathit{RelOp}$*, with ValOp and RelOp as introduced in Definition 3.4.2 and 3.4.3. For all other (unnegated or negated) atomic conditions* χ_0*, we let*

$$L_{req}(\$x,\ \chi_0) := L(\Sigma^*).$$

Finally, let

$$\begin{array}{rcl}
L_{req}(\$x,\ \chi_1 \texttt{ and } \chi_2) & := & L_{req}(\$x,\ \chi_1) \cap L_{req}(\$x, \chi_2) \\
L_{req}(\$x,\ \chi_1 \texttt{ or } \chi_2) & := & L_{req}(\$x,\ \chi_1) \cup L_{req}(\$x,\ \chi_2)\,.
\end{array}$$

Therewith, we overload the definition of L_{req} for expressions:

$$\begin{aligned} L_{req}(\$x,\ \texttt{\{if (}\chi\texttt{) then }\alpha\texttt{\}}) &:= L_{req}(\$x,\ \chi) \\ L_{req}(\$x,\ \texttt{\{for }\$y\texttt{ in }\$x/a\texttt{ return }\alpha\texttt{\}}) &:= L(\Sigma^*.a.\Sigma^*) \end{aligned}$$

and we let

$$L_{req}(\$x,\ \alpha) := L(\Sigma^*)$$

for all other kinds of expressions α in normal form.

Given an XQuery$^-$ expression Q in normal form and a subexpression α of Q, we define

$$L_{req\uparrow Q}(\$x,\ \alpha) := \bigcap_{\beta\ :\ \alpha \preceq \beta \preceq Q} L_{req}(\$x,\ \beta)\,.$$

Intuitively, $L_{req\uparrow Q}(\$x,\ \alpha)$ denotes the requirements that expression α imposes on the sequence of children of the node bound by variable $\$x$ in the following sense: If this sequence does *not* belong to $L_{req\uparrow Q}(\$x,\ \alpha)$, then the subquery α will never be executed while $\$x$ is bound to this node. This gives rise to the rewrite rule *[DTD-Impl]* in Figure 3.5. The condition for the application of this rule can be checked using finite automata (cf., [HU79]).

We now provide some notation needed for the elimination of *conditions* that can never be satisfied or are always satisfied, respectively, in XML documents compliant with a given DTD. Let v, v' be value expressions as defined in Definition 3.4.2 and v or v' contain a path $\pi = a_1/\cdots/a_n$ such that $a_1 \notin\ symb(\$x)$ or there is an $i < n$ such that $a_{i+1} \notin symb(a_i)$. In other words, v or v' contain a path that is not valid according to the given DTD. We say that an atomic condition χ_0 is *unsatisfiable with respect to the DTD*, if χ_0 is of one of the forms

- `fn:exists(`v`)` or
- v *RelOp* v'.

Informally, a condition is unsatisfiable, if it contains a path which is not valid according to the given DTD. Analogously, we say that an atomic condition χ_1 of the form

- `fn:empty(`v`)`

is *always satisfied with respect to the DTD*.

Let χ be a condition in negation normal form that occurs in an XQuery$^-$ expression Q. We define

$$\textit{replace-unsat}_{\textit{DTD}}(\chi)$$

to be the condition (in negation normal form) that is obtained from χ by replacing every unnegated (or negated, respectively) occurrence of an atomic condition χ_0 that is unsatisfiable with respect to the DTD by the Boolean constant `fn:false()` (or `fn:true()`, respectively). Analogously, all occurrences of an always satisfied atomic condition χ_1 are replaced by the Boolean constant `fn:true()`.

For every condition χ (possibly, with occurrences of the Boolean constants `fn:true()` and `fn:false()`) we define its *simplification* as follows.

Definition 3.6.2 (Condition Simplification) *We inductively define the* simplification $simpl_{basic}(\chi)$ *of* χ*, where the Boolean constants are propagated as far as possible, as follows:*

- *If* χ *is a Boolean constant or a negated or unnegated atomic condition, then*

$$simpl_{basic}(\chi) := \chi.$$

- *If* χ *is of the form "*χ_1 `and` χ_2*", then*

$$simpl_{basic}(\chi) := \begin{cases} \texttt{fn:true()} & simpl_{basic}(\chi_1) = \texttt{fn:true()} \wedge \\ & simpl_{basic}(\chi_2) = \texttt{fn:true()} \\ \texttt{fn:false()} & simpl_{basic}(\chi_1) = \texttt{fn:false()} \vee \\ & simpl_{basic}(\chi_2) = \texttt{fn:false()} \\ simpl_{basic}(\chi_1) & simpl_{basic}(\chi_2) = \texttt{fn:true()} \\ simpl_{basic}(\chi_2) & simpl_{basic}(\chi_1) = \texttt{fn:true()} \\ simpl_{basic}(\chi_1)\ \texttt{and}\ simpl_{basic}(\chi_2) & \textit{otherwise.} \end{cases}$$

- *If* χ *is of the form "*χ_1 `or` χ_2*", then*

$$simpl_{basic}(\chi) := \begin{cases} \texttt{fn:true()} & simpl_{basic}(\chi_1) = \texttt{fn:true()} \vee \\ & simpl_{basic}(\chi_2) = \texttt{fn:true()} \\ \texttt{fn:false()} & simpl_{basic}(\chi_1) = \texttt{fn:false()} \wedge \\ & simpl_{basic}(\chi_2) = \texttt{fn:false()} \\ simpl_{basic}(\chi_1) & simpl_{basic}(\chi_2) = \texttt{fn:false()} \\ simpl_{basic}(\chi_2) & simpl_{basic}(\chi_1) = \texttt{fn:false()} \\ simpl_{basic}(\chi_1)\ \texttt{or}\ simpl_{basic}(\chi_2) & \textit{otherwise.} \end{cases}$$

Finally, we define for a condition χ *in negation normal form, occurring in an XQuery* Q

$$simpl_{DTD}(\chi) \quad := \quad simpl_{basic}\big(replace\text{-}unsat_{DTD}(\chi)\big).$$

Informally, $simpl_{DTD}(\chi)$ is the simplified condition derived by using knowledge of the DTD to eliminate unsatisfiable (or always satisfied, respectively) atomic conditions.

Obviously, $simpl_{DTD}(\chi)$ can be computed in time linear in the size of χ. The following is true for every XML document D compliant with the DTD: For every variable binding that occurs while evaluating Q on D, the condition χ is satisfied if, and only if, the condition $simpl_{DTD}(\chi)$ is. This gives rise to the rewrite rules *[Simplify-If]*, *[Sat-If]*, and *[Unsat-If]* in Figure 3.5.

Further, let $\alpha[\$x \rightarrow \$y]$ denote the expression obtained by substituting all occurrences of $\$x$ in α by $\$y$.

Now, given a DTD, a normalized XQuery^{-} expression Q is *algebraically optimized* by rewriting (subexpressions of) it using the rules in Figure 3.5 until no further changes are possible. It is not difficult to see that the resulting XQuery^{-} expression is, again, in normal form. Furthermore, the rewrite rules in Figure 3.5 are, in fact, equivalences.

In the following, we briefly characterize the complexity of our algebraic optimizations [KSSS04c]. Let ℓ_{DTD} denote the maximum length of the regular expressions occurring in the DTD. For every normalized XQuery$^-$ expression Q let v_Q be the number of variables occurring in Q, d_Q the depth of Q (i. e., the maximum number of nested subexpressions), and c_Q the maximum size (in terms of "`and`" and "`or`" connectives) of conditions in Q. For every XQuery$^-$ query Q in normal form and every subquery $\alpha \preceq Q$ it takes $O\big(\ell_{DTD} \cdot v_Q \cdot 4^{c_Q \cdot d_Q}\big)$ computation steps to check whether rule *[DTD-Impl]* can be applied to α and to actually apply the rule. This complexity can be deduced by estimating the size of the automaton needed to check the condition $L(\$x) \cap L_{req\uparrow Q}(\$x, \alpha) = \emptyset$ of the rule *[DTD-Impl]* for each variable in Q. Further, it is easy to see, that for each rule except *[DTD-Impl]*, it takes $O(|Q|)$ computation steps to check whether the rule can be applied to α and to actually apply the rule. Similar to rewriting an XQuery into its normal form, the rewriting rules of Figure 3.5 can be implemented in such a way that the following is true: Given a DTD and a normalized XQuery$^-$ query Q, the rewriting terminates after $O(|Q|)$ rule application with a (unique) normalized XQuery$^-$ query Q' which is equivalent to Q on all XML documents compliant with the given DTD.

3.6.2 Examples

In this section, we further explain the rewrite rules from Figure 3.5 with some examples.

When it can be derived from the DTD that certain elements will not appear (in combination), we may prune such expressions accordingly. This is the basic idea behind the rules *[DTD-Impl]* and *[Simplify-If]*, *[Sat-If]*, and *[Unsat-If]*.

Example 3.6.3 *Consider the third DTD from Section 3.1, i. e., the DTD:*

```
<!ELEMENT bib  (book)*>
<!ELEMENT book (title, (author+|editor+), publisher, price)>
```

Let Q be the following XQuery:

```
<result>
  { for $bib in $ROOT/bib return
      ( if (fn:not(fn:exists($bib/book/subtitle))) then
          <remark> books don't have subtitles </remark>,
        for $book in $bib/book return
          if ($book/author = "Hull" and $book/editor = "Vianu") then
            $book ) }
</result>
```

We abbreviate the second `if`*-condition as χ and the string* `<remark>` $\cdots$ `</remark>` *as s for syntactic brevity. Directly rewriting Q into FluX (without algebraically optimizing)*

leads to the following FluX query F:

```
{ ps $ROOT:
  on-first past() return <result>;
  on bib as $bib return
   { ps $bib:
     on-first past(book) return
       { if (fn:not(fn:exists($bib/book/subtitle))) then s };
     on-first past(book) return
       { for $book in $bib/book return
         { if (χ) then {$book} } } };
  on-first past(bib) return </result>; }
```

When evaluating F on an input XML stream, all *books will be buffered. This will be circumvented if, prior to the transformation into FluX, the XQuery Q is algebraically optimized by using the rewrite rules of Figure 3.5: From the given DTD we know that books do not have subtitles. Therefore, the first* `if`*-condition in Q is always satisfied, and we can use the rule [Sat-If] to rewrite the first* `if`*-expression. Additionally, the DTD tells us that books do* either *have authors* or *editors. Therefore, the second* `if`*-condition* χ *in Q is never satisfied, and we can use rule [DTD-Impl] to rewrite this* `if`*-expression. Altogether, using the rules [DTD-Impl], [Simplify-If], [Sat-If], [Unsat-If], and [Silent-For], Q is rewritten into the XQuery* Q'

```
<result>
  { for $bib in $ROOT/bib return s }
</result>
```

which, by applying our rewriting algorithm, leads to the equivalent FluX query F'*:*

```
{ ps $ROOT:
  on-first past() return <result>;
  on bib as $bib return s;
  on-first past(bib) return </result>; }
```

Hence, no buffering is necessary when evaluating F' *on an XML stream compliant with the aforementioned DTD.* □

Although the algebraic optimizations in the former example might seem to be deviously, they are of importance when thinking of processing (possibly infinite) data streams with respect to user-friendliness. Consider a query containing a wrong path with respect to the DTD (probably due to a typing error) and hence yielding an empty result. Without our optimizations, users will probably wait a long time for the expected result, which obviously never will be computed, because the (non-existent) path cannot be matched on the data stream. Exploiting our optimizations, in the best case the query can be reduced in such a way that immediately an empty result can be delivered without even having to process the data stream.

With the algebraic optimization rules *[Merge1]* and *[Merge2]*, `for`-loops over the same path may be merged if it is known from a cardinality constraint of the DTD that they are executed at most once. Optimizing an XQuery Q by means of these rules leads to an equivalent XQuery Q' which, when transformed into FluX, can lead to a FluX query that needs significantly less buffering than the original query. Analogously, rule *[Simplify]* shortens condition paths and eliminates nested `for`-loops, again provided that the `for`-loop is executed only once. This will pay off when computing the dependency sets, since buffers will appear more locally. Altogether, the algebraic optimization rules often reduce the amount of buffering needed for processing a query. Consequently, the FluX query obtained from the optimized XQuery will be executed much more efficiently than that obtained from the original XQuery. Let us demonstrate this effect using the following examples.

Example 3.6.4 *Consider a DTD in the bibliography domain with the productions*

```
<!ELEMENT bib          (dissertation)*>
<!ELEMENT dissertation (title, author, pubyear)>
<!ELEMENT author       (lastname, firstname, address)>
```

and the XQuery:

```
<address>
  { for $d in $ROOT/bib/dissertation
    where $d/author/lastname = "Goedel"
    return
      $d/author/address }
</address>
```

Normalization of this query yields the following XQuery$^-$ *query* Q_2*:*

```
<address>
{ for $d in $bib/dissertation return
  { for $a in $d/author return
    { for $addr in $a/address return
      { if ($d/author/lastname = "Goedel") then {$addr} } } } }
</address>
```

Directly rewriting Q_2 *into FluX leads to the FluX query* F_2

```
{ ps $ROOT:
  on-first past() return <address>;
  on bib as $bib return
    { ps $bib:
      on dissertation as $d return
        { ps $d:
          on-first past(author) return
            { for $author in $d/author return
              { for $addr in $author/address return
                { if ($d/author/lastname = "Goedel") then {$addr} } } }; }; };
  on-first past(bib) return </address>; }
```

that needs to buffer the entire `author` *node (because of the "*`on-first past(author)`*"-handler).*

If we first rewrite Q_2 using the algebraic optimization rule [Simplify], which simplifies the `if`*-expression to "*`{if ($author/lastname = "Goedel") then ...}`*", and then transform the resulting query into FluX, we obtain the following FluX query F_2':*

```
{ ps $ROOT:
  on-first past() return <address>;
  on bib as $bib return
    { ps $d:
      on author as $author return
      { ps $author:
        on address as $addr return
          { if ($author/lastname = "Goedel") then {$addr} }; }; };
  on-first past(bib) return </address>; }
```

Note that for processing F_2' on a stream conforming to the DTD no buffering is needed at all. □

Example 3.6.5 *Consider again the DTD of the previous example and the following XQuery retrieving last and first names of all authors:*

```
<authors>
  { for $a in $ROOT/bib/dissertation
    return
      ($a/author/lastname,
       $a/author/firstname) }
</authors>
```

Normalizing this query yields the following XQuery$^-$ query Q_3:

```
1  <authors>
2  { for $bib in $ROOT/bib return
3    { for $d in $bib/dissertation return
4      { for $author in $d/author return
5        { for $ln in $author/lastname return
6          {$ln} } }
7      { for $author in $d/author return
8        { for $fn in $author/firstname return
9          {$fn} } } } }
10 </authors>
```

Directly rewriting Q_3 into FluX leads to the following FluX query F_3:

```
1  { ps $ROOT:
2    on-first past() return <authors>;
3    on bib as $bib return
4      { ps $bib:
5        on dissertation as $d return
6          { ps $d:
7            on author as $author return
8              { ps $author:
9                on lastname as $ln return {$ln}; };
10           on-first past(author) return
11             { for $author in $a/author return
12               { for $fn in $author/firstname return
13                 {$fn} } }; }; };
14   on-first past(bib) return </authors>; }
```

Because of the two consecutive `for`*-loops over authors in lines 4 and 7 of Q_3, when ignoring cardinality constraints we are only able to execute the first loop in a streaming fashion (using an* `on`*-handler), whereas the second loop has to work on previously buffered* `author` *nodes (by means of an "*`on-first`*"-handler).*

By applying rule [Merge1] on Q_3 the two `for`*-loops can be merged into a single loop. The resulting query Q'_3 is obtained from Q_3 by replacing lines 4–9 with:*

```
{ for $author in $a/author return
  { for $ln in $author/lastname return {$ln} }
  { for $fn in $author/firstname return {$fn} } }
```

Rewriting Q'_3 into FluX yields the following FluX query F'_3:

```
{ ps $ROOT:
  on-first past() return <authors>;
  on bib as $bib return
    { ps $bib:
      on dissertation as $d return
        { ps $d:
          on author as $author return
            { ps $author:
              on lastname as $ln return {$ln};
              on firstname as $fn return {$fn}; }; }; };
  on-first past(bib) return </authors>; }
```

Not only is F'_3 aesthetically preferable, but it can be executed without any buffering at all on streams conforming to the given DTD.

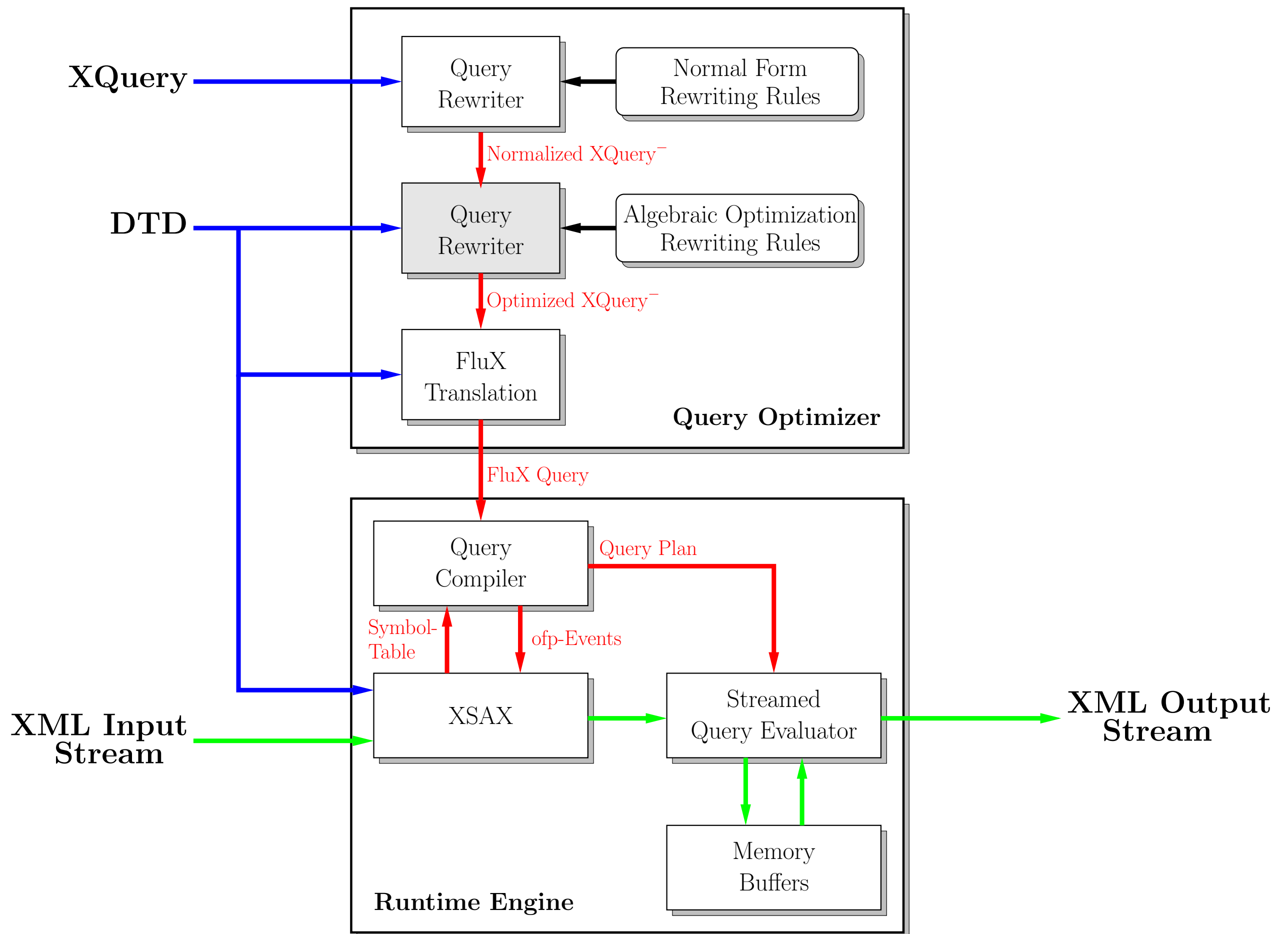


Figure 3.6: Architecture of the FluX Query Engine

3.7 Implementation

Figure 3.6 depicts the architecture of our FluX query engine with the query optimizer and a runtime engine as the main modules.

The query optimizer gets an XQuery and a DTD as input and transforms this query successively into a FluX query as described in the previous sections. The optional algebraic optimizations are shaded grey.

In this section, we will discuss various aspects of the runtime engine for evaluating FluX queries computed by the query optimizer, which are shown in the bottom half of the figure. We first explain the implementation of our SAX-based XML parser for producing "on-first-past"-events in Section 3.7.1. In Section 3.7.2 we show how a physical query plan is generated by the query compiler and how it is executed. Finally, we describe the handling of main memory buffers in Section 3.7.3.

3.7.1 The *XSAX Parser*

For the evaluation of FluX queries special “on-first-past”-events (ofp-events) are needed. Our *eXtended SAX (XSAX) parser* accomplishes this task. The XSAX parser is based upon a conventional SAX parser. Such a SAX parser produces a sequence of the following events while parsing an XML stream:

- Start of an element (with associated tag name and attributes)
- Character content (with associated content)
- End of an element (with associated tag name)

Our XSAX parser extends such a SAX parser by additionally feeding SAX events through an *ofp manager*. The ofp manager handles SAX events through to the FluX query engine and additionally inserts appropriate ofp-events into this stream of SAX events.

The basic mode of operation of our XSAX parser is the *full validation mode*. Given a DTD of the input stream, the ofp manager computes the Glushkov automaton for every production of the DTD as described in Section 3.3. Further, the query compiler analyzes the FluX query with respect to which ofp-events have to be generated in what productions and registers those ofp-events at the XSAX parser. The ofp manager then computes the past sets for each ofp-event and annotates the events at the corresponding transitions of the automata. Since making a single transition may trigger more than one ofp-event for different sets of symbols, ofp-events are encoded as bitmaps, where each bit corresponds to a unique ofp-event. If ofp-events have to be thrown on making a transition of an automaton, a bitmap with the corresponding bits set is handed to the query engine as a single ofp-event representing all ofp-events that have to be processed. Note that automata transitions can be annotated at query compile time, so ofp-events can be generated very efficiently at runtime by only simulating the automata. To be able to switch between the automata of all productions during parsing the input stream, the ofp manager maintains a stack of automata representing the current nesting level of the input stream. Therefore, the ofp manager handles SAX events as follows:

- Upon seeing a start element event “<*tagname*>”, the automaton on top of the stack is simulated by making the transition labeled *tagname* from its current state. If ofp-events are associated with this transition, they are thrown. If no transition can be made, the document is not valid according to the given DTD and an error is signaled. Afterwards, we enter the production of this tag by pushing the corresponding automaton on top of the stack.

- Upon seeing an end element event “</*tagname*>”, we simulate the automaton on top of the stack using input symbol “$\odot$”. Note that the end tag is a valid symbol in this automaton because of the construction described in Section 3.3. Afterwards, we remove this automaton from the stack, which represents a decrease in the current nesting depth of the XML input stream.

Having simulated the appropriate automata and possibly generated ofp-events, the original SAX event is forwarded to the query engine. Mixed content is managed by introducing a special symbol for character content (independent from the real content) and therefore is handled like any other symbol (i. e., a tag name). Since for every production in the DTD an automaton is built and simulated, the whole input stream is validated against the DTD.

The second mode of operation of our XSAX parser is the *lazy mode*. Obviously, only automata for those productions are needed, in which actually ofp-events have to be generated. Hence, in this mode of operation we only build and simulate automata for such productions. For all remaining productions a "dummy" automaton, which does not try to make a transition and does not throw any ofp-events, is used. First, this conserves main memory, because fewer automata have to be held in memory (in the worst case, i. e., no order constraints among the children, an automaton of a production has $\mathcal{O}(n^2)$ transitions with n being the number of children). Second, the efficiency of the ofp manager is improved, because the dummy automata do not simulate any transitions. Of course, only parts of the data stream are now validated against the given DTD. Since we assume that the input stream is valid with respect to the given DTD, this is the default mode of operation.

Another important task of the XSAX parser is online compression of the XML data stream. Using the given DTD, the XSAX parser assigns each production (and therewith, each possible tag name) a unique integer identifier. When forwarding SAX start/end element events to the query engine, the actual name of the tag is replaced by its identifier[6]. Additionally, the ofp manager works on these identifiers rather than using the original element names. SAX events representing character content remain untouched. This compression is similar to the dictionary-based compression of structural content in XMill [LS00]. The query engine benefits in two ways from this approach: First, the execution of `on`-handlers is improved, since we do not have to compare string values but only the integer identifiers for determining whether a handler matches an event, which obviously is much more efficient. Second, if the query engine buffers parts of the input document (which is realized as storing a list of SAX events—see Section 3.7.3 for details), only the identifier together with the type of the event has to be stored. This dramatically reduces main memory consumption. When outputting events, the query engine uses the symbol table of the XSAX parser to restore its original name.

Note also that the XSAX parser performs an on-the-fly conversion of attributes into elements as described in Appendix C.1.

3.7.2 Query Execution

In this section, we present how the runtime engine evaluates FluX queries by means of the *query compiler* and the *streamed query evaluator* depicted in Figure 3.6.

The query compiler transforms an optimized FluX query into a physical query plan by

[6]Actually, when forwarding SAX events we hand both its name and identifier to the query engine to avoid compressing in the parser and immediately uncompressing in the query engine, e. g., when directly outputting the input stream. Apart from that, the query engine only works with the compressed events.

rewriting the logical query operations into physical operators. Physical query operators are classified as follows:

Stream Operators This class of operators consists of operators handling streams, e. g., for controlling buffering or direct outputting of streams, for matching paths on a stream, and so on.

Buffered Operators This class basically contains all standard XQuery constructs working on buffered data as used in "`on-first`"-handlers. The most prominent representatives are operators for evaluating `for`-loops on buffers.

Common Operators These are general purpose operators that may be used both on streaming data and buffered data. Examples for such operators are conditional execution of subplans (representing `if`-statements) or computation of expressions (e. g., used in conditions).

3.7.2.1 Implementation of the "`process-stream`"-Statement

The most important extension of FluX compared to standard XQuery is the introduction of the process-stream construct for directly processing data streams. Every "`{ps` $\$x$`:` ... `}`"-statement defines a *scope*. A scope corresponds to a set of nodes at a certain depth of the tree representation of the input XML stream, which have a common parent node. Note that a certain depth in the XML tree may have different (sibling) scopes. For each scope of a query, i. e., for each "`{ps` $\$x$`:` ... `}`"-statement, a *scope-handler* is created, which processes the events it receives from the XSAX parser.

The functionality of a scope-handler slightly differs from the semantics of a `ps`-statement (with respect to the execution of handlers) as follows. The semantics of an "`on` a"-handler in FluX is defined such that on seeing an a node and processing the handler the whole subtree rooted at a is known. This is somewhat different when processing the input stream as a sequence of SAX (or, in detail, XSAX) events. Here, we receive an event corresponding to the opening tag `<`a`>` and a different event representing the closing tag `</`a`>` (of course, with the sequence of events corresponding to the subtree rooted at a in between). Hence, for a single "`on` a"-handler of a `ps`-statement two handlers, one for the start and the other for the end tag event, have to be managed in a scope-handler. The query compiler appropriately schedules physical query operators into the start tag handler, the end tag handler, or both of them. For instance, consider the case that we want to buffer the whole subtree rooted at a. Then, the query compiler schedules a physical operator for starting buffering in the start tag handler of a and another operator for turning buffering off in the end tag handler of a. "`on-first`"-handlers and events are handled by a scope-handler exactly as defined in Section 3.4.2.

Using the physical operators and scope-handlers, the query compiler is able to construct a query plan, which consists of a set of nested scope-handlers containing all the physical query operators derived from the optimized FluX query in the appropriate event handlers. Additionally, the query compiler registers all ofp-events occurring in a FluX query at the

XSAX parser and utilizes the symbol table of the XSAX parser for substituting all labels of elements by their identifier.

3.7.2.2 Execution of a Query Plan

We have the following two possibilities for executing the query plan compiled by the query optimizer:

- The query plan is directly interpreted by means of the streamed query evaluator.
- The query plan can be transformed into native Java code, which eventually yields a directly executable query.

Both approaches share common core functionality. Hence, in the remainder we focus on describing how the streamed query evaluator executes a query plan.

The streamed query evaluator maintains a stack of scope-handlers representing the current nesting of the XML input stream. XSAX events are always passed to the scope-handler on top of the stack, i. e., to the current scope. This scope-handler executes the appropriate handlers (and associated operators). If a new scope is entered, a scope-handler for the new scope is generated, pushed on top of the stack, and all "`on-first past()`"-handlers are executed. From now on, this new handler receives all XSAX events and processes them accordingly, which is equivalent to descending one level in the tree representation of the input stream. Upon seeing the end tag event of the symbol associated with the current scope, it is left by executing all "`on-first past(*)`"-handlers and removing the topmost scope-handler from the stack. Analogously, this is equivalent to ascending one level in the XML tree. In special cases events are handed from the current scope to parent scopes. For instance, if a parent scope has turned on buffering of the stream, the current scope-handler also propagates all events it receives to the parent scope-handler, so that it is able to populate its buffers accordingly. This is, however, only an implementation subtlety.

3.7.2.3 Condition Evaluation

Another interesting aspect of the query compiler/streamed query evaluator is the handling of `if`-conditions.

Due to the normalization, `if`-statements (or, conditional operators in the query plan) are always pushed down to the "leaves" of the query tree. This has the effect that often a single `if`-expression of the query is translated into multiple `if`-statements with equal conditional expressions (this can be easily observed in the prior examples). At first glance, this seems to be rather poor with respect to efficiency. To avoid multiple computation of conditions in such situations we separate the evaluation of the condition itself from the execution of an `if`-expression (or, a conditional operator in the query plan). That is, the query compiler schedules an operator evaluating the conditional expression such, that this operator is executed (and therewith, the expression is evaluated) right before the first conditional operator actually needs the result. The result of the condition is stored in a

Boolean flag. Then, all conditional operators only check this flag rather than repeatedly computing the conditional expression. This can be applied for all types of conditional expressions and only implies a very low additional need for main memory consumption.

However, there is a striking difference between the evaluation of (join) conditions and *streaming atomic conditions*.

Definition 3.7.1 (Streaming Atomic Condition) *A* streaming atomic condition *is an atomic condition of the form*

$$\$x/\pi \ RelOp \ s$$

with $\$x$ *being a variable bound by an* `on`*-handler,* π *a (non-empty) fixed path, and* s *a constant string.*

Streaming atomic conditions may be checked on-the-fly and the values of the fixed path π do not have to be buffered. In detail, whenever a value of the corresponding path is seen on the stream, the condition is immediately checked and the result is stored in a Boolean flag for use in the appropriate conditional operators. Since our rewriting algorithm always generates safe queries, it is assured that all nodes which need to be checked for correctly evaluating the expression have been seen before the result of the expression is used for the first time in a conditional operator. In other words, the rewriting algorithm delays the execution of `if`-statements, until all necessary data has been seen[7].

3.7.2.4 Examples

For the sake of briefness, we refrain from giving a more detailed description of the query compiler and the streamed query evaluator and provide an instructive example instead.

Example 3.7.2 *Consider again query XMP-Q1 introduced in Example 3.5.1 and the FluX query* F_1' *generated in the optimal case as described in Example 3.5.4. The query engine pursues the evaluation strategy, which has been computed by the query compiler, as shown in Figure 3.7.*

Scope-handlers are denoted as `PSNode` *(corresponding to a scope, see lines 1, 8, and 17) and handle the events* `startStream`*,* `endStream`*,* `on-first`*, and start/end tags of elements*[8]*.* `startStream`*- and* `endStream`*-handlers are executed when entering and leaving, respectively, the scope and correspond to "*`on-first past()`*"- and "*`on-first past(*)`*"-handlers in FluX. Handlers for start and end tag events are separated into the* `startTag` *and* `endTag` *sections as described before.*

Query processing starts by executing the `startStream`*-handler of the main scope (line 2), which outputs the start tag* `<bib>` *(line 3). When receiving the events* `<bib>` *and* `<book>`*, the scope of a book is entered (line 17) and the corresponding content of a book*

[7]This is usually accomplished by placing the corresponding statement(s) into an appropriate "`on-first`"-handler if the DTD does not assure the correct order of elements.

[8]For presentation purposes, all handlers are shown with the real labels of the symbols. Of course, internally only the corresponding identifiers are used.

```
1  PSNode[symbol=__ROOT]
2    * startStream:
3      Output[data=<bib>]
4    * endStream:
5      Output[data=</bib>]
6    * startTag:
7      on 'bib':
8        PSNode[symbol=bib]
9          * startStream:
10           (NOP)
11         * endStream:
12           (NOP)
13         * startTag:
14           on 'book':
15             SequenceNode
16               BufferStream[bufID=0; state=true]
17               PSNode[symbol=book]
18                 * startStream:
19                   SequenceNode
20                     BufferStream[bufID=0; state=false]
21                     InitializePathMatcher[path=publisher; AtomicCondition: 1]
22                     InitializePathMatcher[path=year; AtomicCondition: 3]
23                 * endStream:
24                   BufferStream[bufID=0; state=true]
25                 * startTag:
26                   on 'title':
27                     Conditional[conditionID=4]
28                       OutputStream[state=true]
29                   on 'year':
30                     BufferStream[bufID=0; state=true]
31                 * endTag:
32                   on 'title':
33                     Conditional[conditionID=4]
34                       OutputStream[state=false]
35                   on 'year':
36                     BufferStream[bufID=0; state=false]
37                 * on-first:
38                   OnFirstHandler {year, publisher}:
39                     SequenceNode
40                       Conditional[conditionID=4]
41                         Output[data=<book>]
42                       BufferedFor[var=$year; baseVar=$b; step=year, buffer=0]
43                         Conditional[conditionID=4]
44                           BufferedOutput[var=$year]
45                   OnFirstHandler {year, title, publisher}:
46                     Conditional[conditionID=4]
47                       Output[data=</book>]
48         * endTag:
49           on 'book':
50             SequenceNode
51               InitializeBuffer[bufID=0]
52               BufferStream[bufID=0; state=false]
53         * on-first:
54           (NOP)
55   * endTag:
56     (NOP)
57   * on-first:
58     (NOP)
```

Figure 3.7: Query Plan for Query XMP-Q1

can be processed. Upon entering this scope, the sequence of operators in lines 20–22 is executed. The first manages the needed buffering of the `year` *nodes. Details on buffering will be presented in the next section and we will refrain from describing the* `BufferStream` *operators in the following. The* `InitializePathMatcher` *operators in lines 21 and 22 handle the evaluation of the condition*

```
$b/publisher = "Addison-Wesley" and $b/year > 1991
```

In detail, it consists of two streaming atomic conditions, which can be checked on-the-fly. For each condition the following is accomplished by the `InitializePathMatcher` *operator:*

- *A Boolean flag is created, which stores the result of the atomic condition, and initialized to* `false`*.*
- *An automaton corresponding to the path that has to be checked (in this example,* `publisher` *or* `year` *relative to the current* `book` *node) is set up, which receives the incoming events. Whenever this automaton is in an accepting state, i. e., a match with the given path has occurred, it executes a specific action. In this case, the action is to evaluate the given atomic condition on the current content of the path by examining the character data read on the stream without the explicit use of buffers. If, at some point in time, an atomic condition already has been evaluted to* `true`*, no further evaluation is performed.*

Hence, while receiving XSAX events of a `book` *node the two atomic conditions are transparently checked on-the-fly. Eventually, an ofp-event signaling that no further* `publisher` *and* `year` *nodes can be encountered anymore triggers the execution of the lines 39–44. If the composite condition evaluates to* `true`*, the* `Conditional` *operator executes line 41 and hence prints the opening tag* `<book>` *to the output. The* `BufferedFor` *operator (line 42) loops over all (previously buffered)* `year` *nodes and the* `BufferedOutput` *writes the current binding to the output if the condition is fulfilled (lines 43–44). From now on, we may receive* `title` *nodes (and not before, because of the order constraints of the DTD). Whenever a start tag event* `<title>` *is processed, we signal the scope-handler to turn on outputting of the incoming data stream, if the condition is fulfilled (lines 27–28). Analogously, directly writing the stream to the output is terminated at the closing tag event* `</title>` *(lines 33–34). When no further titles can be encountered anymore, the closing tag is written to the output by means of the ofp-handler in lines 45–47. Upon receiving the end tag event* `</book>` *the scope of book is left and all buffers needed in this scope are cleared (lines 49–52).*

Entering, processing, and leaving the `book`*-scope continues until the data stream ends. Then, query processing terminates by executing the end stream handler in the main scope in lines 4–5 and printing the closing tag* `</bib>` *of the result.* □

The following example briefly shows, how non-streaming conditions are handled during the evaluation of a FluX query.

```
1  * on-first:
2    OnFirstHandlerNode {article, book}:
3      BufferedFor[var=$article; baseVar=$bib; step=article, buffer=0]
4        BufferedFor[var=$book; baseVar=$bib; step=book, buffer=0]
5          SequenceNode
6            BufferedConditionCheck[conditionID=2]
7            Conditional[conditionID=2]
8              Output[data=<result>]
9            BufferedFor[var=$author; baseVar=$article; step=author]
10             Conditional[conditionID=2]
11               BufferedOutput[var=$author]
12           Conditional[conditionID=2]
13             Output[data=</result>]
```

Figure 3.8: Fragment of Query Plan for Example 3.5.5

Example 3.7.3 *Consider the FluX query F_3 shown in Example 3.5.5. This query contains the (non-streaming) join condition*

`$article/author = $book/editor`,

which is processed inside an "`on-first past(book, article)`*"-handler. Figure 3.8 shows the relevant fragment of the query plan for this handler inside the* `bib` *scope. The operators* `BufferedFor`, `Conditional`, *and* `BufferedOutput` *directly reflect the structure of the XQuery⁻ expression in lines 6–11 of F_3. The condition itself is evaluated by means of the operator* `BufferedConditionCheck` *in line 6 of Figure 3.8, which is scheduled right before the result of the condition is needed for the first time in a* `Conditional` *operator, i. e., in line 7. This operator iterates over the current bindings of the variables* `$article` *and* `$book` *contained in the buffer and sets a flag corresponding to the current result of the condition. The* `Conditional` *operators (in lines 7, 10, 12) then only read this flag to determine whether to execute the corresponding subplan, rather than evaluating the condition each time.* □

3.7.3 Buffer Management

In this section, we discuss the allocation of buffers and their use during query evaluation. Given a FluX query, we statically infer the buffers which are necessary in order to avoid superfluous buffering. Our pre-filtering techniques generalize those of [MS03] to the scenario where certain parts of the input do not need to be buffered—even though they are used by the query—because they can be processed on-the-fly.

Buffers are implemented as lists of SAX events[9]. A buffer is always associated to its corresponding variable, which must be bound by an `on`-handler and processed by a `ps`-statement. Hence, it is tied to the scope of this variable. On entering this scope the buffer

[9] Note that ofp-events are not needed for "classic" evaluation of XQuery over buffers inside ofp-handlers and hence it is sufficient to buffer SAX events.

is initialized and cleared on leaving it. The events stored in a buffer represent well-formed XML in the sense that start/end element events are properly nested within each other. This renders data read from (a stream replayed from) a buffer indistinguishable from data read from the input stream. In our implementation, we employ a common set of operators for handling both events originating from streams and buffers, e. g., for condition evaluation[10].

In the following, we say that a FluX query is in normal form if all of its (maximal) XQuery$^-$ subexpressions are in normal form.

Let Q be a safe FluX query in normal form. The FluX query engine identifies all nodes that must be stored in buffers, i. e., all nodes compared in non-streaming atomic conditions, e. g., a join condition, the roots of buffered subtrees that are output, and buffered nodes over which `for`-loops iterate. The following definition formalizes this idea of buffered paths.

Definition 3.7.4 (Buffered Paths) *Let α and β be XQuery$^-$ subexpressions of Q. We define $\Pi(\$r, \alpha)$, the set of all buffered paths in α starting with variable $\$r$, as*

$$\begin{array}{rcl}
\Pi(\$r,\ \epsilon) = \Pi(\$r, s) & := & \emptyset \\
\Pi(\$r,\ \texttt{\{}\$r\texttt{\}}) & := & \{\$r\} \\
\Pi(\$r,\ \alpha\beta) & := & \Pi(\$r,\ \alpha)\ \cup\ \Pi(\$r,\ \beta) \\
\Pi(\$r,\ \texttt{\{for}\ \$x\ \texttt{in}\ \$y/a\ \texttt{return}\ \alpha\texttt{\}}) & := & \Pi(\$r,\ \alpha) \\
 & & \quad \cup\ \{\$r/a \mid \$y = \$r \wedge \Pi(\$x,\ \alpha) = \emptyset\} \\
 & & \quad \cup\ \{\$r/a/w \mid \$y = \$r \wedge \$x/w \in \Pi(\$x,\ \alpha)\} \\
\Pi(\$r,\ \texttt{\{if}\ \chi\ \texttt{then}\ \alpha\texttt{\}}) & := & \Pi(\$r,\ \alpha) \\
 & & \quad \cup\ \{\$r/\pi \mid \chi \textit{ contains a non-streaming} \\
 & & \qquad\qquad \textit{atomic condition with} \\
 & & \qquad\qquad \textit{a path } \$r/\pi\}
\end{array}$$

For a variable $\$r$ and a safe FluX query Q in normal form, we now define $\Pi(\$r)$ as

$$\Pi(\$r) := \bigcup_{\alpha} \Pi(\$r,\ \alpha)$$

with α being a maximal XQuery$^-$ subexpression of Q.

Let $\mathcal{T}(\$r)$ be the prefix tree constructed by merging all paths from $\Pi(\$r)$. Intuitively, the prefix tree defines a projection of the input document, as it describes which parts of the input tree will be buffered.

We optimize the prefix tree in order to restrict the amount of data being buffered as follows. Let $\$r/\pi = \$r/\ldots/a \in \Pi(\$r)$. Now, let $\mathcal{T}^m(\$r)$ be the tree obtained from $\mathcal{T}(\$r)$ by marking the node labeled a if $\$r/\pi$ does *not* stem from the set

$$\{\$r/a \mid \$y = \$r \wedge \Pi(\$x,\ \alpha) = \emptyset\}$$

[10]Thus, physical query evaluation proceeds in a way similar to that followed in XQRL [FHK$^+$00].

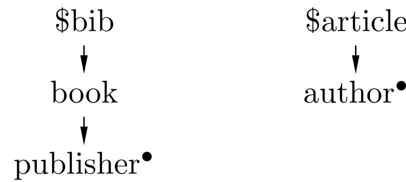


Figure 3.9: Buffer Trees of Example 3.7.5.

contained in Definition 3.7.4 of buffered paths of a `for`-expression. Therefore, marked nodes are buffered together with their entire subtree. For unmarked nodes in $\mathcal{T}^m(\$r)$, we merely store the SAX events for the opening and the closing tag.

Clearly, if a node is marked and we buffer it together with its subtree, we also buffer the subtrees of any descendant nodes at the same time. Thus we only buffer the data of the topmost marked nodes in $\mathcal{T}^m(\$r)$. For example, if we need to buffer two subtrees reachable by paths π and π' respectively, where π is a prefix of π', we restrict ourselves to buffering the subtree identified by π. Let $\mathcal{T}^p(\$r)$ be the pruned prefix tree obtained from $\mathcal{T}^m(\$r)$ by successively removing a subtree rooted at a node v' if an ancestor node v is marked. We refer to $\mathcal{T}^p(\$r)$ as a *buffer tree*.

Without loss of generality, we assume that variables in queries are used uniquely, i. e., each variable name is bound at most at one place in the query. For a safe FluX query Q in normal form, let X be the set of variables that are free in maximal XQuery$^-$ subexpressions of Q. The variables in X are precisely those for which we will later define buffers.

Example 3.7.5 *The following FluX query selects all book publishers whose CEO has published articles.*

```
{ ps $ROOT:
  on bib as $bib return
    { ps $bib:
      on article as $article return
        { ps $article:
          on-first past(author) return
            { for $book in $bib/book return
              { for $p in $book/publisher return
                { if $article/author = $book/publisher/ceo
                  then {$p} } } }; }; }; }
```

Here, $X = \{\texttt{\$bib}, \texttt{\$article}\}$ *and we compute the sets of buffer paths for each variable in* X *as:*

$$\begin{aligned}\Pi(\texttt{\$bib}) &= \{\texttt{\$bib/book/publisher/ceo},\\ &\quad\ \texttt{\$bib/book/publisher}\}\\ \Pi(\texttt{\$article}) &= \{\texttt{\$article/author}\}\end{aligned}$$

We construct a buffer tree for each variable in X with a nonempty set of buffer paths. Here, we obtain the trees shown in Figure 3.9 (the bullet denotes a marked node). Note that the leaf node `ceo` *has been pruned off the buffer tree of variable* `$bib`*.* □

With the next example we show how the additional optimizations of the buffer tree work.

Example 3.7.6 *Consider the following FluX query, which only prints an* `<aBook/>` *element for each book:*

```
{ ps $ROOT:
  on-first past(bib) return
    { for $bib in $ROOT/bib return
      { for $book in $bib/book return
          <aBook/> } }; }
```

Note that this query is safe, but for presentation purposes it has not been obtained through our algorithm and hence it is not optimal with respect to minimal buffer usage (in this case, no buffering would be required at all).

Here, $X = \{\texttt{\$ROOT}\}$ *and* $\Pi(\texttt{\$ROOT}) = \{\texttt{\$ROOT/bib}, \texttt{\$ROOT/bib/book}\}$*. The optimized buffer tree* $\mathcal{T}^p(\$ROOT)$ *consists only of the unmarked path* `$ROOT/bib/book`*. Hence, for each book only the sequence of events* `<book></book>` *will be buffered (nested inside the surrounding* `bib` *events). This is sufficient for being able to iterate over all books and print the tag* `<aBook/>`*, since only the number of books, but not their actual contents, is needed.* □

In other words, the buffer tree of a variable $\$r$ can be considered a schema for all events stored in the associated buffer.

The query compiler prepares buffering of nodes to be able to evaluate a safe FluX query Q in normal form as follows. At first, it computes the set X of Q and constructs the buffer tree $\mathcal{T}^p(\$r)$ for each variable $\$r \in X$. As already said at the beginning, each variable corresponds to a scope. Hence, in each scope belonging to some $\$r$ a buffer is set up. This buffer is initialized on entering the scope of $\$r$ and freed after leaving it by adding appropriate physical query operators to the query plan. To realize the selective buffering, in all needed subscopes of $\$r$ physical query operators for turning buffering on and off are scheduled at the appropriate start and end tag events such that the buffer is correctly filled. In cases where no such subscopes exists (because the query itself does not need them), we introduce new variables and scopes accordingly. As there is at most one (start and end tag) event-handler for a given node label, it is clear where corresponding commands are to be introduced.

Example 3.7.7 *Consider again the query plan of Example 3.7.2 depicted in Figure 3.7. For this query,* $X = \{\texttt{\$b}\}$ *and* $\mathcal{T}^p(\$b)$ *consists only of the path* `$b/title`$^\bullet$ *with a marked* `title` *node. Recall, that the atomic condition referring to publishers can be checked on-the-fly and thus no publishers have to be buffered. To ensure correctly buffering of the* `year` *nodes of the stream, the query compiler schedules physical operators as follows.*

Before entering the scope of $b, buffering is turned on by means of the `BufferStream` *operator (line 16) to ensure, that the events of the surrounding* `book` *node are buffered*[11]*. Upon entering the scope, buffering is immediately turned off again in the* `startStream`*-handler to achieve that no unnecessary subsequent events are buffered (line 20). Upon receiving a* `<year>` *event, buffering is turned on again (line 30). Analogously, when receiving the* `</year>` *event buffering is switched off (line 36). Because of that, all events belonging to the whole* `year` *subtree (including the opening and closing* `year` *events) are buffered. Eventually, in the* `endStream`*-handler of $b buffering is turned on to assure that the closing tag* `</book>` *is correctly buffered (line 24).*

After having left the scope of $b (and thus not needing the buffer anymore), in the end tag handler of a book the buffer is freed (line 51). □

Due to the computation of the set X and associating buffers to scopes it is possible, that some nodes are buffered more than once. The following example illustrates this situation.

Example 3.7.8 *Consider the following FluX query:*

```
{ ps $ROOT:
  on bib as $bib return
    { ps $bib:
      on book as $book return
        { ps $book:
          on-first past (title, author) return
            { for $title in $book/title return {$title} }
            { for $author in $book/author return {$author} }; };
      on-first past(book) return
        { for $book in $bib/book return
          { for $title in $book/title return {$title} } }; }; }
```

Note that this safe FluX query is again written by hand only for presentation purposes. For the sake of brevity we have left out the construction of surrounding tags in the result (and thus, the result might not be well-formed XML).

In this example, $X = \{$`$bib, $book`$\}$ *and hence we define buffers in the* `bib` *scope and the* `book` *scope. The buffer in the* `bib` *scope contains* all books *together with their titles and no other nodes such as* `author`*. The buffer of* `book` *buffers* only for each book *titles and authors. Hence, for the book currently being processed the* `title` *node is buffered twice.*

Of course, this situation could be avoided by re-using buffers. In the former example, the `bib` buffer could also be used inside the `book` scope to avoid double-buffering of `title` nodes. However, this complicates buffer management: In the superordinate buffer the union of the schemas of both buffers would have to be stored. In our example, the `bib`-buffer would have to store titles and authors of books. If we do it this way, we would buffer too many nodes, since the `author` node is only needed for the current book and not for

[11] Note that in this case storing the tags of the surrounding scope would not be needed, but it is required for being able to handle the extensions shown in Section 3.8.

all books. Hence, after having processed a book, the superordinate buffer would have to be cleaned, such that only that nodes, i. e., titles, are contained, which are actually needed later on for all books. Since this cleaning of buffers would introduce additional overhead, we do not pursue this approach and accept multiple buffering of nodes. However, this is not much of a drawback, since it only occurs in situations where a superordinate scope buffers all instances of a certain node n and only for each instance of n a subordinate scope buffers some children of n. Hence, the amount of wasted memory can be expected to be generally very low.

3.8 Extending the FluX Query Language

In this section, we present some principles and examples on how to extend our XQuery$^-$ fragment of XQuery. In Section 3.8.1 we will show how aggregate functions can be handled. In Section 3.8.2 we augment our fragment with novel syntactical constructs for dealing with data windows on streams.

3.8.1 Aggregate Functions

Computing aggregates is a very common task in data streams processing, especially in the field of sensor networks. In the following, we will show the basic steps of extending our FluX query engine by means of the example of aggregate operations: First we will extend the XQuery$^-$ fragment. Then, we will show how to adapt the syntax and semantics of FluX. Finally, we will present the evaluation strategy implemented in the query compiler and streamed query evaluator.

3.8.1.1 Extending the XQuery$^-$ Fragment

In the following, we denote an XQuery aggregate function [W3C05d] or an user-defined function (UDF) as *function*. All XQuery aggregate functions, i. e., `fn:count`, `fn:avg`, `fn:max`, `fn:min`, and `fn:sum`, are supported. To achieve maximum flexibility, certain UDFs are also allowed. In detail, supported UDFs are supposed to have the following signature using the standard XQuery notation for function signatures [W3C05d]:

```
udf:function-name($arg as xdt:anyAtomicType*) as xdt:anyAtomicType
```

UDFs are simply identified by the namespace prefix "`udf:`" and handled appropriately by the query optimizer and the streamed query evaluator[12].

To be able to correctly handle all kinds of functions—in detail, all kinds of user-defined functions—we classify functions into *holistic* and *non-holistic* (i. e., distributive or algebraic) functions. The result of non-holistic functions, e. g., `fn:sum` or `fn:avg`, can be

[12]Note that we do not yet use the XQuery syntax "`declare function ...`" for declaring UDFs as external functions.

```
Function    ::= ( "fn:" ("count" | "avg" | "max" | "min" | "sum" )
                | "udf:" QName ) "(" FixedPath ")"
ValExpr     ::= ( Literal
                | FixedPath
                | Function
                | ValExpr ( "+" | "-" ) ValExpr
                | ValExpr ( "*" | "div" | "idiv" | "mod" ) ValExpr )
Expression  ::= ( "()"
                | Literal
                | Expression ( Expression )+
                | "{" "for" VarRef "in" FixedPath ( "where" AtomicCondition )?
                    "return" Expression "}"
                | "{" "let" VarRef ":=" Function ( "where" AtomicCondition )?
                    "return" Expression "}"
                | "{" FixedPath "}"
                | "{" VarRef "}"
                | "{" Function "}"
                | "{" "if" "(" AtomicCondition ")" "then" Expression "}" )
```

Figure 3.10: XQuery^{-} Grammar Fragment for Handling Aggregate Functions

computed or updated with every new data value that is seen on the stream[13]. In case of a holistic function, e. g., the median of a set of values, all values have to be known at once to be able to correctly compute the function result. Note that all built-in XQuery functions are non-holistic.

Using this notion of functions, we extend the original grammar of XQuery^{-} as shown in Figure 3.10. For the sake of brevity, we only show new or changed productions with respect to Figure 3.2 and we will refer to this new grammar also as XQuery^{-} in the rest of this thesis. The definition of value expressions (Definition 3.4.2) and of XQuery^{-} (Definition 3.4.5) can be easily extended with respect to the novel XQuery^{-} grammar and will not be shown.

Within this language, functions can be used in `if`-conditions, their result can be printed, or their result may be assigned to some variable $\$x$ by means of the novel `let`-expression. Note that the `let`-expression only constitutes syntactical sugar, since all occurences of $\$x$ bound by this expression can be substitutet by the original function. Because of our definition of the `let`-expression, a variable $\$x$ cannot be bound to a set of nodes in this fragment. The definition of functions also implies that they cannot be applied to intermediary results of a query, but only directly on data of the XML stream.

[13]Note that in some cases not only the current result of the function has to be held in memory, but a *set* of values (with constant size). For instance, in case of `fn:avg` the current sum *and* the number of values is stored to be able to correctly update and compute the function result at any point in time.

Again, this new language XQuery$^-$ is very similar to a (now bigger fragment being of relevance for processing data streams) of standard XQuery [W3C05b]. As for the original XQuery$^-$ fragment, this new fragment still differs to XQuery in the treatment of fixed strings. Besides that, the semantics of the novel XQuery$^-$ fragment is equal to the semantics of XQuery and the document stream produced by evaluating an XQuery$^-$ query is the same as that of an equivalent XQuery.

3.8.1.2 Syntax and Semantics of FluX Revisited

In this section, we will present the needed changes to the syntax and semantics of the FluX query language in order to correctly handle (aggregate) functions introduced above. Normally, functions are handled inside "`on-first`"-handlers to ensure that all elements needed for the computation of the function have been seen. In some special cases, i. e., if functions are contained in simple expressions and certain order constraints are satisfied, functions are also processed in `on`-handlers. Hence, no additional syntactical constructs are needed[14]. However, some definitions of Section 3.4 have to be adapted to be able to handle the novel constructs of the extended XQuery$^-$ language. Note that we do not address the treatment of the new `let`-expression in the remainder of this section and show how to handle it in the next section.

At first, we have to revise the definition of simple conditions (Definition 3.4.9) as follows.

Definition 3.8.1 (Simple Condition) *Let χ be a condition as defined in Definition 3.4.9. Then, χ is denoted as a* simple condition*, if*

- *χ is an atomic condition "s RelOp s'" or "$\$x/\pi$ RelOp s", where s, s' are strings (constants), $\$x$ is a variable, and π is a fixed path.*
- *χ is an atomic condition "$f(\$x/\pi)$ RelOp s" or "$f(\$x/\pi)$ RelOp $f'(\$y/\pi')$" with s being a string (constant), $\$x$, $\$y$ variables, f, f' non-holistic functions, and π, π' fixed paths.*
- *χ is a Boolean combination (using "`and`", "`or`", and "`fn:not`") of simple conditions.*

In other words, beyond the old definition, a simple condition may now contain comparisons of constant values with function results.

An important characterization of expressions have been simple expressions (Definition 3.4.10). A simple expression can be evaluated on the current stream without having to buffer (parts) of the input stream. Dealing with functions, the definition of simple

[14]Remember, that inside an "`on-first`"-handler only XQuery$^-$ expressions (including functions) are used. Further, simple expressions inside an `on`-handler are also XQuery$^-$ expressions.

expressions has to be revised as follows.

Definition 3.8.2 (Simple Expressions) *Let* $\overline{\alpha}$ *be an XQuery*$^-$ *expression of the form* $\overline{\alpha} := \alpha\, \beta\, \gamma$. *Further, let* $\$x := parentVar(\overline{\alpha})$. $\overline{\alpha}$ *is* simple*, if all of the following hold:*

- α *and* γ *are possibly empty sequences of expressions "*δ*" and of expressions of the form "*`{if` χ `then` δ`}`*", where* χ *is a simple condition and* δ *is an expression of the form*
  - *"s", with s being a string (constant), or*
  - *"*`{`$f(\ldots)$`}`*", with* f *being a function as defined above.*
- β *is either empty, "*`{`$\$x$`}`*", or "*`{if (`$\chi$`) then {`$\$x$`}}`*" for a variable* $\$x$ *and a simple condition* χ*.*
- *if* β *is of the form "*`{`$\$x$`}`*", or "*`{if` χ `then {`$\$x$`}}`*" then* $\$x$ *does not occur in any condition or parameter of a function in* $\alpha\, \beta$*.*

The following example motivates the need to adapt the definition of simple expressions for correctly treating functions.

Example 3.8.3 *The expression* α

```
<a>{$x}</a> {if (fn:count($x/b) = 5) then <b>5</b>}
```

with $parentVar(\alpha) := \$x$ *is a simple expression, since the number of* `b` *elements can be counted during outputting* $\$x$ *and afterwards the* `if`*-expression can be correctly executed.*

In contrast, the expression β

```
<numb>{fn:count($x/b)}</numb> <a>{$x}</a>
```

is not simple, because the number of `b` *elements has to be printed* before *outputting the whole stream of* $\$x$*. Hence, during counting* `b` *elements the stream of* $\$x$ *has to be buffered for output later on.* □

The definition of free variables of a XQuery$^-$query Q must be slightly extended to cover the novel statement for outputting function results. For the sake of brevity, in the following we show definitions in an "overloaded" style, where only changes and additional constructs are shown.

Definition 3.8.4 (Free Variables) *Let the set of* free variables *be defined as in Definition 3.4.13. Additionally, let*

$$free(f(\$x/\pi)) \;:=\; \{\$x\},$$

with f *being a function as defined above.*

We will show how to deal with `let`-expressions later on.

Analogously, the definition of the set of dependencies with respect to a given variable is extended as follows.

Definition 3.8.5 (Dependencies) *Let the set of dependencies with respect to a variable $\$y$ of a XQuery⁻ expression be defined as in Definition 3.4.17. Further, let*

$$dep(\$y;\ \texttt{\{}f(\alpha)\texttt{\}}) := dep(\$y;\ f(\alpha))$$

where f is a function as defined at the beginning of this section or as defined in Definition 3.4.17.

Using these "overloaded" definitions covering functions, we may re-define the notion of safe queries as follows.

Definition 3.8.6 (Safe Queries) *A FluX query Q is called* safe *with respect to a given DTD if and only if for each subexpression "`{ps` $\$y$`:` ζ `}`" of Q, the following two conditions are satisfied:*

1. *For each handler "`on-first past(`S`) return` α" in the list ζ, the following is true:*
   - *$\forall\ b \in dep(\$y, \alpha)$ we have:*

     $$(b \in S) \vee (\exists a \in S : Ord_{\$y}(b, a))$$

   - *$\forall\ \$z \in free(\alpha)$ such that "$\{\$z\}$" $\preceq \alpha$, "$\{\$z/\pi\}$" $\preceq \alpha$, or "$\{f(\$z/\pi)\}$" $\preceq \alpha$ (for some π and function f) we have:*

     $$(\$z = \$y) \wedge (\forall\ b \in symb(\$y) : (b \in S) \vee (\exists a \in S : Ord_{\$y}(b, a))).$$

2. *For each handler "`on` a `as` $\$x$ `return` $\tilde{Q}$" in the list ζ, and for each maximal XQuery⁻ subexpression α of $\tilde{Q}$, the following is true:*
   - *$\forall\ b \in dep(\$y, \alpha)$ we have:*

     $$Ord_{\$y}(b, a)$$

   - *if $\alpha = \tilde{Q}$ (note that according to Definition 3.4.12 α must then be* simple*), then for all $\$u$ such that $\{\$u\} \preceq \alpha$ we have:*

     $$\$u = \$x.$$

Note that this definition is basically equal to that of Definition 3.4.18. In detail, only the expression for outputting function results has been added to the second bullet of (1).

$$\frac{\texttt{\{let } \$x \texttt{ := } f(\$y/\pi) \texttt{ where } \chi \texttt{ return } \beta\texttt{\}}}{\texttt{\{let } \$x \texttt{ := } f(\$y/\pi) \texttt{ return \{if (}\chi\texttt{) then } \beta\texttt{\}\}}} \quad \textit{[Elim-Where2]}$$

$$\frac{\texttt{\{let } \$x \texttt{ := } f(\$y/\pi) \texttt{ return } \beta\texttt{\}}}{\beta[\$x \rightarrow f(\$y/\pi)]} \quad \textit{[Elim-Let]}$$

Figure 3.11: Additional Normal Form Rewrite Rules for Extended XQuery^{-}

3.8.1.3 Rewriting Extended XQuery^{-} into FluX

In this section, we will show how to rewrite queries of the extended XQuery^{-} fragment into FluX. First, we will present how to deal with the novel `let`-expression, which has not been addressed yet. Afterwards, we will give some examples for illustrating the transformation of XQuery^{-} into FluX.

Obviously, the novel `let`-expression only constitutes syntactical sugaring in the sense, that it is more convenient to assign the result of a function to a variable if it is used more than once in a query. Hence, we handle an expression

$$\texttt{\{let } \$x \texttt{ := } f(\$y/\pi) \texttt{ return } \alpha\texttt{\}}$$

by simply eliminating this expression and substituting every occurrence of $\$x$ in α by $f(\$y/\pi)$ during the normalization step of a XQuery^{-} query. This is accomplished by the additional normal form rewriting rules shown in Figure 3.11. This approach might seem to perform worse with respect to efficiently evaluating the normalized query at first glance, since in a naive evaluation approach one evaluation of the function is replaced by having to evaluate each occurrence separately. However, our approach dramatically simplifies rewriting such a normalized query into FluX. We will show in the next section how to efficiently handle this situation.

Apart from adding the novel normal form rewrite rules to the normalization step and using the "overloaded" definitions of, e. g., dependencies, no further changes have to be made to our rewriting algorithm shown in Function "rewrite" in Section 3.5.2 to correctly handle functions. Functions used in conditions are correctly handled due to the re-definition of dependencies and hence the execution of conditional subexpressions is automatically delayed until all needed values for computing the functions have been seen in the stream. Expressions outputting function results are handled by the "catch-everything" **else**-statement in line 29 of the algorithm. Therefore, such an output expression is placed inside an appropriate "`on-first`"-handler delaying the execution until all elements needed for the computation of the function are known.

We show the effect of our rewrite algorithm with respect to handling of functions on some sample queries in the bibliography domain.

Example 3.8.7 *Consider the following query Q_4, which counts the number of authors of each book and prints it with the title of the book grouped into a new* `result` *element.*

```
<results>
{ for $book in /bib/book return
    <result>
    {$book/title}
    <numAuthors>{fn:count($book/author)}</numAuthors>
    </result> }
</results>
```

Normalization of Q_4 yields the query Q'_4 as follows:

```
1  <results>
2  { for $bin in /bib return
3    { for $book in $bib/book return
4      <result>
5      { for $title in $book/title return {$title} }
6      <numAuthors>
7      {fn:count($book/author)}
8      </numAuthors>
9      </result>
10 </results>
```

We first assume to have a stream compliant to the following DTD known from Section 3.1:

```
<!ELEMENT bib  (book)*>
<!ELEMENT book (title | author)*>
```

When rewriting Q'_4 into FluX, the `for`*-expressions in lines 2 and 3 are rewritten into* `on`*-handlers and we eventually make the recursive call "*rewrite*($book, $\perp$, α)" with α being the sequence of expressions in lines 4–9 of Q'_4. Since $H = \perp$ and dep($book,* `<result>`*) $= \emptyset$, the expression* `<result>` *in line 4 will be transformed into:*

```
on-first past() return <result>;
```

Analogously, the algorithm proceeds by rewriting the `for`*-loop into an* `on`*-handler to directly output titles. Now, when rewriting line 7, $H = \{$*`title`*$\}$ and dep($book,* `<numAuthors>`*) $=$ $\emptyset$. Thus, a handler*

```
on-first past(title) return <numAuthors>;
```

*is created. Next, "*rewrite*($book, $\{$*`title`*$\}$, $\{$*`fn:count($book/author)`*$\}$)" is called. Here, $H = \{$*`title`*$\}$ and dep($book, $\{$*`fn:count($book/author)`*$\}$) $= \{$*`author`*$\}$. Hence, line 29 of the rewriting algorithm is executed and a handler*

```
on-first past(title, author) return {fn:count($book/author)};
```

is generated. The remainder of the query is analogously transformed and eventually the following FluX query F_4 is generated:

```
{ ps $ROOT:
  on-first past() return <results>;
  on bib as $bib return
    { ps $bib:
      on book as $book return
        { ps $book:
          on-first past() return <result>;
          on title as $title return {$title};
          on-first past(title) return <numAuthors>;
          on-first past(title, author) return {fn:count($book/author)};
          on-first past(title, author) return </numAuthors>;
          on-first past(title, author) return </result>; }; };
  on-first past(bib) return </results>; }
```

In detail, the "`on-first past(title, author)`*"-handler delays the execution of the function* `{fn:count($book/author)}` *after all titles* and *authors have been seen on the stream. Thus, the query is executed in the correct order, i. e., titles are outputted before the number of authors is printed (because of delaying it after all* `title` *nodes), and the function can be correctly computed, because it is assured that all* `author` *nodes have been read from the stream. Hence, the FluX query F_4 is safe.* □

The following example shows how a query using the novel `let`-expression is handled.

Example 3.8.8 *Consider the following XQuery^{-} query Q_5, which prints titles and the number of authors of each book, if the book has more than five authors.*

```
<results>
{ for $book in /bib/book return
  { let $a := fn:count($book/author)
    where $a > 5
    return
      <result>
      {$book/title}
      <numAuthors>{$a}</numAuthors>
      </result> } }
</results>
```

This query has the following normalized form denoted as Q'_5:

```
1  <results>
2  { for $bib in $ROOT/bib return
3    { for $book in $bib/book return
4      { if (fn:count($book/author) > 5) then <result> }
5      { for $title in $book/title return
6        { if (fn:count($book/author) > 5) then {$title} } }
7      { if (fn:count($book/author) > 5) then <numAuthors> }
8      { if (fn:count($book/author) > 5) then {fn:count($book/author)} }
9      { if (fn:count($book/author) > 5) then </numAuthors> }
10     { if (fn:count($book/author) > 5) then </result> } } }
11 </results>
```

Further, we will use the DTD given in the previous example. As before, the `for`*-loops in lines 2 and 3 are rewritten into* `on`*-handlers. In order to achieve a safe query, the conditional expression in line 4 is delayed by line 29 of the rewriting algorithm until all authors have been seen, due to* $dep(\$book,$ `{fn:count($book/author)}`$) = \{$`author`$\}$. *Hence, it is rewritten into:*

```
on-first past(author) return
  { if (fn:count($book/author) > 5) then <result> };
```

The algorithm proceeds by rewriting the `for`*-loop in lines 5–6. To ensure the correct execution order as required by the query and to achieve a safe FluX query, in contrast to the previous example this* `for`*-loop is not rewritten into an* `on`*-handler, but delayed until all authors* and *titles have been seen. In detail, the following handler is created:*

```
on-first past(title, author) return
  { for $title in $book/title return
    { if (fn:count($book/author) > 5) then {$title} } };
```

Delaying this `for`*-loop until all titles (and authors, of course) have been seen is needed to assure that all titles can be buffered until we see no further* `author` *nodes (note that there is no order constraint between titles and authors). The remainder of the query is rewritten as before, and we eventually get the following FluX query* F_5*:*

```
{ ps $ROOT:
  on-first past() return <results>;
  on bib as $bib return
   { ps $bib:
     on book as $book return
      { ps $book:
        on-first past(author) return
          { if (fn:count($book/author) > 5) then <result> };
        on-first past(title, author) return
          { for $title in $book/title return
            { if (fn:count($book/author) > 5) then {$title} } };
        on-first past(title, author) return
          { if (fn:count($book/author) > 5) then <numAuthors> };
        on-first past(title, author) return
          { if (fn:count($book/author) > 5) then {fn:count($book/author)} };
        on-first past(title, author) return
          { if (fn:count($book/author) > 5) then </numAuthors> };
        on-first past(title, author) return
          { if (fn:count($book/author) > 5) then </result> }; }; };
  on-first past(bib) return </results>;}
```

Note that this FluX query is also safe. □

3.8.1.4 Implementation

In this section, we will show some core implementation details of the runtime engine concerning the evaluation of functions. In detail, we will show how function results are computed and how buffers are handled.

The key idea of handling functions in the runtime engine is to separate the computation of function results from their usage. This approach is similar to the treatment of condition results shown in Section 3.7.2. To assure that the result of a function is correctly computed before it is being used for the first time, the query compiler appropriately schedules physical query operators for evaluating and storing the result in a temporary variable in main memory. All functions contained in conditional expressions or expressions printing function results only refer to the actual result of a function being held in main memory. As a result, multiple evaluation of functions introduced by eliminating a `let`-expression is prevented and an efficient execution is achieved. Note that this approach is always correct, since the definition of safety and dependencies of a function ensure that all values needed for the computation have already been seen before the result is used for the first time.

Apart from this separation of computation and usage of function results being common to all types of functions, there is a striking difference between holistic and non-holistic functions with respect to the actual evaluation strategy and buffer management. For a non-holistic function $f_n(\$x/\pi)$, with $\pi = a_1/\ldots/a_n$ and $n \geq 1$, an automaton matching the path π is installed in the scope of $\$x$, if $\$x$ is processed as a stream[15]. Whenever a match occurs, the function result is updated. Here, the notion of a "match" depends on the actual function: E. g., for `fn:count` a match occours upon seeing the start tag of a_n and the current result is increased by one. In case of functions working on real data values, e. g., `fn:avg`, a match occurs upon seeing the "character content" event for the path π (and, of course, this content is then used for updating the result). Since the rewriting algorithm always produces safe FluX queries, as soon as f_n is used in some expression, all values needed to compute the correct result have been processed and the result of the function may safely be read from main memory. Hence, if $\$x$ is processed as a stream, non-holistic functions are transparently computed on-the-fly and no data needs to be buffered. Otherwise, non-holistic functions are computed on the already buffered data. In contrast, to be able compute a holistic function $f_h(\$x/\pi)$ all its arguments are needed. Thus, the path $\$x/\pi$ is always completely buffered. Before using f_h for the first time in some expression, the query compiler schedules a physical operator, which evaluates f_h using the buffered data. We formalize this in the following definition.

Definition 3.8.9 (Streaming Function) *A function $f(\$x/\pi)$ is a* streaming function *if f is non-holistic and $\$x$ is bound by an* `on`*-handler.*

To only buffer arguments of holistic functions, we overload the definition of streaming atomic conditions (Definition 3.7.1) as follows.

[15] That is, $\$x$ is not bound by a `for`-loop inside an "`on-first`"-handler.

```
1  * startTag:
2    on 'book':
3      PSNode[symbol=book]
4        * startStream:
5          SequenceNode
6            Output[data=<result>]
7            InitializePathMatcher[path=author; InternalAggregate: exprID=0, type=2]
8        * endStream:
9          (NOP)
10       * startTag:
11         on 'title':
12           OutputStream[state=true]
13       * endTag:
14         on 'title':
15           OutputStream[state=false]
16       * on-first:
17         OnFirstHandlerNode[symbols={title}]
18           Output[data=<numAuthors>]
19         OnFirstHandlerNode[symbols={title, author}]
20           SequenceNode
21             OutputExprResult[exprID=0]
22             Output[data=</numAuthors>]
23             Output[data=</result>]
```

Figure 3.12: Fragment of Query Plan for Example 3.8.1.4

Definition 3.8.10 (Streaming Atomic Condition) *A* streaming atomic condition *is an atomic condition of the form*

- $\$x/\pi$ *RelOp* s
- $f(\$x/\pi)$ *RelOp* s
- $f(\$x/\pi)$ *RelOp* $f'(\$y/\pi')$

with $\$x$, $\$y$ *being variables bound by an* `on`*-handler,* π, π' *(non-empty) fixed paths,* f, f' *non-holistic functions, and* s *a string value.*

As a result, optimal buffer trees only buffering arguments of holistic functions are computed exactly as described in Section 3.7.3.

We refrain from giving a more detailed description of the evaluation strategy and provide some instructive examples instead.

Example 3.8.11 *Consider again the query from Example 3.8.7, which prints for each book its title and the number of authors. The query compiler computes a query plan as shown in Figure 3.12. For the sake of brevity, only the relevant part of the query plan handling the* `book` *scope is shown.*

Since `fn:count` *is a non-holistic function, upon entering the scope of* `book` *an automaton matching the path* `author` *(relative to a book) is set up and the result of the function is initialized (line 7). The built-in function* `fn:count` *is depicted by "*`InternalAggregate`*" with "*`type=2`*". The automaton transparently receives all events of a book and updates the result of the function if an* `author` *path (relative to the current scope) is matched. As soon as no further authors and titles can be encountered, the "*`on-first`*"-handler in line 19 is executed, which prints the result of the function by means of the* `OutputExprResult` *operator in line 21 (in detail, it outputs the result of the expression with* `exprID=0`*, which is the* `fn:count` *function). Note that nothing is buffered and no additional operations for explicitly computing the function result are needed.* □

The following example illustrates the handling of holistic functions during query evaluation.

Example 3.8.12 *We modify the DTD used in the previous Example 3.8.11 to also be able to handle shops which sell a book and the price of the book in each shop:*

```
<!ELEMENT book      (title | author | bookstore)*>
<!ELEMENT bookstore (name, price)>
```

Consider the following XQuery Q_6, which prints the title of books and the median of all prices, computed by a user-defined median function:

```
<results>
{ for $book in /bib/book return
    <result>
    {$book/title}
    <medPrice>{udf:median($book/bookstore/price)}</medPrice>
    </result> }
</results>
```

Our rewriting algorithm generates the following safe FluX query F_6:

```
{ ps $ROOT:
  on-first past() return <results>;
  on bib as $bib return
    { ps $bib:
      on book as $book return
        { ps $book:
          on-first past() return <result>;
          on title as $title return {$title};
          on-first past(title) return <medPrice>;
          on-first past(title, shop) return
            {udf:median($book/bookstore/price)};
          on-first past(title, shop) return </medPrice>;
          on-first past(title, shop) return </result>; }; };
  on-first past(bib) return </results>; }
```

```
 1  * startTag:
 2    on 'book':
 3      SequenceNode
 4        BufferStream[bufID=0; state=true]
 5        PSNode[symbol=book]
 6          * startStream:
 7            SequenceNode
 8              BufferStream[bufID=0; state=false]
 9              Output[data=<result>]
10          * endStream:
11            BufferStream[bufID=0; state=true]
12          * startTag:
13            on 'title':
14              OutputStream[state=true]
15            on 'bookstore':
16              SequenceNode
17                BufferStream[bufID=0; state=true]
18                PSNode[symbol=bookstore]
19                  * startStream:
20                    BufferStream[bufID=0; state=false]
21                  * endStream:
22                    BufferStream[bufID=0; state=true]
23                  * startTag:
24                    on 'price':
25                      BufferStream[bufID=0; state=true]
26                  * endTag:
27                    on 'price':
28                      BufferStream[bufID=0; state=false]
29                  * on-first:
30          * endTag:
31            on 'title':
32              OutputStream[state=false]
33            on 'bookstore':
34              BufferStream[bufID=0; state=false]
35          * on-first:
36            OnFirstHandlerNode[symbols={title}]
37              Output[data=<medPrice>]
38            OnFirstHandlerNode[symbols={bookstore, title}]
39              SequenceNode
40                BufferedFunction[exprID=0; iterator=buf[0]/bookstore/price; ExternalAggregate: median]
41                OutputExprResult[exprID=0]
42                Output[data=</medPrice>]
43                Output[data=</result>]
44  * endTag:
45    on 'book':
46      SequenceNode
47        InitializeBuffer[bufID=0]
48        BufferStream[bufID=0; state=false]
```

Figure 3.13: Fragment of Query Plan for Example 3.8.12

The query compiler transforms F_6 as shown in Figure 3.13 into a physical query plan. Again, only the relevant part of the `book` *scope is shown.*

All `price` *nodes (and, no other nodes) are buffered, because* `udf:median` *is a holistic function. The buffer is handled by the* `InitializeBuffer` *and* `BufferStream` *operators as described in Section 3.7.3. The function itself is computed after all* `title` *and* `bookstore` *nodes have been seen (i. e., in the "*`on-first`*"-handler in lines 38–43) by the* `BufferedFunction` *operator (line 40), right before the result is needed for the first time (in*

line 41). This operator handles an iterator over all `price` *nodes contained in the buffer to the evaluation method of the external function* `udf:median`*. This method returns the result of the function, which is then stored in a temporary variable in main memory for later re-usage. This variable is read by* `OutputExprResult` *operator to print the result to the output.* □

3.8.2 Data Windows

For applications monitoring and processing (possibly) infinite data streams, e. g., sensor measurements, an important query processing task is to break the stream into (possibly overlapping) subsets of data, denoted as *data windows*, and to perform computations on each of these portions of the stream. In this section, we first present our notion of data windows. Then, we show how to integrate data windows into our XQuery$^-$ fragment, how to extend and rewrite into our FluX language, and finally how it is implemented in the runtime engine.

We start by defining the semantics of data windows. Let $(d_i)_i$ be an ordered (and possibly infinite) sequence $(d_1, d_2, \ldots, d_n)$ of data objects representing the data stream. In our case, a data object d_i represents a node of the XML Stream. A *window specification* maps $(d_i)_i$ to an ordered (and also possibly infinite) sequence $(W_j)_j$ of sets of data objects $(W_1, W_2, \ldots, W_m)$, such that

1. $W_j := (d_i, d_{i+1}, \ldots, d_{i+c_j})$ and
2. $W_{j+1} := (d_{i+s_j}, d_{i+s_j+1}, \ldots, d_{i+s_j+c_{j+1}})$,

with c_k being some constant and $s_k \geq 1$ $(1 \leq k \leq m)$. In other words, the data windows W_j slide over the data stream in a forward-only fashion possibly overlapping each other and always contain a contiguous interval of the data stream.

In this work, we focus on two types of window specifications: *element-based* and *time-based* data windows, which are defined as follows.

Definition 3.8.13 (Element-Based Data Windows) *Let $\Delta, \mu \in \mathbb{N}^+$. The sequence of data windows is defined as:*

$$\begin{aligned} W_1 &:= (d_1, d_2, \ldots, d_\Delta) \\ &\vdots \\ W_j &:= (d_{(j-1)\mu+1}, d_{(j-1)\mu+2}, \ldots, d_{(j-1)\mu+\Delta}) \\ &\vdots \end{aligned}$$

That is, each element-based data window W_j contains exactly Δ data objects (except for the last data window, which may contain fewer elements—if the stream is finite). Further, the data window “slides” forward by discarding the first μ elements and adding the next μ data objects to it.

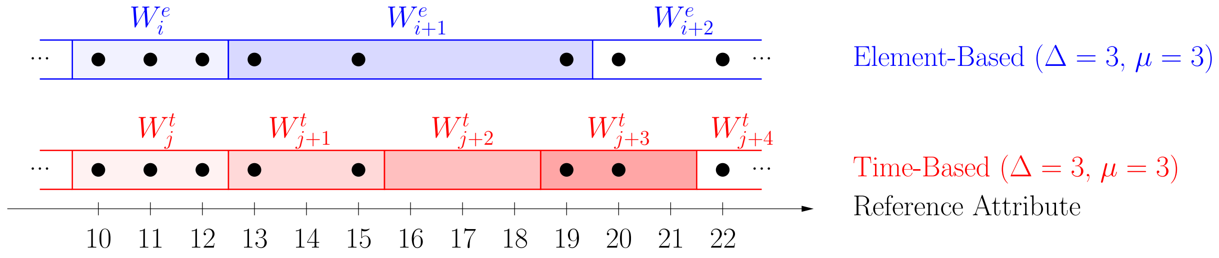


Figure 3.14: Element-Based and Time-Based Data Windows

Definition 3.8.14 (Time-Based Data Windows) *Let $\Delta, \mu \in \mathbb{R}^+$ and t be an attribute of each data object d_i, denoted as the* reference attribute*. We refer to the reference attribute of a data object d_i by "$d_i.t$". Further, let the sequence S of data objects be ordered according to the reference attribute, i. e., $d_i.t \leq d_{i+1}.t$. The sequence of data windows is defined as:*

1. *For each data window $W_j := (d_i, d_{i+1}, \ldots, d_{i+c_j})$*

$$(d_{i+c_j}.t - d_i.t < \Delta) \wedge (d_{i+c_j+1}.t - d_i.t \geq \Delta)$$

   holds.

2. *The sequence of data windows $(W_1, \ldots, W_m)$ is defined as*

$$\begin{aligned} W_1 &:= (d_1, d_2, \ldots, d_{c_1}) \\ &\vdots \\ W_j &:= (d_i, d_{i+1}, \ldots, d_{i+c_j}) \quad \textit{with} \quad d_i.t = (j-1)\mu + d_1.t \\ &\vdots \end{aligned}$$

Here, c_k $(1 \leq k \leq m)$ is a constant representing the size of each data window.

In other words, a time-based data window contains all elements differing less than Δ in the reference attribute and slides forward by the amount of μ. Note that the number of data objects contained in the individual data windows is not fixed and completely depends on the values of the reference attribute. Usually, the reference attribute is some kind of a time-stamp (and thus this type is called a "time-based" data window). In that case, the sequence of data objects is naturally ordered according to the reference attribute.

Figure 3.14 illustrates element-based and time-based data windows for $\Delta = 3$ and $\mu = 3$. In this example, we use an attribute in the domain of integer values as reference attribute, e. g., constituting a time-stamp. Data objects are symbolized by dots and delivered from left to right. Note that the data objects are not dense with respect to the reference attribute in the sense that not for every possible value of the reference attribute a data object exists. All element-based data windows W_i^e exactly contain three data objects. In contrast, all

RelPath	::=	**QName** ("/" **QName**)*
FixedPath	::=	**VarRef** ("/" *RelPath*)?
WinPath	::=	*FixedPath* "|" *WinSpec* "|"
WinSpec	::=	("`count`" **IntegerLiteral** ("`step`" **IntegerLiteral**)?
		| (*RelPath*)? "`diff`" **NumericLiteral** ("`step`" **NumericLiteral**)?)
Expression	::=	("`()`"
		| **Literal**
		| *Expression* (*Expression*)+
		| "`{`" "`for`" **VarRef** "`in`" *FixedPath* ("`where`" *AtomicCondition*)? "`return`" *Expression* "`}`"
		| "`{`" "`let`" **VarRef** "`:=`" *Function* ("`where`" *AtomicCondition*)? "`return`" *Expression* "`}`"
		| "`{`" "`for`" **VarRef** "`in`" *WinPath* ("`where`" *AtomicCondition*)? "`return`" *Expression* "`}`"
		| "`{`" *FixedPath* "`}`"
		| "`{`" **VarRef** "`}`"
		| "`{`" *Function* "`}`"
		| "`{`" "`if`" "`(`" *AtomicCondition* "`)`" "`then`" *Expression* "`}`")

Figure 3.15: XQuery$^-$ Grammar Fragment for Handling Data Windows

time-based data windows W_j^t have the same size with respect to the reference attribute, but contain a different number of data objects or are even empty. Note that if the data objects would be dense with respect to the reference attribute, i. e., a data object exists for every possible value of the reference attribute, then $W_i^e = W_i^t$.

This semantics of data windows and window specifications is equal to that of other works, e. g., of [LMT+05] or [ABW03]. Of course, not all instances of data windows are computed at once on the whole input data and then a computation is performed on each data window. The goal is to use the data window as a moving view of the stream and to perform processing only in the current data window with a minimum amount of the data stream being buffered. When computing data windows on a data stream in such a sliding fashion, we say that a window specification is fulfilled, if the current data window is completely filled by adding the current data object and hence the current data window is ready to be processed.

3.8.2.1 Extending the XQuery$^-$ Fragment

We will now present an extension of XQuery$^-$ to support data windows. Since standard XQuery has been designed for materialized XML data, it currently does not yet provide any syntactical means for specifying data windows. Hence, we also propose the following extensions of XQuery$^-$ for standard XQuery. To provide for a seamless integration, we thus adhere to the standard XQuery semantics as closely as possible.

Figure 3.15 shows the needed extensions to the XQuery$^-$ grammar for supporting data

windows. Again, only new or changed productions are shown.

To enable windowed computations *windowed paths* are introduced. We define the semantics of windowed paths as follows. In standard XQuery, the result of a path expression "$\$x/a_1/\dots/a_n$" is a sequence of a_n nodes, which are reachable by the given path in the current context. In case of a windowed path "$\$x/a_1/\dots/a_n|winSpec|$", the path "$\$x/a_1/\dots/a_n$" denotes the sequence of a_n nodes, which should be processed by sliding data windows. This sequence of a_n nodes is mapped to a sequence of sets of a_n nodes by means of the window specification "*winSpec*", which is enclosed by "|". Hence, the result of a windowed path expression is a sequence of data windows according to the window specification. Element-based windows are specified by "`|count` Δ `step` μ`|`" under the semantics given in Definition 3.8.13. "`|`$a_{n+1}/\dots/a_{n+i}$ `diff` Δ `count` μ`|`" specifies time-based data windows as defined in Definition 3.8.14 with "$a_{n+1}/\dots/a_{n+i}$" specifying the reference attribute relative to the base path "$\$x/a_1/\dots/a_n$". If the path to the reference attribute is omitted, the content of each window element is used. Note that the path to the reference attribute is never directly evaluated, but only used as some kind of "specification" of (or "reference" to) the reference attribute. In both window specifications the "`step` μ" part may be omitted. In that case, we assume $\mu := \Delta$.

Data windows are processed by means of the standard XQuery `for`-expression. Because of the semantics of windowed path expressions, the usual semantics of the XQuery `for`-expression exactly achieves the desired handling of data windows. In detail, consider the expression:

`for $w in $x/`$a_1/\dots/a_n$`|`$winSpec$`| return` α

The windowed path $\$x/a_1/\dots/a_n|winSpec|$ evaluates to a sequence of data windows. The data windows are sequentially bound to the variable $\$w$ by means of the `for`-expression and the subquery α is executed for each data window. Inside α, the contents of the current data window are accessible by means of $\$w$. Although the grammar supports an arbitrary number of (nested) data windows, we focus in the remainder of this thesis on queries having a single data window, e. g., for computing windowed aggregate functions. Supporting multiple data windows, e. g., for windowed joins, is subject of current research.

The following example illustrates the usage of data windows.

Example 3.8.15 *Consider the DTD*

```
<!ELEMENT root (a*)>
<!ELEMENT a    (b*, c*)>
<!ELEMENT c    (d*)>
```

and the following query Q_7*, which prints for each* `a` *node its* `b` *children and an element-based*

sliding window average of the values of all `d` *nodes under this* `a` *node:*

```
<results>
  { for $x in /root/a return
    <result>
      {$x/b}
      {for $w in $x/c/d|count 4 step 2| return
         <winavg>
           {fn:avg($w)}
         </winavg>}
    </result> }
</results>
```

We evaluate this query on the following XML document:

```
<root>
  <a>
    <b>some b data</b>
    <b>other b data</b>
    <c>
      <d>1</d> <d>2</d> <d>3</d> <d>4</d>
    </c>
    <c>
      <d>5</d> <d>6</d> <d>7</d>
    </c>
  </a>
</root>
```

The windowed path `$x/c/d|count 4 step 2|` *evaluates to:*

```
(  (<d>1</d>, <d>2</d>, <d>3</d>, <d>4</d>),
   (<d>3</d>, <d>4</d>, <d>5</d>, <d>6</d>),
   (<d>5</d>, <d>6</d>, <d>7</d>)              )
```

The result of the query is:

```
<results>
  <result>
    <b>some b data</b>
    <b>other b data</b>
    <winavg>2.5</winavg>
    <winavg>4.5</winavg>
    <winavg>6</winavg>
  </result>
</results>
```

Note that processing data windows can be expressed in native XQuery by implementing a function computing the sequence of data windows according to the window specification. The former query can be phrased in native XQuery as follows:

```
1  declare function cwin($count as xs:integer, $step as xs:integer,
2                        $e as node()*)
3    as node()*
4  {
5    let $win := fn:subsequence($e, 1, $count)
6    let $rest := fn:subsequence($e, $step + 1)
7    return
8      if (fn:count($e) <= $count) then
9        (<wc>{$win}</wc>)
10     else
11       (<wc>{$win}</wc>, cwin($count, $step, $rest))
12 };
13
14 for $x in /root/a return
15   <result>
16     {$x/b}
17     {for $tmp in cwin(4, 2, $x/c/d)
18      let $w := $tmp/*
19      return
20        <winavg>{fn:avg($e)}</win>}
21   </result>
```

Observe that the function `cwin` *(lines 1–12) implements element-based windows in a functional style. The query itself in lines 14–21 is basically equal to the original query using our notation for data windows. The main difference is the* `let`*-expression in line 18. It is needed to remove the artificial* `wc`*-tags, which are introduced by the function* `cwin` *to separate the individual data windows in the sequence of data windows.* □

3.8.2.2 Syntax and Semantics of FluX Revisited

To provide a better understanding of how windowing techniques are integrated into the event-based part of FluX, we start with some basic thoughts on processing data windows on data streams.

As sketched out before, data windows are used to restrict computations on a small portion of the (possibly infinite) data stream, which "slides" along the stream over time. Given a window specification, we can observe the following: If $\Delta > \mu$, the data stream is divided into a sequence of overlapping data windows $(W_i)_i$, i. e., $W_i \cap W_{i+1} \neq \emptyset$. Since data in a data stream is volatile, i. e., it can be processed only once if it is not buffered somewhere, at least the overlapping part of two subsequent data windows has to be buffered for being able to process it in each data window it is contained. In contrast, if $\Delta \leq \mu$, the data stream is divided into a disjoint sequence of data windows $(W_i)_i$. Here, no data object has to be processed in more than one data window and hence potentially nothing has to be buffered for being able to correctly process each data window. Of course, this only holds, if the query itself, which is executed on each data window, can be processed without any buffering.

Further, the window specification itself has a crucial impact on processing data windows on data streams. In case of an element-based data window, upon seeing an element of the data window it can be immediately determined, if the window specification is fulfilled or not. Hence, this element can be correctly processed in a streaming fashion with respect to the current state of the data window. Whether the window specification of a time-based data window is fulfilled or not, cannot be determined based on the current element. In detail, we are only able determine that the current data window is completely filled by including the current element, if the *next* element is outside of the current data window. Hence, the current element has to be buffered to be able to process the data window correctly as soon as it is known, that the window specification is fulfilled upon seeing some future elements.

We formalize these observations in the following definition.

Definition 3.8.16 (Simple Window Specification) *Let ws be a window specification with parameters Δ and μ as introduced at the beginning of this section. We call ws* simple, *iff ws is element-based and $\Delta \leq \mu$.*

We extend the syntax of FluX to incorporate these two notions of processing data windows as follows.

Definition 3.8.17 (FluX) *Let the class of FluX expressions be as defined in Definition 3.4.12. Further, let ws be a window specification and $\$y/a/\pi|ws|$ a windowed path. We additionaly introduce the following two types of event handlers:*

- *(so-called "*`on-window`*"-handler)*

  `on` $\$y/a/\pi|ws|$ `as` $\$x$ `return` Q

  where $\$x$ is a variable, ws is simple, and Q is a simple expression.

- *(so-called "*`on-each past`*"-handler)*

  `on-each past` $\$y/a/\pi|ws|$ `as` $\$x$ `return` α

  where $\$x$ is a variable and α is a XQuery$^-$ expression.

The semantics of the new "`on-window`"- and "`on-each past`"-handler is defined as follows. The event condition of such a handler is fulfilled each time the first node is seen on the stream, which fulfills the given window specification. That is, if such a handler fires, we have seen all nodes of the stream, such that the content of the current data window is completely known and can be processed. Of course, these handlers fire multiple times while processing an input stream, i. e., once for each data window. Given a windowed path $\$y/a/\pi|ws|$ a window handler eventually fires, if and only if we have seen all occurrences of a nodes and no new nodes of the path $\$y/a/\pi$ have been encountered since the last invocation of the window handler. That is, the final data window is always processed, even

if the window specification is not yet fulfilled for the last data window (and never will be fulfilled, since no a nodes will be seen on the stream anymore). Whenever such a handler fires, the associated subquery is executed using the standard XQuery semantics, i. e., it is assumed that the whole content of the data window is bound to the variable given after the `as` keyword. Informally, the difference between an "`on-window` ... `as` $\$x$ ..."- and an "`on-each past` ... `as` $\$x$ ..."-handler is, that the "`on-window`"-handler does not bind the whole content of the data window to $\$x$, but again processes $\$x$ as a stream—similar to an `on`-handler. In contrast, an "`on-each past`"-handler actually binds the whole data window to $\$x$ by buffering the data and processing the subquery on the buffered data—similar to an "`on-first`"-handler. We will show details on how buffer management and query execution is done for each handler in Section 3.8.2.4.

Note that we restrict subqueries of "`on-window`"-handlers to simple expressions. Basically, arbitrary streaming FluX expressions, i. e., a `ps`-expression, might be possible. But, the semantics of `ps`-expressions is not sufficient for handling data windows, because it is designed for processing a single node, whereas in a data window a sequence of nodes has to be processed. This could be addressed by introducing some kind of a "process window"-statement, which individually processes each node contained in a data window by means of a `ps`-expression. However, the optimization potential is very limited in that case, since no order constraints can be derived to reduce buffering. We illustrate this fact by means of the following example. We use the DTD given in Example 3.8.15 and extended it by the production

```
<!ELEMENT d (e, f)>.
```

Now, consider the following XQuery$^-$ expression, which prints for each window all `e` nodes of *all* `d` nodes *before* all `f` nodes of, again, *all* `d` nodes.

```
{ for $w in $x/c/d|ws| return
  { for $e in $w/e return {$e} }
  { for $f in $w/f return {$f} } }
```

The content of an arbitrary data window is then $((e_1, f_1), \ldots, (e_n, f_n))$, where (e_i, f_i) denotes some node d_i. The FluX idea of scheduling parts of the query with respect to order constraints to process this data window in a streaming fashion would be applied as follows: We would scan through the data window, directly output all nodes e_i (as it is done by an `on`-handler), buffer all nodes f_i, and eventually output all buffered f_i nodes at the end of the data window (similar to an "`on-first`"-handler). Since we subsequently walk through the d_i nodes (the parent node of each e_i and f_i), we cannot derive an order constraint between `e` and `f` nodes, such that we might be able to directly output both `e` and `f` nodes on-the-fly assuring that *all* e_i are printed *before all* f_i. Generally speaking, only the first expression of a subquery being executed on a data window can potentially be evaluated directly on the stream without buffering. Data needed for the remaining expressions always has to be buffered for being processed at the end of the data window. Of course, by directly executing the first expression on the stream without buffering (if the expression allows for that) buffer consumption is reduced. However, since both data windows itself

and the contained data items are generally not very large, there is no huge benefit and therefore we did not pursue this approach further.

The definition of free variables of a FluX expression (Definition 3.4.13/3.8.4) can be extended to include "`on-window ... as` $\$x$ ..."- and "`on-each past ... as` $\$x$ ..."-handlers, such that they bind the variable $\$x$ and remove it from the set of free variables of the superexpressions—similar to an `on`-handler. We will not show the extended definition for the sake of brevity.

To be able to define safety for FluX queries involving data windows, we have to slightly adapt the definition of dependencies (Definition 3.4.17/3.8.5).

Definition 3.8.18 (Dependencies) *Let the set of dependencies with respect to a variable* $\$y$ *of an XQuery*$^-$ *expression be defined as in Definition 3.8.5. Further, let*

$$\begin{aligned} dep(\$y;\ \texttt{\{for}\ \$x\ \texttt{in}\ \$y/\pi|ws|\ \texttt{return}\ \alpha\texttt{\}}) &:= dep(\$y,\ \$y/\pi|ws|) \\ &\quad \cup\ dep(\$y;\ \alpha) \\ dep(\$y;\ \texttt{\{for}\ \$x\ \texttt{in}\ \$y/\pi|ws|\ \texttt{where}\ \chi\ \texttt{return}\ \alpha\texttt{\}}) &:= dep(\$y,\ \$y/\pi|ws|) \\ &\quad \cup\ dep(\$y;\ \alpha) \cup dep(\$y;\ \chi) \\ dep(\$y;\ \$y/\pi|ws|) &:= dep(\$y;\ \$y/\pi) \end{aligned}$$

with $|ws|$ *denoting a window specification,* π *a path, and* α *an XQuery*$^-$ *expression.*

Note that the reference path of a time-based window does not contribute to the dependencies, since it only refers to the window element itself or a deeper nested element. With this, we are able to extend the notion of safe queries given in Definition 3.4.18/3.8.6 to include data window computations.

Definition 3.8.19 (Safe Queries) *A FluX query Q is called* safe *with respect to a given DTD if and only if for each subexpression "*`{ps` $\$y$`:` ζ `}`*" of Q, the two conditions given in Definition 3.8.6 and the following condition are satisfied:*

3. *For each handler "*`on-each past` $\$y/a/\pi|ws|$ `as` $\$x$ `return` α*" in the list* ζ*, the following is true:*
   - $\forall\ \$z \in free(\alpha)$ *such that "*$\{\$z\}$*"* $\preceq \alpha$ *we have:* $\$z = \$x$
   - $\forall\ \$z \in free(\alpha)$ *such that "*$\{\$z/b/\pi'\}$*"* $\preceq \alpha$*, "*$\{f(\$z/b/\pi')\}$*"* $\preceq \alpha$*, or "*$f(\$z/b/\pi')$*" contained in some condition* χ *(for some function* f *and path* π'*) we have:*

$$(\$z = \$x)\ or\ ((\$z = \$y) \wedge (Ord_{\$y}(b, a))$$

In other words, (3) ensures that a subquery of an "`on-each past`"-handler references only contents of the current data window or data which is completely known before processing the sequence of data windows. Hence, data which is needed inside data windows but stems from outside of the data windows can be completely buffered before processing the data windows. Note that "`on-window`"-handlers are per definition safe, since they may only contain simple expressions as subqueries, which can be processed directly on the stream without buffering at all.

3.8.2.3 Rewriting Extended XQuery$^-$ into FluX

We will now address the problem of rewriting a query containing the novel parts for data window processing into an equivalent FluX query that employs as little buffering as possible.

Again, the query is first transformed into its normal form by applying the rules depicted in Figure 3.3 and 3.11. We introduce the restriction, that the rule *[Single-Step-For]* must not be applied to `for`-expressions defining data windows, i. e., `for`-expressions containing windowed paths. That is, paths referring to the elements of data windows are not broken into single-step `for`-loops. The remaining rewriting rules concerning `for`-expressions, i. e., *[Elim-Where]* and *[Pushdown-If-1]*, do not distinguish between windowed and non-windowed paths and are applied as usual. The following example illustrates the normalization process.

Example 3.8.20 *Consider the query Q_7 and the DTD as given in Example 3.8.15. Q_7 has the following normalization, denoted as Q_7':*

```
1  <results>
2  { for $r in $ROOT/root return
3    { for $x in $r/a return
4        <result>
5        { for $b in $x/b return {$b} }
6        { for $w in $x/c/d|count 4 step 2| return
7          <winavg>
8            {fn:avg($w)}
9          </winavg> }
10       </result> } }
11 </results>
```

Note that the windowed path in the `for`*-expression in line 6 is not broken into single-step* `for`*-loops.* □

To formulate the extensions of our rewrite algorithm, we need to adapt the definition of handler symbols to include the novel handlers as follows.

Definition 3.8.21 (HSymb) *Let the set $hsymb(\zeta)$ of handler symbols be defined as in Definition 3.5.2. Additionally, we inductively define $hsymb(\zeta)$ for the novel "*`on-window`*"- and "*`on-each past`*"-handler as follows:*

$$hsymb(\zeta;\ \texttt{on-window}\ \$x/a/\pi\ \texttt{as}\ \$y\ \texttt{return}\ \alpha) := hsymb(\zeta) \cup \{a\}$$
$$hsymb(\zeta;\ \texttt{on-each past}\ \$x/a/\pi\ \texttt{as}\ \$y\ \texttt{return}\ \alpha) := hsymb(\zeta) \cup \{a\}$$

With this, we extend our rewrite algorithm presented in Section 3.5.2 to handle the novel constructs for data window processing as depicted in Figure 3.16. For the sake of brevity, we only show the new fragment of the rewriting function handling data windows. This fragment has to be inserted between lines 27 and 28 of the Function "rewrite". The complete rewriting algorithm is shown in Appendix B.

Figure 3.16: Extension of Function "rewrite" for Handling Data Windows

101 **else if** *β is of the form* `{for` $\$y$ `in` $\$z/a/\pi|ws|$ `return` α`}` **then**
102 $X := \{b \in dep(\$x, \alpha) \cup H \mid \neg Ord_{\$x}(b, a)\}$;
103 **if** $(\$z \neq \$x) \vee (X \neq \emptyset)$ **then**
104 **return** `{ps` $\$x$`: on-first past(`$X \cup \{a\}$`) return` β`}`;
105 **else**
106 **if** *β is simple and $|ws|$ is simple* **then**
107 **return** `{ps` $\$x$`: on-window` $\$z/a/\pi|ws|$ `as` $\$y$ `return` β`}` ;
108 **else**
109 **return** `{ps` $\$x$`: on-each past` $\$z/a/\pi|ws|$ `as` $\$y$ `return` β`}` ;
110 **endif**
111 **endif**
112 **endif**

Instead of discussing the novel part of the rewriting algorithm in detail, we provide some illustrative examples in the following.

Example 3.8.22 *Consider again query Q_7 and the corresponding DTD shown in Example 3.8.15. We slightly modify the window specification of this query and obtain the following normalized query Q'_8:*

```
1  <results>
2  { for $r in $ROOT/root return
3    { for $x in $r/a return
4        <result>
5        { for $b in $x/b return {$b} }
6        { for $w in $x/c/d|count 4 step 4| return
7          <winavg>
8            {fn:avg($w)}
9          </winavg> }
10       </result> } }
11 </results>
```

Rewriting lines 1–5 of Q'_8 using our rewrite algorithm is done similarly as described in Section 3.5.3 and eventually yields the following fragment of a FluX query:

```
{ ps $ROOT:
  on-first past() return <results>;
  on root as $r return
    { ps $r:
      on a as $x return
        { ps $x:
          on-first past() return <result>;
          on b as $b return {$b};
```

Note that the `for`*-loop of line 5 is rewritten into an* `on`*-handler directly outputting* `b` *nodes, because at this point,* $H = \emptyset$ *and* $X = \emptyset$*. Next, the* `for`*-expression defining the data window in lines 6–9, which will be denoted as* β *in the following, is rewritten by means of the recursive call "rewrite(*$\$x$*, {*`b`*},* β*)" performed in lines 13/15 of the rewrite algorithm. Since* β *specifies a data window, the novel part of the algorithm shown in Figure 3.16 is executed. Here,* $\$x = \$z$ *and* $X = \emptyset$*, because the given DTD assures the order constraint* $Ord_{\mathtt{a}}(\mathtt{b}, \mathtt{c})$*. Further, the subquery of the* `for`*-loop is simple and also the window specification is simple* ($\Delta = \mu = 4$)*. Hence, line 107 is executed, which yields the following FluX expression:*

```
{ ps $x:
  on-window $x/c/d|count 4 step 4| as $w return
    <winavg>
    {fn:avg($w)}
    </winavg>; }
```

Because of the definition of handler symbols, rewriting the remaining expressions proceeds with $H = \{\mathtt{b}, \mathtt{c}\}$*. Eventually, the following FluX query* F_8 *is produced:*

```
{ ps $ROOT:
  on-first past() return <results>;
  on root as $r return
    { ps $r:
      on a as $x return
        { ps $x:
          on-first past() return <result>;
          on b as $b return {$b};
          on-window $x/c/d|count 4 step 4| as $w return
            <winavg>
            {fn:avg($w)}
            </winavg>;
          on-first past(c, b) return  </result>; }; };
  on-first past(root) return </results>; }
```

Note that the FluX query F_8 *is safe, because (1) the DTD assures, that all* `b` *nodes have been printed before the first* `c` *node is seen on the stream and (2) the data window computation itself can be executed directly on the data stream, since the window specification and the subquery itself are simple. Hence, no buffers are needed at all for evaluating this query. We will show details on the actual execution strategy of such computations on data windows in the next section.* □

We demonstrate by means of the following example how more complex computations on data windows, i. e., non-simple queries, are optimized.

Example 3.8.23 *Consider query* Q_7 *and the DTD given in Example 3.8.15 with its normalization* Q'_7 *presented in Example 3.8.20.*

Rewriting lines 1–5 into FluX produces the same result as shown in the previous example. As before, the `for`*-expression defining the data window in lines 6–9, which will be*

*denoted as β in the following, is rewritten by means of the recursive call "*rewrite($x, {b}, β)*" performed in lines 13/15 of the rewrite algorithm. Again, $\$x = \$z$ and $X = \emptyset$, because the given DTD assures the order constraint* $Ord_{\mathsf{a}}(\mathsf{b}, \mathsf{c})$*. Now, the window specification is not simple ($\Delta = 4 \neq \mu = 2$), because two subsequent data windows have an overlap of two nodes. Hence, line 109 is executed returning the following expression:*

```
{ ps $x:
  on-each past $x/c/d|count 4 step 2| as $w return
    <winavg>
    {fn:avg($w)}
    </winavg>; }
```

Rewriting the remaining part of Q_7' eventually produces the following FluX query F_7:

```
{ ps $ROOT:
  on-first past() return <results>;
  on root as $r return
    { ps $r:
      on a as $x return
        { ps $x:
          on-first past() return <result>;
          on b as $b return {$b};
          on-each past $x/c/d|count 4 step 2| as $w return
            <winavg>
            {fn:avg($w)}
            </winavg>;
          on-first past(c, b) return </result>; }; };
  on-first past(root) return </results>; }
```

Again, because of the order constraint between `b` *and* `c` *nodes all* `b` *nodes can be directly output before all* `c` *nodes are processed in a windowed fashion. The "*`on-each past`*"-handler buffers the whole data window, i. e., four* `d` *nodes, and triggers the execution of the subquery as soon as the window specification is fulfilled (i. e., if four* `d` *nodes are contained in the buffer). Hence, all nodes needed for correctly computing the subquery on the data window are fully buffered and F_7 is safe. More details on the execution of such a handler, e. g., how data windows are moved, will be provided in the next section.* □

Eventually, the most general case of processing data windows is shown in the following example.

Example 3.8.24 *Consider again query Q_7 of Example 3.8.15 and its normalized form Q_7' shown in Example 3.8.20. Now, we assume the following DTD:*

```
<!ELEMENT root (a*)>
<!ELEMENT a    (b | c)*>
<!ELEMENT c    (d*)>
```

That is, no order constraint between `b` *and* `c` *nodes can be derived from the DTD.*

Rewriting lines 1–5 into FluX produces the same result as shown in the previous examples. As before, the `for`*-expression defining the data window in lines 6–9, which will be denoted as* β *in the following, is rewritten by means of the recursive call "rewrite(*$\$x$*, {*`b`*},* β*)" performed in lines 13/15 of the rewrite algorithm. Now,* $\$x = \$z$ *and* $X = \{\mathtt{b}\}$*, because the order constraint* $Ord_{\mathsf{a}}(\mathtt{b}, \mathtt{c})$ *does not hold for this DTD. Hence, line 104 of the extended rewrite algorithm is executed returning the following expression:*

```
{ ps $x:
  on-first past(b, c) return
    { for $w in $x/c/d|count 4 step 2| return
        <winavg>
        {fn:avg($w)}
        </winavg> }; }
```

Again, the remaining part of Q'_7 *is rewritten as described in the previous examples and eventually the following FluX query* F'_7 *is produced:*

```
{ ps $ROOT:
  on-first past() return <results>;
  on root as $r return
    { ps $r:
      on a as $x return
        { ps $x:
          on-first past() return <result>;
          on b as $b return {$b};
          on-first past(b, c) return
            { for $w in $x/c/d|count 4 step 2| return
              <winavg>
              {fn:avg($w)}
              </winavg> };
          on-first past(b, c) return </result>; }; };
  on-first past(root) return </results>; }
```

In this case, all `b` *nodes are directly output without buffering. Due to the "*`on-first`*"-handler,* all `c` *nodes are buffered. After all* `b` *and* `c` *nodes have been seen, the data windows are computed and processed based on the buffered data. Obviously,* F'_7 *is also safe.* □

Of course, it could be argued that an evaluation strategy as shown in Example 3.8.24 is undesireable, since buffering *all* nodes to be processed in a windowed fashion is somewhat contradictory to the goal of limiting buffer size by applying windowing techniques. However, results must adhere to the structure given by an XQuery, e. g., in the previous example *all* `b` nodes must be outputted *before* processing the `c` nodes in a windowed fashion. If the DTD does not impose sufficient order constraints, the evaluation strategy of buffering all data and computing the data windows afterwards is the best that can be done in order to produce results compliant to the query.

3.8.2.4 Implementation

In this section, we show how data windows are handled by the query compiler and the runtime engine. For the sake of brevity, we refrain from giving an abstract description and show how the former three examples are compiled into a query plan instead.

To be able to correctly process conditions and functions, we revise the definitions of "streaming atomic conditions" (Definition 3.7.1/3.8.10) and "streaming functions" (Definition 3.8.9) to incorporate the novel handlers as follows.

Definition 3.8.25 (Streaming Atomic Condition) *Let a* streaming atomic condition *be exactly as defined in Definition 3.8.10, with $\$x$, $\$y$ being variables bound by an* `on`-*handler* or *an "*`on-window`*"-handler.*

Definition 3.8.26 (Streaming Function) *A function $f(\$x/\pi)$ is denoted as* streaming function*, if and only if f is non-holistic and $\$x$ is bound by an* `on`-*handler* or *an "*`on-window`*"-handler.*

The first two examples show the execution of an "`on-window`"-handler, which can be directly processed on the stream without buffering any data. The basic idea of executing such a handler is as follows: Recall that a subquery $\overline{\alpha}$ of such a handler has to be simple. According to the definition of simple queries, $\overline{\alpha}$ can be rewritten as a sequence $\alpha\beta\gamma$, where α and γ may (conditionally) output some constants (or, function results, which are actually treated as constants, since the scheduling of expressions assures that the result has been completely computed before) and β processes the current input stream. Applied on data windows, α is executed at the beginning of a new data window, β is processed on-the-fly as the data streams into the query engine, and γ is executed at the end of the current data window.

Example 3.8.27 *Consider FluX query F_8 of Example 3.8.22. The interesting part of F_8 dealing with data windows is the scope of* `a`*, which has been rewritten into the following FluX query fragment:*

```
1  on a as $x return
2    { ps $x:
3      on-first past() return <result>;
4      on b as $b return {$b};
5      on-window $x/c/d|count 4 step 4| as $w return
6        <winavg>
7        {fn:avg($w)}
8        </winavg>;
9      on-first past(c, b) return  </result>; }; };
```

This part of the query is compiled into the query plan depicted in Figure 3.17. Printing the tag `<result>` *(line 3), the contents of all* `b` *nodes (line 4), and the tag* `</result>` *(line 9) is done as already shown in Section 3.7.2 in lines 4, 30–31, 33–34, and 41–42 of the query*

```
1  PSNode[symbol=a]
2    * startStream:
3      SequenceNode
4        Output[data=<result>]
5        InitializePathMatcher[path=c/d; CountWinMatcher: size=4; step=4]
6        InitializePathMatcher[path=c/d; InternalAggregate: exprID=0, type=0]
7    * endStream:
8      (NOP)
9    * startTag:
10     on 'c':
11       PSNode[symbol=c]
12         * startStream:
13           (NOP)
14         * endStream:
15           (NOP)
16         * startTag:
17           on 'd':
18             StreamWindow[win=c/d|count 4 step 4|, type=startWindow]
19               SequenceNode
20                 InitializeExprResult[exprID=0]
21                 Output[data=<winavg>]
22         * endTag:
23           on 'd':
24             StreamWindow[win=c/d|count 4 step 4|, type=endWindow]
25               SequenceNode
26                 OutputExprResult[exprID=0]
27                 Output[data=</winavg>]
28         * on-first:
29           (NOP)
30     on 'b':
31       OutputStream[state=true]
32   * endTag:
33     on 'b':
34       OutputStream[state=false]
35   * on-first:
36     OnFirstHandlerNode[symbols={c}]
37       StreamWindow[win=c/d|count 4 step 4|, type=finalWindow]
38         SequenceNode
39           OutputExprResult[exprID=0]
40           Output[data=</winavg>]
41     OnFirstHandlerNode[symbols={c, b}]
42       Output[data=</result>]
```

Figure 3.17: Fragment of Query Plan for Example 3.8.22

plan. The result of `fn:avg` *is computed directly on the stream as shown in Section 3.8.1 by means of line 6 of the query plan.*

To determine the current state of the data window, the window specification, i. e., `|count 4 step 4|`*, has to be evaluated on the data stream. This is done by setting up*

an automaton, which receives all events of this scope and matches the path of the window specification (line 5). Here, we have an element-based window and thus the path is equal to the base-path of the window elements, i. e., `c/d` *(relative to $\$x$). Whenever a match occurs, the window specification (here,* `CountWinMatcher`*) is updated, i. e., the current size of the window is increased by one.*

The "`on-window`*"-handler is transformed into physical query operators as follows. First, the path* `$x/c/d` *defining the elements of the data window is decomposed into one-step paths and corresponding nested scope-handlers are generated. Here, a scope-handler for* `c` *is introduced (lines 11–29). As sketched before, the subquery of the "*`on-window`*"-handler (lines 6–8 of the FluX fragment) is simple and can be decomposed into $\alpha =$* `<winavg>`*, $\beta = \epsilon$, and $\gamma =$* `{fn:avg($w)} </winavg>`*. Appropriate execution of α and γ is achieved by the physical operator* `StreamWindow`*, which conditionally executes a subplan depending on the state of the current data window. The execution of α at the beginning of the current data window is performed in lines 18–21 of the query plan. Whenever a start-tag of a* `d` *element (which are the elements contained in the data window) is encountered, the* `StreamWindow` *operator executes lines 19–21 if and only if the window specification indicates that upon seeing this* `<d>` *tag the current data window starts, which is denoted by "*`type=startWindow`*". Additionally, at the beginning of each data window the query compiler inserts operators for resetting all expressions, e. g., conditions or function results, depending on the window variable. Here, in line 20 the result of* `fn:avg($w)` *is initialized to ensure that only elements of the current data window are considered. Whenever an end-tag* `</d>` *is encountered (lines 23–27), the* `StreamWindow` *operator of type "*`endWindow`*" executes γ (lines 25–27), if and only if the window specification is fulfilled, i. e., the current data window is at its end. Having executed γ, the* `StreamWindow` *operator moves the data window by appropriately updating the window specification, i. e., setting the current size to zero. The final window is processed in an "*`on-first past(c)`*"-handler (lines 36-40) by a* `StreamWindow` *operator of type "*`finalWindow`*", which executes γ (lines 38–40), if and only if any new* `d` *elements have been seen since the beginning of the current data window.*

Note that no special handler for processing an "`on-window`*"-handler is needed in the query plan, but the appropriate operators are scheduled into conventional start-/end-tag and ofp-handlers.* □

Example 3.8.28 *Consider again the FluX fragment of the previous example. We replace the subquery of the "*`on-window`*"-handler in lines 6–8 with*

```
<win>
{$w}
</win>
```

to print the contents of the entire data window nested inside a `win` *element instead of computing a windowed average. Observe that this sequence of expressions is also simple with $\alpha =$* `<win>`*, $\beta =$* `{$x}`*, and $\gamma =$* `</win>`*. As a result, the following handlers for start- and end-tags of* `d` *nodes are produced:*

```
1  * startTag:
2    on 'd':
3      SequenceNode
4        StreamWindow[win=c/d|count 4 step 4|, type=startWindow]
5          Output[data=<win>]
6        OutputStream[state=true]
7  * endTag:
8    on 'd':
9      SequenceNode
10       OutputStream[state=false]
11       StreamWindow[win=c/d|count 4 step 4|, type=endWindow]
12         Output[data=</win>]
```

Now, each `d` *node is printed on-the-fly by means of the* `OutputStream` *operators in lines 6 and 10, which switch outputting appropriately on and off in the start- and end-tag events, respectively. To correctly nest all* `d` *nodes of a data window into a* `win` *node, the start-tag* `<win>` *is printed* before *the first* `d` *node of each data window due to the* `StreamWindow` *operator in lines 4–5. The end-tag* `</win>` *is outputted* after *the last* `d` *node by means of the* `StreamWindow` *operator in lines 11–12.* □

The following example shows how an "`on-each past`"-handler is compiled into a query plan. Basically, it is handled similar to an "`on-first`"-handler, except that the window specification determines the amount of nodes to be buffered and the point in time when the execution of the handler is triggered. Similarily to buffering data for "`on-first`"-handlers, not the entire elements of the data windows are buffered, but only the actual needed parts. This is accomplished by computing the buffer tree for the subquery of an "`on-each past`"-handler and translating it into a projection scheme for the query plan as described in Section 3.7.3.

Example 3.8.29 *Consider FluX query F_7 presented in Example 3.8.23. The interesting part dealing with data windows is again the scope of* `a`*, which has been rewritten into the following FluX query fragment:*

```
1  on a as $x return
2    { ps $x:
3      on-first past() return <result>;
4      on b as $b return {$b};
5      on-each past $x/c/d|count 4 step 2| as $w return
6        <winavg>
7        {fn:avg($w)}
8        </winavg>;
9      on-first past(c, b) return </result>; }; };
```

This part of the FluX query is compiled into the query plan shown in Figure 3.18. The handlers in lines 3, 4, and 9 are transformed into physical operators as shown in Example 3.8.27.

```
1  PSNode[symbol=a]
2    * startStream:
3      SequenceNode
4        Output[data=<result>]
5        InitializePathMatcher[path=c/d, CountWinMatcher: size=4; step=2]
6    * endStream:
7      InitializeBuffer[bufID=0]
8    * startTag:
9      on 'c':
10       PSNode[symbol=c]
11         * startStream:
12           (NOP)
13         * endStream:
14           (NOP)
15         * startTag:
16           on 'd':
17             BufferStream[bufID=0; state=true]
18         * endTag:
19           on 'd':
20             SequenceNode
21               BufferStream[bufID=0; state=false]
22               BufferedWindow[win=c/d|count 4 step 2|; type=matched]
23                 SequenceNode
24                   Output[data=<winavg>]
25                   BufferedFunction[exprID=0; path=buf[0]/d, InternalAggregate: 0]
26                   OutputExprResult[exprID=0]
27                   Output[data=</winavg>]
28         * on-first:
29           (NOP)
30     on 'b':
31       OutputStream[state=true]
32   * endTag:
33     on 'b':
34       OutputStream[state=false]
35   * on-first:
36     OnFirstHandlerNode[symbols={c}]
37       BufferedWindow[win=c/d|count 4 step 2|; type=final]
38         SequenceNode
39           Output[data=<winavg>]
40           BufferedFunction[exprID=0; path=buf[0]/d, InternalAggregate: 0]
41           OutputExprResult[exprID=0]
42           Output[data=</winavg>]
43     OnFirstHandlerNode[symbols={c, b}]
44       Output[data=</result>]
```

Figure 3.18: Fragment of Query Plan for Example 3.8.23

Again, the path defining the elements of the data window, i. e., `$x/c/d`*, is decomposed into single-step paths and appropriate scope-handlers are inserted (lines 10–29). Further, the window specification is transparently checked on the input stream as already shown in Example 3.8.27 (line 5).*

To process an "`on-each past`*"-handler the content of an entire data window is buffered. This is accomplished by the* `BufferStream` *operators in lines 17 and 21 as described in Section 3.7.3. Note that no further projection of* `d` *nodes is needed, since they only contain character content.*

Processing a complete data window is performed by the `BufferedWindow` *operator, which conditionally executes its subplan (lines 22–27) if and only if the window specification is fulfilled (which is denoted by "*`type=matched`*"). Due to scheduling this operator into the end-tag handler of* `d`*, the last* `d` *node of the data window is completely known and buffered. Hence, the subquery in lines 23–27 can be correctly executed. Having executed the subquery, the window is moved by appropriately updating the window specification and deleting the first μ elements from the buffer. Similar to the previous examples, the final data window is processed in an "*`on-first past(c)`*"-handler by means of a* `BufferedWindow` *operator of type "*`final`*" (lines 36–42). Here, the* `BufferedWindow` *operator executes the subquery only, if new window elements have been seen since its last invocation.* □

Time-based data windows are handled very similarly to element-based data windows shown in the previous example. Beforehand, we outlined whether a time-based window specification is fulfilled or not can only be determined upon seeing the next window element. We address this problem as follows: When buffering the current window element, it is marked as "pending". If the window specification is not fulfilled after having seen this element, it is contained in the current window and we remove the "pending"-mark. Otherwise, this element belongs to the next data window and must not be processed in the current data window. This is realized by modifying iterators over buffers to ignore pending nodes. Hence, the subquery can be processed as usual. Afterwards, the window is moved by removing appropriate elements from the beginning of the buffer and removing the "pending"-mark of the current element to make it visible in the data window.

We briefly show how data windows inside an "`on-first`"-handler are processed on behalf of the following example.

Example 3.8.30 *Consider FluX query F_7' presented in Example 3.8.24. Data windows are processed by means of the following FluX query fragment inside an "*`on-first`*"-handler:*

```
1  on-first past(b, c) return
2    { for $w in $x/c/d|count 4 step 2| return
3      <winavg>
4      {fn:avg($w)}
5      </winavg> };
```

This fragment is compiled into an "`on-first`"-handler of the query plan as follows:

```
1 OnFirstHandlerNode[symbols={c, b}]
2   SequenceNode
3     BufferedWindowedFor[var=$w; win=c/d|count 4 step 2|; buffer=0]
4       SequenceNode
5         Output[data=<winavg>]
6         BufferedFunction[path=$w, Matcher: d, InternalAggregate: 0]
7         OutputExprResult[exprID=0]
8         Output[data=</winavg>]
```

Because of our buffering strategy, all `d` *nodes have been buffered. The* `BufferedWindowedFor` *operator subsequently iterates over all possible data windows and binds the current data window to $\$w$, which is then processed by the operators in lines 4–8.* □

3.9 Performance Evaluation

To assess the merits of the approach presented in this chapter, we have experimentally evaluated our prototype query engine implemented in Java using a number of queries on data obtained using the XMark benchmark generator.

Our implementation supports the XQuery^{-} fragment as defined in Sections 3.4 and 3.8. We have taken selected queries of the XMark benchmark [SWK^{+}02] and, as XQuery^{-} does not include certain features that are used in these queries, we have adapted them correspondingly. In detail, attributes have been converted into subelements of their parent element in our tests[16]. Occurrences of the XPath kind test `text()` have been replaced by `{$x}`-expressions that print the whole element instead. We have eliminated `count(`$\$x$`)` aggregations by outputting $\$x$ instead. Note that these aggregations are not yet covered by the extensions shown in Section 3.8.1, since they do not directly work on the data stream, but on intermediary results.

XMark queries 1, 5, 8, 11, 13, 16, and 17 have been adjusted as sketched above. We have extracted the last FLWR subexpression of original query 20 (which computes persons whose income is not available) for our novel query 20. Further, we have introduced a novel query "8b", which basically performs a join similar to query 8, but with inner and outer `for`-loops swapped. The queries thus obtained can be found in Appendix C.1.

We have used data generated by the XMark "xmlgen" data generation tool (V. 0.96) of the sizes 5MB, 10MB, 50MB, and 100MB as input data. All tests have been performed with the SUN JDK 1.5.0_03. The XSAX parser is based on the Apache Xerces2 SAX-Parser (V. 2.6.2). The tests have been conducted on a "HP ProLiant BL20p G20" server blade (dual Xeon 2.8GHz processor, 4 GB main memory) running SuSE Enterprise Linux V. 9 (kernel 2.6.5). Note that neither FluX nor Galax (to the best of our knowledge) is optimized for multiprocessor systems.

[16] The XMark DTD was adjusted accordingly.

Our query engine is implemented precisely as described before. As a reference implementation the Galax query engine (V. 0.5.0) has been employed with projection turned on [MS03] by means of the "`-projection optimized`" parameter.

The performance of query evaluation has been studied by measuring the execution time (in seconds) and maximum memory consumption (in bytes) of each engine. The memory and CPU usage of both query engines have been retrieved by internal monitoring functions (excluding the memory consumption of the Java Virtual Machine). Unfortunately, internal monitoring of execution time of Galax has dramatically slowed down query execution, so the execution time has been measured by means of the Unix "time" command. The times taken for rewriting an XQuery into FluX are negligible—namely in the order of the time needed for parsing the query itself—and thus are not reported separately in our experiments.

3.9.1 XSAX Parser

To get an idea of the maximum achievable performance, i. e., the minimum execution times or maximum throughput, of the FluX query engine, we first evaluate the efficiency of the XSAX parser.

We have made two tests, each varying the size of the XMark input document. The first experiment has been to parse the input document in lazy validation mode having not registered any "`on-first`"-events. Hence, no automata for generating ofp-events are produced and the whole input document is not validated. This test denotes the absolute upper bound of the performance for executing FluX queries. In the second experiment, we have parsed the input document in full validation mode. Here, for every production a validating automaton is generated and the whole input document is thus validated. Therefore, this experiment constitutes the worst case with respect to the XSAX parser. Note that this worst case is independent from the actual number of registered "`on-first`"-events, since ofp-events can be generated during the simulation of the automata without any additional cost. With respect to the XSAX parser, only the number of productions, which are being validated, determines its performance. See Section 3.7.1 for further details of the XSAX implementation.

Figure 3.19 shows the results of these experiments. To minimize side-effects, the depicted execution times are the average of five runs. As expected, parsing time in lazy validation without having registered any "`on-first`"-events scales linearly with the size of the input document. The overhead of validating the whole input document, i. e., the maximum overhead for generating ofp-events, is low (27% on the 100MB document). Also, validating the whole document scales linearly with the size of the input document. The throughput of the XSAX parser, which is given as the number of produced SAX-events per seconds, is relatively constant—apart from the 5MB and 10MB documents. Here, the overall parsing time is relatively small, so side-effects such as class-loading or byte-code compilation have a noticeable influence and lower the throughput.

	Lazy Validation		Full Validation	
	Time [s]	Throughput [$\frac{kEvents}{s}$]	Time [s]	Throughput [$\frac{kEvents}{s}$]
XMark 5MB	0.562	672	0.799	472
XMark 10MB	0.904	836	1.083	697
XMark 50MB	3.558	1054	4.352	862
XMark 100MB	6.575	1141	8.349	898

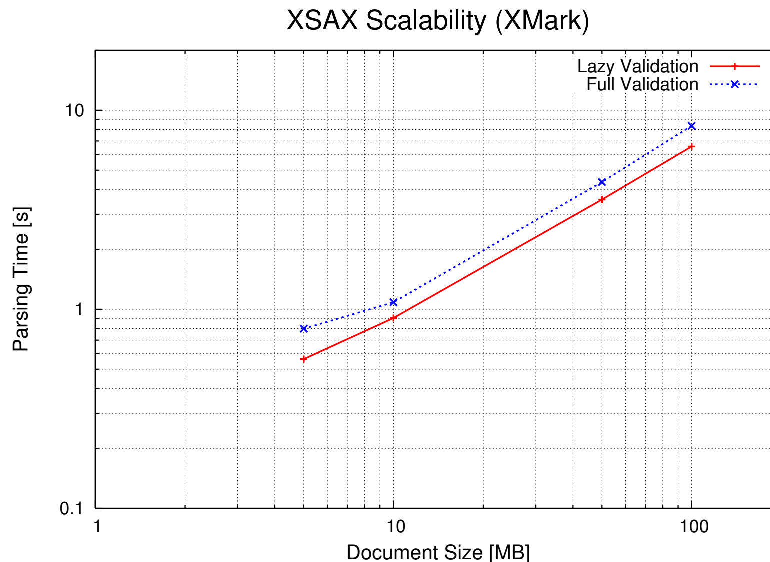


Figure 3.19: XSAX Performance

3.9.2 Basic Performance Tests

In this section, we show the overall performance of our FluX query engine with respect to execution time, memory consumption, and investigate its scalability. Therefore, we employ the adapted XMark queries as briefly described at the beginning of this section. Again, to minimize side-effects all execution times have been averaged over three independent runs. Detailed benchmark results can be found in Section C.3.

Figure 3.20 shows the execution times of our queries varying the size of the input document. The results of our FluX query engine have been obtained exploiting order constraints of the DTD and applying algebraic optimizations, which will be referred to as *fully optimized* in the remainder. Obviously, our FluX query engine (blue bars) outperforms Galax (red bars) on each of our queries and input documents. The rapid increase in execution time, which can be observed for queries 8, 8b, and 13, is due to the fact that these queries contain joins. We currently compute joins by a naive nested-loops algorithm, which clearly is a drawback with respect to efficiency. Note that query 11 could not be measured on the 100MB document with Galax, since an “Out-Of-Memory” error occurred.

Next, Figure 3.21 shows the memory consumption of our FluX query engine (blue

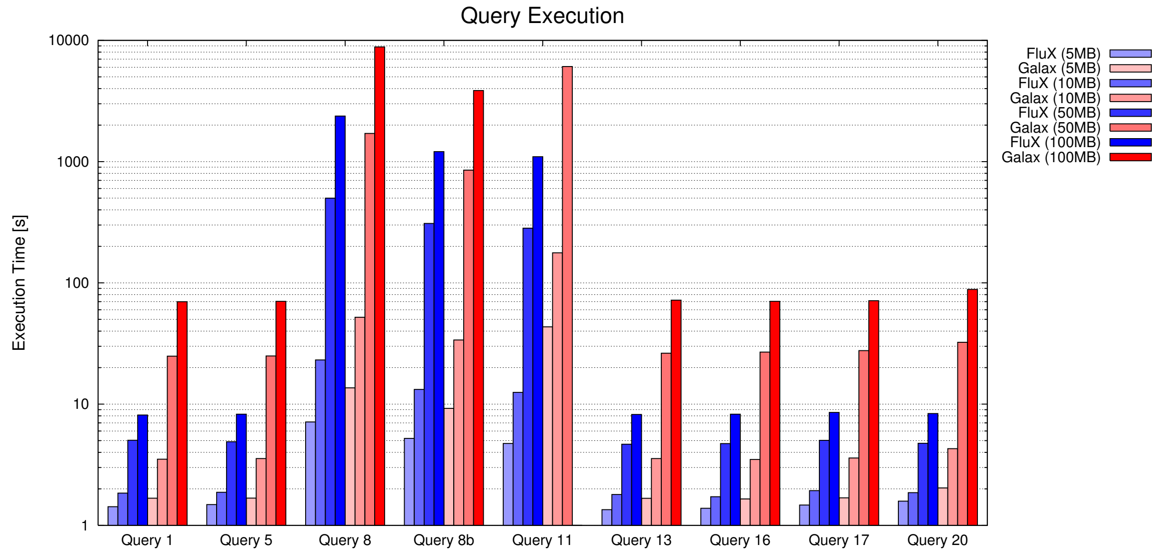


Figure 3.20: FluX Overall Performance: Execution Time

bars) and Galax (red bars) for each query varying the size of the input document. As before, FluX queries have been executed fully optimized. Obviously, independent from the actual query, Galax needs a relatively constant amount of memory decided by the actual size of the input document. In contrast, memory consumption of our FluX query engine is mostly decided by the query (and, of course, the underlying DTD) and clearly outperforms Galax. Queries 1, 13, and 20 can be evaluated directly only the input stream without buffering at all because of the order constraints imposed by the DTD. Queries 5, 16, and 17 show a constant amount of memory consumption independent from the size of the input document. This is due to the fact that only a (singular) current element of the input stream has to be buffered at a time for being able to correctly process it. Having processed this element, the buffer is freed and filled with the next element. Further, due to our projection scheme, only the actual needed part of these elements is buffered, which additionally lowers memory usage. Queries 8, 8b, and 11 perform a join on two subtrees (i. e., of `people` and `closed_auction` respectively `open_auction`) and therefore inevitably have to buffer all affected elements or join partners, respectively. Nevertheless, due to our effective projection scheme, only a small fraction of the original data is buffered compared to Galax. Because of the order constraints imposed by the DTD, when evaluating Query 8b we are further able to exploit the fact that all subtrees of one join partner (`person`) are fully known before we see the subtrees of the other join partner (`closed_auction`) on the stream. Hence, only subtrees of the first join partner have to be buffered and the join can be evaluated on-the-fly as the other join partners are seen on the stream. This yields a reduction of memory usage by approximately a factor of ten compared to original Query 8 (remember, that Query 8 and Query 8b basically compute the same join besides having swapped inner and outer loops).

Altogether, our optimization approach performs very well with respect to execution

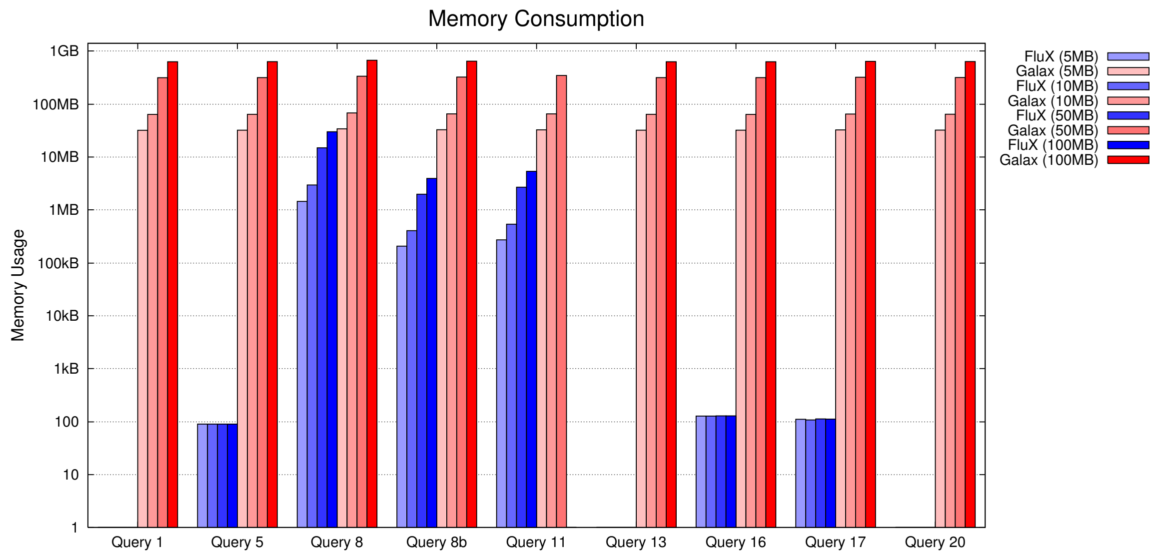


Figure 3.21: FluX Overall Performance: Memory Consumption

time and maximum memory consumption.

We further investigate the performance of our optimization techniques by comparing the execution time and memory usage of our query engine utilizing different levels of optimizations during rewriting an XQuery into FluX and executing it.

Figure 3.22 shows the execution times of our benchmark queries applying different levels of optimization and using the 50MB document as input. “Fully Optimized” denotes the situation of exploiting order constraints imposed by the DTD and applying algebraic optimizations. For the “Not Optimized” experiment, order constraints have been ignored—which is equal to using a very weak DTD, which does not impose any order constraints—and no algebraic optimizations have been performed. In case of “Compiled”, all queries have been fully optimized and eventually the query plan has been compiled into stand-alone Java code, which can be directly executed. Further, the dotted red line in the figure marks an average parsing time of the input document. First, it can be observed that the execution time of Queries 1, 5, 13, 16, 17, and 20 is mostly determined by the parsing time of the document. These queries filter or transform special elements of the stream, which does not result in high computational complexity. Hence, only a slightly better performance can be achieved by compiling those queries into Java code. Since there is only one element at a time involved in the computation (which is even reduced in its size due to our projection scheme), it practically makes no difference whether this element is processed buffered (in the not optimized case) or fully streaming (in the fully optimized case) with respect to execution time. The situation is different for Queries 8, 8b, and 11. Those queries have a high computational complexity (the join), which dominates the parsing time of the document. Here, performing a full optimization improves the performance. This is due to the fact that algebraic optimizations are able to eliminate and merge (duplicate) `for`-loops of the queries, which obviously pays off during the execution of the queries. Due

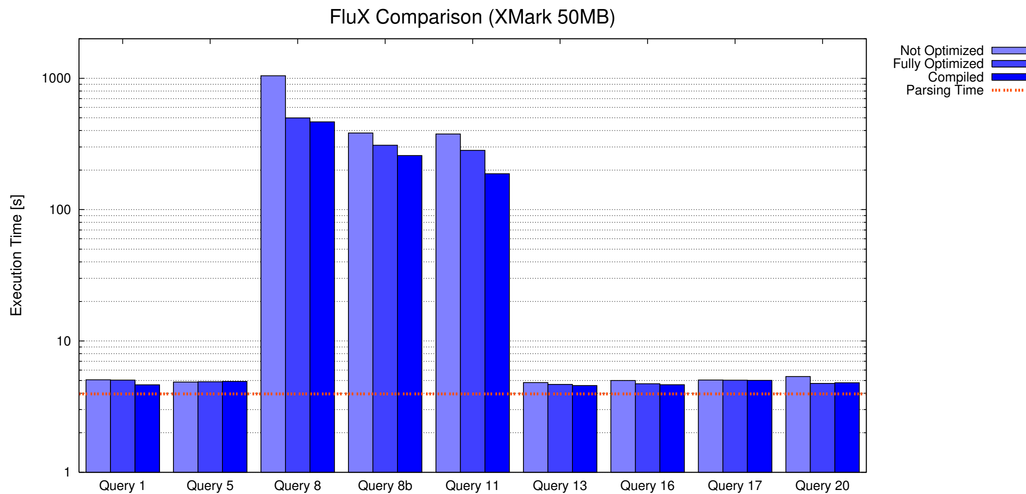


Figure 3.22: Comparison of FluX Optimization Variants: Execution Time

to the high computational complexity, compiling the query plan into stand-alone Java additionally improves performance. This is the result of directly compiling structures of the query plan into corresponding Java constructs, which may additionally be optimized by the Java compiler on the byte-code level. For example, loops can be directly compiled into corresponding Java loops, while for the normal (interpreted) execution these loops are wrapped inside general physical operators.

Figure 3.23 shows the memory usage of the same experiment. Here, the "Compiled" case is omitted, since there is no difference in memory consumption compared to the "Fully Optimized" case. In the "Not Optimized" case, Queries 1, 5, 13, 16, 17, and 20 have to buffer a single element at a time[17]. If order constraints imposed by the DTD are taken into account, Queries 1, 13, and 20 can be optimized such that they can be evaluated without any buffering. Since Queries 8, 8b, and 11 perform a join, they inevitably have to buffer all join partners. Exploiting order constraints, memory consumption of Query 8b can be reduced by approximately a factor of ten by buffering only one of the join partners.

Altogether, it can be observed that even unoptimized FluX queries, i. e., using a DTD without any specific order of elements, show a good performance and very low memory consumption. Exploiting order constraints imposed by the DTD, memory consumption can further be significantly reduced when processing well suited queries.

We conclude this basic performance evaluation by having a closer look at the scalability and the throughput of our query engine. Figure 3.24 shows the scalability of our query engine. For each query, the execution times are normalized to that of the 5MB document. Input documents of sizes 5MB/50MB are shown in red and input documents of sizes 10/100MB in blue color, i. e., for each color, the step from a circle to a filled circle is increasing the input size by a factor of ten. Obviously, Queries 1, 5, 13, 16, 17, and 20 scale

[17]This can be verified by looking at the detailed results shown in Figure C.1. For these queries, the memory consumption is practically constant—independent from the size of the input document.

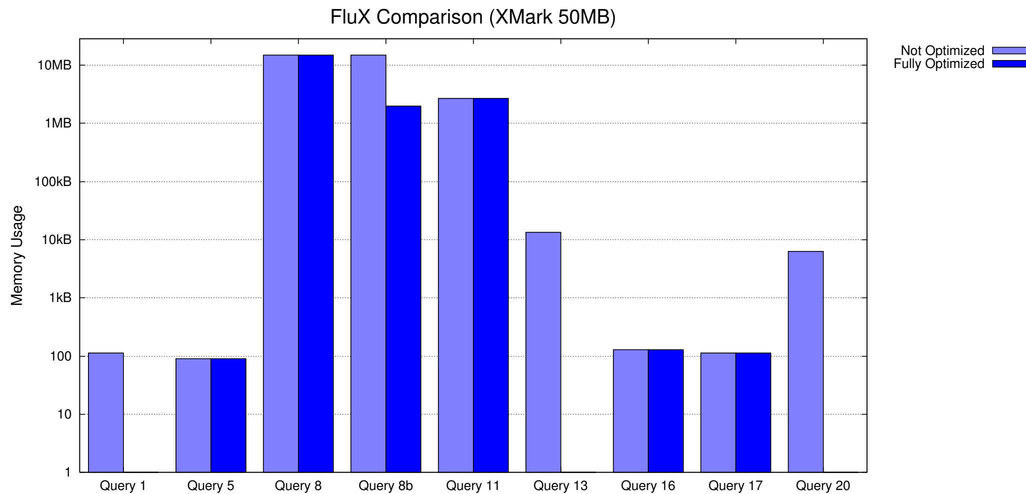


Figure 3.23: Comparison of FluX Optimization Variants: Memory Usage

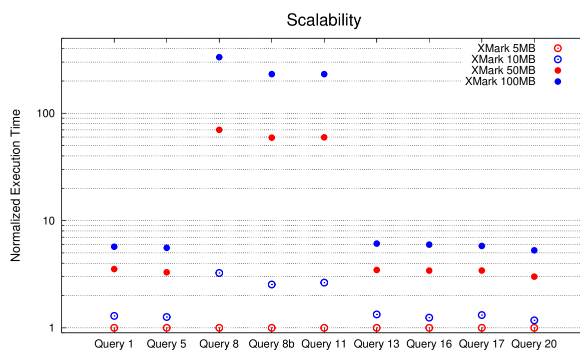


Figure 3.24: Scalability

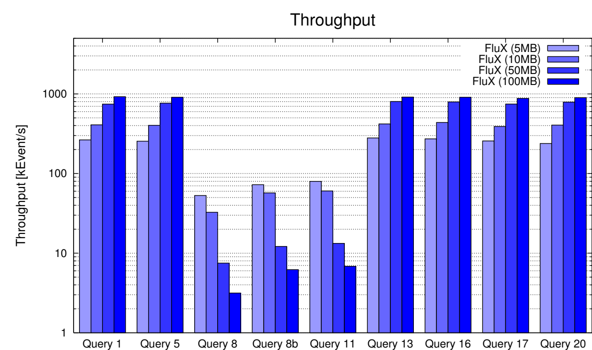


Figure 3.25: Throughput

very well with the size of the input document (execution times raise lower than a factor of ten upon increasing the size of the input document by a factor of ten). Queries 8, 8b, and 11 scale quadratically with the size of the input document due to performing a nested-loops join. Figure 3.25 shows the throughput of the query engine of each query varying the size of the input document. The throughput is defined as the number of processed input SAX-events per seconds. As expected, Queries 1, 5, 13, 16, 17, and 20 show a high and relatively constant throughput. Due to their computational complexity, Queries 8, 8b, and 11 naturally show a lower throughput. Less throughput on processing smaller input documents can be traced back to parasitic effects such as class-loading, byte-code compilation, etc., which in this case have a bigger impact due to low overall execution times.

3.9.3 Pipelining Behavior and Buffer Allocation

In this section, we examine the pipelining behavior and memory consumption of our query engine using some exemplary queries. Therefore, we plot the number of produced output

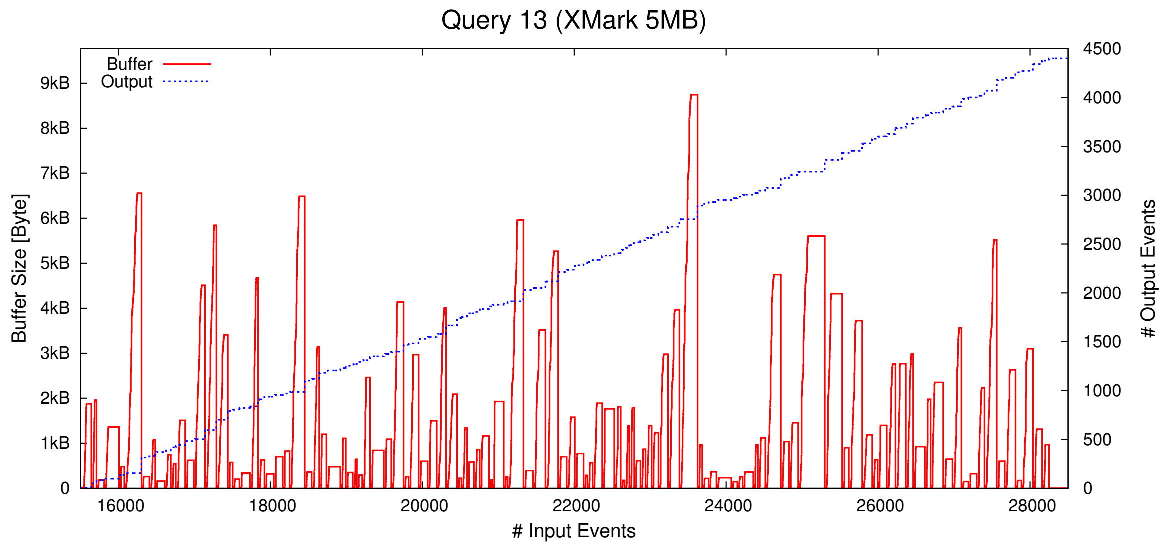


Figure 3.26: Execution of Query 13

events and the current memory usage versus the number of processed input SAX events.

Figure 3.26 illustrates the execution of Query 13 on the 5MB XMark document. For presentation purposes, this query was executed ignoring order constraints to also be able to show buffer usage (remember, that in the fully optimized case this query can be executed without any buffering at all). The x-axis shows the number of processed SAX events. To obtain more details, we have narrowed the range to the relevant part of the document. The number of produced SAX events is shown on the right y-axis and the current buffer size is shown on the left y-axis. It can be observed that while processing the input stream of the document, the results are continually outputted (dotted blue line). With respect to buffer usage, the figure proofs the claim, which we have made in the previous section: Each current element (here, an item of Australia) to be processed is first buffered, then processed after it has been completely buffered, and afterwards, the buffer is freed. The actual size of the buffer is not constant, because the contents of each `item` element, in detail, the `description`, are different. When executing this query fully optimized, outputs are produced even more continually right after having seen the corresponding input events. Here, a whole element is outputted after this element has been completely buffered and processed (which may be observed, e. g., at the spike right before input event 24000).

Figures 3.27 and 3.28 compare the execution of Queries 8 and 8b. Remember, that these two queries are more or less identical except from having swapped inner and outer loops of the join. This time, both queries are executed fully optimized. As shown in Figure 3.27, Query 8 buffers both join partners (from approximately input event 0–225000 and 325000–400000). As soon as all elements are buffered, the whole output is produced in the corresponding ofp-handler and the buffer is cleared. Figure 3.28 illustrates the execution of Query 8b. As already explained before, only the first join partner has to be buffered for this query, which are again the events 0–225000 (approximately). In contrast

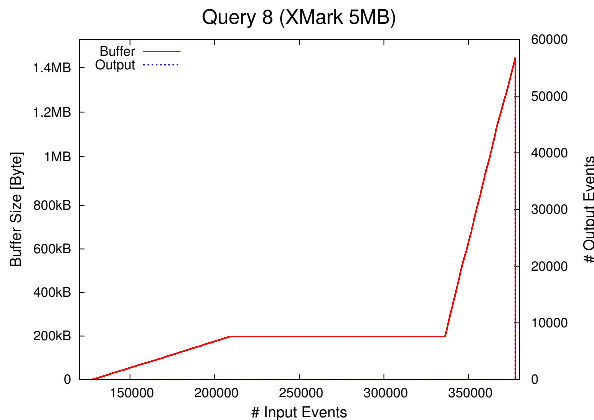


Figure 3.27: Execution of Query 8

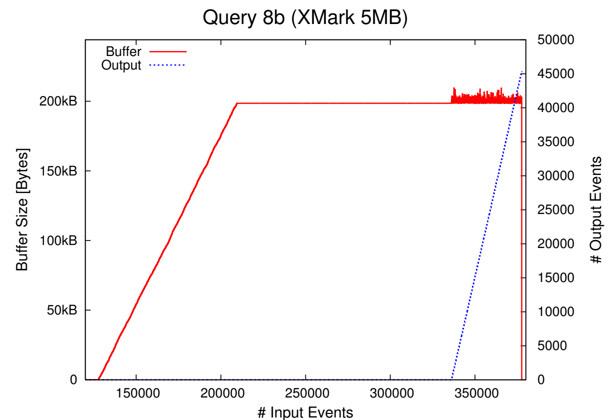
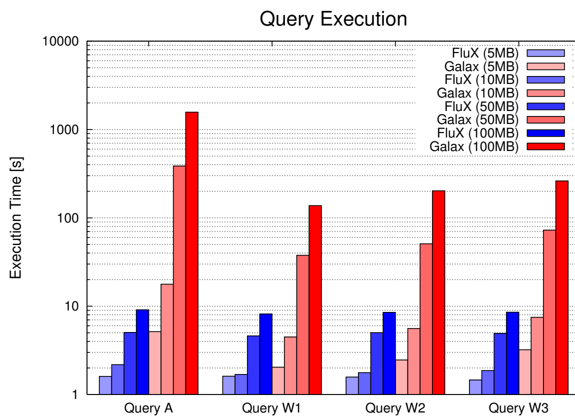


Figure 3.28: Execution of Query 8b



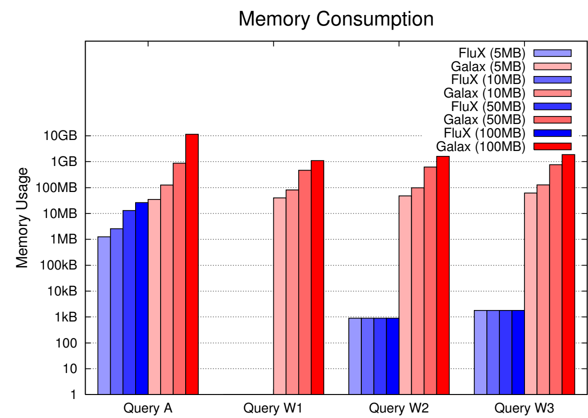


Figure 3.29: Performance of FluX Extensions

to Query 8, when reaching the elements of the second join partner on the stream (starting approximately with event 325000), only the current element is buffered (visible as the "noise" at the end of the red line). The join of this current element with all buffered elements is computed and the results are immediately outputted. Hence, results are continually produced while processing this part of the input stream.

3.9.4 Extensions: Aggregate Functions and Data Windows

In this section, we briefly assess the performance of our extensions of the basic FluX query engine, i. e., aggregate functions and data windows, by means of some sample queries. For this purpose, we employ the additional queries shown in Section C.2, which are not part of the XMark benchmark.

Figure 3.29 shows the execution time and the memory consumption of the additional queries varying the size of the input document. Detailed benchmark results can be found in Section C.3. Clearly, our FluX query engine outperforms Galax both with respect to

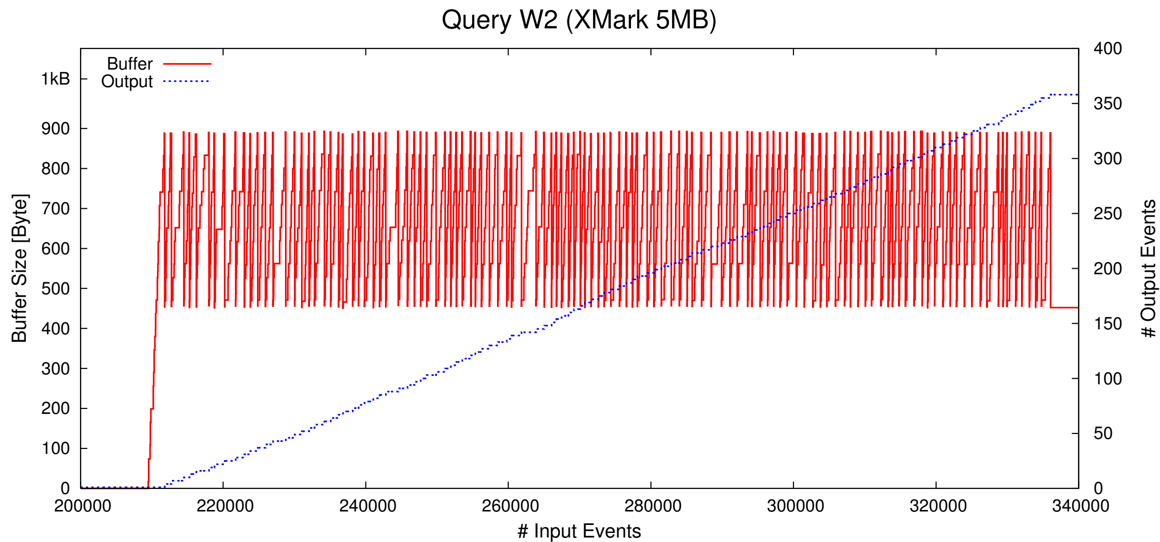


Figure 3.30: Execution of Query W2

execution times and memory consumption on these queries. With respect to execution times, it can be observed that our FluX query engine scales to increasing sizes of the input document. Due to our effective projection scheme, Query A shows a good performance in respect of memory usage. Note that this query has to buffer all closed items to be able to print those with a maximum price. Query W1 has a simple data window and hence can be executed without any buffering directly on the input stream. Queries W2 and W3 cannot be executed directly on the stream since subsequent data windows overlap. Hence, always a single data window is buffered. Obviously, its size is independent from the actual size of the input document. Clearly, these strategies of using data windows as a sliding view of the data stream perform much better with respect to memory consumption than an unoptimized evaluation. Note that Query W3 uses a data window twice as large as it is used in Query W2 and thus needs twice as much memory[18].

Figure 3.30 shows a detailed analysis of the streaming behavior and buffer usage, which already has been introduced in the previous section for other queries, of Query W2 on the 5MB XMark document. For presentation purposes, only the relevant part of the document is shown. Again, while processing the data windows results are continually produced in a streaming fashion. It can be easily seen that all elements of the current data window are buffered until the window specification is fulfilled. Then, results are output and the window is moved by discarding elements from the beginning of the buffer according to the window specification (here, half of the elements are thrown away).

[18]Note that the y-axis in the Figure is in a logarithmic scale. See Figure C.2 for the exact values.

3.10 Related Work

In this section, we give an overview of some work relevant to the problem of query evaluation on XML data. Since XPath and XQuery have been proposed as W3C Recommendations, much research has been done to efficiently implement these standards. These works can roughly be classified into query evaluation on *stored XML data* and query evaluation on *XML data streams*. Since our goal is efficient query evaluation on data streams, we focus on works of the latter class. Works of the former class might be of interest under certain circumstances, e. g., whenever working on buffered data. In such situations, techniques developed for stored XML data, such as special indexes supporting the evaluation of path expressions or compression techniques, could play a role for further improving query evaluation on buffered data. However, since the current status of our FluX query engine does not yet employ such techniques, we do not cover this research area in this section.

Algebras for XML Query Processing

An important goal is to devise a well-principled machinery for processing XQuery. Such machinery needs to be based on intermediate representations of queries that are syntactically close to XQuery and has to allow for an algebraic approach to query optimization. This is necessary to allow for both extensibility and the leverage of a large body of related earlier work done by the database research community.

An algebra for XQuery is part of the XQuery standard [W3C05c]. Several other works propose other approaches of XQuery algebras meant for conventional query processing, e. g., [FSW01, JLST01, BT99]. The work in [FLBC02, BF05] aims at applying such an XQuery algebra in the streaming context.

To the best of our knowledge, no principled work exists on query optimization in the framework of XQuery (rather than automata) for *structured data streams* which honors the special features of stream processing, i. e., having buffer minimization as an optimization target or exploiting DTD information.

XML Stream Transformation Languages

Some works aim at efficient and scalable transformation of an XML stream by defining languages which only allow for a single pass over the XML stream.

STX [Bec04] is an XML transformation language being geared to XSLT [W3C05e]. STX processes an XML document as a (single-pass) data stream. It restricts the view to the input data by enabling the access only to the ancestors of the current context node. Thus, a scalable execution on documents of an arbitrary size is achieved.

XSAGs [KS03] are a novel class of attribute grammars specifically designed for scalable XML stream processing. XSAGs are based on extended regular tree grammars, which are normally given by the DTD of a data stream. An XSAG is obtained by annotating a given extended regular tree grammar with attribution functions that describe the output

to be produced from the input stream. XSAGs can be evaluated strictly in linear time in a streaming fashion, consuming only a stack of memory bounded by the depth of the XML tree being streamed. The TransformX framework [SK05] is a generalization of attribute grammars used in XSAGs. Here, attribution functions are allowed to be arbitrary Java code. The attributed tree grammars are translated into Java source code in the style of parser generators such as YACC. The output of a TransformX attribute grammar need not be XML. Rather, TransformX attribute grammars may be regarded as a comprehensive programming language for XML streams, with all the capabilities provided by Java.

STX and XSAGs only allow for transformations which can be computed using the restricted view of the input stream (in STX) or even without any buffering at all (in XSAGs). This supports for somewhat expressive transformations, but not for general queries. TransformX enables the whole expressive power of Java, especially allowing users to buffer (and process) parts of the data stream on their own. This basically enables the computation of arbitrary transformations (or queries), but since buffering is done inside the Java attribution functions, the framework is not able to optimize the execution with respect to minimize memory consumption.

Processing XPath Queries on Data Streams

The efficient evaluation of XPath queries on data streams has been worked on extensively in the past. These works can basically be classified into *Boolean* and *node-selecting* XPath queries.

Works on Boolean XPath queries, such as [CFGR02] or [AF00], are primarily intended to be used in matching XPath expression against XML documents for selective dissemination of XML documents according to user interests. Relevant work in this area has already been discussed in Section 2.5.3.

In contrast, node-selecting XPath queries produce a set of nodes as a result. In [GMOS03, ACGG+02] the authors propose deterministic finite automata (DFAs) for evaluating XPath expressions on data streams. To overcome the problem of the huge number of states of such a DFA, they suggest a lazy construction of the DFA. Although predicates can be handled in this approach, the efficiency decreases with the number of predicates. This problem is addressed by XPush machines [GS03], which is a modified deterministic pushdown automaton (PDA). A given set of XPath expressions is compiled into a single XPush machine. To avoid the theoretical exponential state blow-up, the XPush machine is again computed lazily. Further, optimizations accounting for order constraints given by a DTD to reduce the number of states of the PDA are presented. XSQ [PC03] compiles pushdown transducers augmented with buffers for building blocks of XPath expressions into a single hierarchical pushdown transducer representing the whole XPath expression. XSQ pushdown transducers only buffer the minimum amount of data needed for processing the XPath query and further support aggregate functions. Another approach built on top of transducers is SPEX [OKB03]. Here, regular path expressions are translated into a network of transducers. SPEX also supports regular path expressions with qualifiers. The

authors of [BCG+03] present an algorithm for streaming XPath expressions with forward and backward axes, which is not based on automata techniques. The key idea of this work is to convert backward into forward axes and to match the input stream by means of a directed acyclic graph representation of the XPath expression.

Although path expressions are never completely evaluated in FluX, we have adopted some basic ideas of these works for matching paths on the data stream, e. g., for checking atomic conditions and computing function results. These works might gain further importance in future work, e. g., when considering paths containing predicates.

In [BGK03] the authors present a compression technique for XML documents and provide algorithms for directly and efficiently executing path queries. Even though this area is not directly related to FluX, such compression techniques could be of interest for further reducing the needed amount of main memory, if some parts of the data stream must be buffered for executing certain queries.

XML Query Processing on Data Streams

Compared to XPath 1.0, XQuery is more expressive and therefore involves additional challenges. Several recent projects have addressed evaluating XQueries on data streams.

The authors of [LMP02] propose XML Stream Machines (XSMs) for efficient stream processing. XSMs are transducers possibly having more than one input stream augmented with buffers. XQueries are translated into a network of XSMs, which are connected by means of buffers, based on building blocks of XQueries. This network is then compiled into a single XSM. It is shown that schema information can be exploited to reduce the size of the XSMs. In contrast to our work, schema information (e. g., order constraints) is not used for minimizing buffer usage. Automata-based techniques are usually quite elegant but are hard to compare or integrate with other approaches and usually do not generalize to real-world query languages such as (full) XQuery with their great expressive power and all their odd features and artifacts of the standardization process.

Raindrop [SRM05] also employs an automata-based execution model for XQuery evaluation. Similar to our approach, schema constraints are exploited to optimize the execution of an XQuery. In detail, schema information is used to terminate computations early, e. g., if it can be determined that an element needed for satisfying a predicate cannot be encountered in the future (similar to our concept of "`on-first past`"-events). This reduces processing cost and, at the same time, may lead to reduced buffer usage, since a buffer might be freed early on, or certain initially buffered elements are no longer buffered. However, schema information is not directly used for minimization of buffer usage. That is, basically all elements that need to be processed are buffered as a start. The techniques for early terminating computations are orthogonal to our approach and thus could also be applied in FluX.

In [LA05] the authors present an approach for directly compiling XQueries into executable Java code. They present a methodology to determine whether an XQuery can be correctly executed with only a single pass over the data stream. Further, some optimiza-

tions are shown to rewrite XQueries initially having to perform multiple passes over the data stream into equivalent queries, which only perform a single pass. This is achieved by merging loops if possible, which is similar to some of our algebraic optimizations. Buffer minimization is no direct optimization goal. However, they employ similar projection techniques as we do to reduce memory usage.

The Tukwila XML Engine [IHW02] employs the X-scan operator for evaluating paths on streaming XML data and generating variable bindings. Bindings are encoded in binding tuples and processed by a set of conventional query operators. Such query plans are optimized very similar to what is done in relational DBMS. Particularly, they do not account for buffer conscious query execution by any means.

The BEA/XQRL [FHK+03] query engine supports pipelined processing of XML data streams by implementing the iterator model at expression level. Like the Tukwila XML Engine, query optimization is not tailored to the peculiarities of processing data streams, and large documents cannot be processed.

Moreover, the problem of optimizing XQueries using a set of constraints holding in the XML data model – rather than a schema – was addressed in [DT03]. The optimization approach taken there is based on a mapping of certain XQueries to relational conjunctive queries and the use of a chase/back-chase procedure for query simplification.

One approach [MS03] towards addressing the problem of reducing main memory consumption in an engine for full XQuery on materialized XML documents aims at reducing the amount of data buffered in main memory by pre-filtering the data read from the stream with the paths occurring in the query. However, for real-world XQueries, the need for substantial main memory buffers cannot be avoided in general. We have adopted this technique in our FluX query engine for further reducing the size of buffers needed for query execution.

3.11 Discussion

Main memory is probably the most critical resource in (streamed) query processing. Keeping main memory consumption low is vital to scalability and has—indirectly—a great impact on query engine performance in terms of running time.

The main contribution of this work is the FluX language together with an algorithm for automatically translating a significant fragment of XQuery into equivalent FluX queries. Our algorithm uses schema information to schedule FluX queries so as to reduce the use of buffers. The FluX language itself—while intended as an internal representation format for queries rather than a language for end-users—provides a strong intuition for buffer-conscious query processing on structured data streams.

FluX, with its intuitive direct way of evaluation, also provides a good measure of relative utility of equivalent XQuery expressions (with respect to memory consumption). This allows for the algebraic optimization of XQueries in order to minimize buffers.

As evidenced by our experiments, our approach indeed dramatically improves the scalability of main memory XQuery engines, even though we think we are not yet close to exhausting this approach, neither with respect to run-time buffer management and query

Q_8	**Nested-Loops Join** Execution Time [s]	**Hash-Based Join** Execution Time [s]
XMark 5MB	7.1	1.8
XMark 10MB	23.1	2.4
XMark 50MB	499.6	7.4
XMark 100MB	2375.5	14.5

Figure 3.31: Join Optimization on the Example of Query 8

processing, nor query optimization.

In particular, an optimization is to push `if`-expressions—which we have moved down the query tree to obtain our normal form—back "up" the expression tree as soon as the other simplifications have been realized. This avoids possibly unnecessary iterations of buffers in case of un-satisfied conditional subexpressions. A subject of our current research is a more sophisticated evaluation of joins. We already have implemented first prototype join optimizations and a hash-based join operator instead of the nested-loops evaluation strategy. Figure 3.31 shows first promising results on the example of Query 8 of the XMark benchmark. Besides very little overhead for maintaining hash-tables, the same parts of the stream are buffered like in the nested-loops case. Also, we are currently extending windowed computations to more than one data window on the data stream, e. g., needed for computing window-joins.

Chapter 4

The *Best-Match Join*

As already outlined in Section 2.2, our StreamGlobe data stream management system enables the execution of new expressive query operators for information retrieval by means of user-defined operators. “Interesting” information is often obtained by combining dynamically generated data streams. Such an operation is needed, for instance, in our astrophysical example scenarios, which have been presented in Section 2.4, for classifying observed objects. Here, a common problem is finding best matching pairs of data objects given user-defined multi-dimensional criteria. Traditional aggregation-based techniques (e. g., *min* or *max*) in conjunction with rating functions do not give satisfying results, because a single “best” pair cannot be determined, since diverse pairs, each being best in different aspects of the comparison, are interesting. We propose the novel class of *best-match join (BMJ) operators* to solve this problem. Unfortunately, the BMJ operators are inherently blocking (pipeline-breakers), such that, in their basic form, they are not applicable to streaming data or (practically) “infinite” data sources. To overcome the blocking nature and to improve the quality of the results we propose the *constrained BMJ operators*. The constraints in combination with physical properties of the data stream, i. e., being ordered according to a constrained (best-match) join attribute, enable our new pipelined BMJ algorithms, which are based on synchronously shifting windows over the data streams. To assess our algorithms, we present experimental results of our StreamGlobe prototype implementation, which demonstrate the required non-blocking behavior and the good performance of our BMJ algorithms.

The work presented in this chapter has been published in the *Technical Report MIP-0204* of the *Universität Passau* [KS02].

This chapter is outlined as follows. At first, we motivate our best-match join operators on behalf of some example application scenarios. In Section 4.2 the class of BMJ operators is formally defined, basic evaluation methods are presented, and the constrained BMJ operators are introduced. The window-based algorithm, its optimizations, and special application scenarios dealing with time-stamped data streams are shown in Section 4.3. Experimental performance results are discussed in Section 4.4. We conclude this chapter by presenting some related work in Section 4.5 and a discussion in Section 4.6.

4.1 Motivation

Information retrieval networks, such as our StreamGlobe data stream management system, which we have presented in Section 2, enable users to globally exchange and query data provided all over the world. Often, complex queries have to be processed to retrieve exactly the information the users are interested in from this huge volume of "raw" data. A common problem is, for instance, finding best matching pairs of data objects provided by different data sources given user-defined criteria. This is done by comparing the data objects in multi-dimensional spaces, which leads to a partial order on the pairs of data objects. Because partial orders naturally do not have a single minimum, diverse pairs, each being best in different aspects of the comparison, are interesting. Hence, traditional aggregation-based techniques in conjunction with rating functions based on determining a single "best" pair fail to produce satisfying results.

We propose the novel class of *best-match join (BMJ) operators* to solve this problem. Nearest neighbor operators or closest point algorithms use an overall distance between two data objects to determine closest pairs. This overall distance is based on all dimensions involved in the comparison. In contrast, our new BMJ considers each dimension individually. The quality of the match of a pair of data objects given a special attribute is determined by user-defined functions, e. g., the distance, set inclusion, etc. Hence, the BMJ computes the best matching pairs of data objects having a maximum similarity on each individual join attribute. The result contains those pairs for which no better matching pair exists. Computing BMJ operators substantially differs from the conventional join, because tentatively computed results might become invalid if a better matching pair is found later on.

Unfortunately, these basic BMJ operators are inherently blocking operators (pipeline-breakers). Thus, they are not applicable on streaming data, because all input data has to be processed before any results can be delivered. Furthermore, less interesting pairs matching very well in one, but worse in all other dimensions may be contained in their result. Therefore, we introduce reasonable restrictions to the basic BMJ operators to improve their results in common application scenarios. These constraints (sometimes also denoted as "window predicates") in combination with physical properties of the data streams, i. e., being ordered according to a constrained join attribute, constitute data windows on the input data streams. We propose an algorithm for computing BMJ operators based on synchronously shifting these data windows over the input data streams, which enables processing of unbounded data streams requiring only a limited amount of main memory and storage capacity. In contrast to other windowing techniques known in the literature we do not assume that the data windows have to fit into main memory and show how to efficiently employ secondary storage for processing large data windows. Because of the pipelined execution, this algorithm is suitable for application in modern stream processing query engines. In particular, we have implemented the best-match join operators as external operators in our StreamGlobe framework.

The BMJ operators for computing best matching pairs of two data sources proposed in this thesis cover a wide spectrum of applications. BMJ operators are not tied to a special data model. Hence, to ease presentation we use the relational data model and SQL

as a query language in the following. In Section 4.2.2 we will also show how to phrase best-match joins in XQuery.

Example 4.1.1 *A first application scenario is assembling a composite part out of two base parts part1 and part2. Various suppliers provide different models of these two parts. Since we want to build the "perfect" composite part, we are only interested in those combinations of the two parts which, e. g., have the lowest cost, minimum overall tolerance, lowest overall weight, and are provided by high quality providers. A single best composite part probably cannot be determined, because the cheapest composite part cannot be compared to a more expensive one with a better tolerance, and so on. In the context of a relational database, the best matching pairs of subparts could be determined using the following SQL query, which considers the cost and weight of subparts for computing best matches as an example:*

```
select   *
from     part1 p1a, part2 p2a
where    not exists (
  select   *
  from     part1 p1b, part2 p2b
  where    p1b.cost + p2b.cost <= p1a.cost + p2a.cost and
           p1b.weight + p2b.weight <= p1a.weight + p2a.weight and
           (p1a.partno <> p1b.partno or p2a.partno <> p2b.partno));
```

Using our syntactical extension of SQL, which will be introduced in Section 4.2.2, this query can be written much more conveniently as:

```
select   *
from     part1 p1   bestmatch join   part2 p2
           on (p1.cost + p2.cost) min, (p1.weight + p2.weight) min;
```

□

Example 4.1.2 *As another application scenario consider a recruitment agency. The task of such an agency might be to find all the best job seekers for each given project advertised as a post. Whether a job seeker is suitable for a given project is determined upon attributes such as the grade, professional qualifications, the distance of the job seekers home to the company, and the experience of the job seeker. Again, comparing all pairs of open projects and job seekers by means of a single rating function does not deliver satisfying results, because for a given project a job seeker with a better grade might be of interest as well as a job seeker with a worse grade, but having more experience. Again, in the relational context we could compute these best matching pairs by executing the following SQL query considering the distance[1], grade, and experience of job seekers as an example:*

[1] We assume that some user-defined function "dist(x, y)" is defined, which computes the distance between its two arguments.

```
select   *
from     project op, person p1
where    not exists (
  select    *
  from      person p2
  where     dist(op.location, p2.location) <= dist(op.location, p1.location) and
            p2.grade >= p1.grade and
            p2.experience >= p1.experience and
            p1.personID <> p2.personID);
```

In our extension of SQL this query is succinctly written as:

```
select   *
from     project op  left outer bestmatch join  person p
           on (dist(op.location, p.location)) min,
               (p.grade) max,
               (p.experience) max;
```

□

Obviously, using multi-dimensional comparisons, the standard SQL queries to determine all best pairs become very complex. Also, it should be obvious that standard SQL evaluation techniques for these correlated subquery formulations are extremely inefficient. Therefore, we propose the above (slight) syntactical extensions of SQL to express our new best-match join operators in a succinct and convenient syntactical format.

Example 4.1.3 (Running Example) *In this chapter, we employ as a running example the matching of different sensor data streams. Consider a weather service which determines the probability of rain at certain locations. For this purpose, it may need the temperature and humidity at these locations. We assume that it is too expensive for the weather service to establish their own meteorological stations. Various providers make the measurements of both types of sensors available through the Internet. These sensors are independently spread over the observed region and moreover it is not necessary for a specific location to accommodate both types of sensors. The measuring data of all sensors are multiplexed by StreamGlobe into two data streams; one data stream contains all temperature data and the other all humidity data. Every data object of these data sources contains the sensor reading, the measurement time, and the (two-dimensional) location of the sensor. To be able to estimate the probability of rain, the weather agency has to find pairs of temperature and humidity measuring data, which are both locally and temporally close together. Additionally, the pairs must satisfy the following requirements to be usable for the estimation:*

- *The time between the measurement of the temperature and the humidity must not exceed 10 minutes.*
- *The distance between the sensors for temperature and humidity must not exceed 100m in each dimension of the sensor's coordinates.*

Since the individual sensors most likely deliver their measurements with different update rates, it is not possible to make a static assignment of sensors to be used as pairs. This assignment changes with time and data being currently available. □

4.2 Definition of the Best-Match Join Variants

In this section, we give a formal definition of the family of BMJ operators. For the sake of a more convenient presentation of the BMJ semantics, we do not consider streaming data yet, but refer to conventional relational data. To further simplify the notation, we assume that the inputs of the BMJ operators are two relational tables R and S with the following schemata:

$$R : \{[x_1, \ldots, x_n, y_1, \ldots, y_d]\},\ S : \{[y_1, \ldots, y_d, z_1, \ldots, z_m]\}$$

With $r \in R$ and $s \in S$ the BMJ operators generate pairs of tuples $(r \times s)$. As mentioned in the introduction, these pairs shall match best according to special attributes, which are denoted as *join attributes*. The attributes $y_1, \ldots, y_d$ of R and S are used as join attributes. The attributes $x_1, \ldots, x_n$ and $z_1, \ldots z_m$ contain additional data and do not participate in the computation of best pairs. The terms "join attributes" and "dimensions" are treated as synonyms in the remainder of this thesis. Of course, all presented techniques are also applicable to other data models, e. g., XML data.

4.2.1 Comparing Pairs Using Partial Orders

For computing the best matching pairs of tuples all different pairs $(r \times s)$ and $(r' \times s')$ of $(R \times S)$ have to be compared. Thus, for every join attribute y_i ($i \in \{1, \ldots, d\}$) an order $\leq_{y_i}$ has to be defined on the elements of $(R \times S)$. $(r \times s) \leq_{y_i} (r' \times s')$ denotes the situation that $(r \times s)$ matches better than $(r' \times s')$ according to y_i. These orders represent the user-defined criteria. Which order shall be used on a specific join attribute depends on the intended application. Hence, the orders $\leq_{y_i}$ may be provided as "black-boxes", e. g., as user-defined functions, which get two pairs of tuples and return `true` in case of $\leq_{y_i}$, `false` in case of $>_{y_i}$, and `null` if the two pairs are incomparable with respect to dimension y_i. In general, $\leq_{y_i}$ may also be a partial order, which is relevant in practice, e. g., for comparisons on set-valued attributes. If the join attributes are of numerical type, an order could, e. g., be defined using the minimum distance of the join attributes.

Definition 4.2.1 (minAttrDist) *Let $r, r' \in R$ and $s, s' \in S$. Then, the order $\leq_{y_i}^{minAttrDist}$ on a join attribute y_i is defined as*

$$(r \times s) \leq_{y_i}^{minAttrDist} (r' \times s') \Leftrightarrow |r.y_i - s.y_i| \leq |r'.y_i - s'.y_i|.$$

An example for a partial order based on the comparison of set-valued attributes is, for instance, as follows.

Definition 4.2.2 (subset) *Let $r, r' \in R$ and $s, s' \in S$. Then, the order $\leq_{y_i}^{subset}$ on a join attribute y_i is defined as*

$$(r \times s) \leq_{y_i}^{subset} (r' \times s') \Leftrightarrow (r'.y_i \cap s'.y_i) \subseteq (r.y_i \cap s.y_i).$$

We distinguish two types of orders on the individual join attributes. *Type I* orders—such as *minAttrDist*—can be reduced to the order of real numbers using a function $f_i : dom(y_i) \times dom(y_i) \to \mathbb{R}$. With this, an order $\leq_{y_i}$ can be written as

$$(r \times s) \leq_{y_i} (r' \times s') \Leftrightarrow f_i(r.y_i, s.y_i) \leq f_i(r'.y_i, s'.y_i).$$

For instance, *minAttrDist* is defined by $f_i(a, b) := |a - b|$. *Type II* orders are arbitrary partial orders, e. g., the order *subset*. The methods presented in this chapter are applicable to both types. In real-world applications mainly orders of the first type are used and hence we focus on this type in the remainder of this work.

Using the individual orders on each join attribute, a single partial order $\prec_{y_1,\ldots,y_d}$ on the elements of $(R \times S)$ can be constructed as follows.

Definition 4.2.3 (Preference Order) *Let $\leq_{y_i}$ be a (partial) order on join attribute y_i $(1 \leq i \leq d)$, $r, r' \in R$, and $s, s' \in S$. Then, an order $\prec_{y_1,\ldots,y_d}$ representing user preferences on all dimensions is defined as*

$$\begin{aligned}(r \times s) \prec_{y_1,\ldots,y_d} (r' \times s') \Leftrightarrow\ & (r \times s) \leq_{y_1} (r' \times s') \ \wedge \cdots \wedge \\ & (r \times s) \leq_{y_d} (r' \times s') \ \wedge (r \times s) \neq (r' \times s').\end{aligned}$$

The inequality predicate in the last clause of this partial order prevents a pair from being compared to itself. $(r \times s) \prec_{y_1,\ldots,y_d} (r' \times s')$ denotes the situation that the tuples r and s match better than r' and s' according to all join attributes; this situation is called $(r \times s)$ *dominates* $(r' \times s')$. Since multiple dimensions are involved in the comparison, two pairs may naturally be incomparable.

4.2.2 The Best-Match Join Operators

We now define the family of BMJ operators on the basis of partial orders for comparing elements of $(R \times S)$ defined in Definition 4.2.3.

4.2.2.1 Definition of the Best-Match Join Operators

The task of the BMJ operator is to find all best matching elements of $(R \times S)$ according to a given partial order $\prec_{y_1,\ldots,y_d}$. That is, all elements of $(R \times S)$ have to be found which are not dominated by any other pair in $(R \times S)$.

Definition 4.2.4 (Best-Match Join) *Let $\prec_{y_1,\ldots,y_d}$ be a partial order reflecting users preference as defined in Definition 4.2.3, $r, r' \in R$, and $s, s' \in S$. Then, the BMJ of R and S, symbolized by $R \bowtie_{y_1,\ldots,y_d} S$, is defined as*

$$R \bowtie_{y_1,\ldots,y_d} S := \big\{(r \times s) \in (R \times S) \big| \neg\exists (r' \times s') \in (R \times S) : (r' \times s') \prec_{y_1,\ldots,y_d} (r \times s)\big\}.$$

Since $\prec_{y_1,\ldots,y_d}$ is a partial order, there may be several minima in $(R \times S)$ which are all incomparable against each other. Hence, the result generally consists of more than one pair. The BMJ computes all these minimal pairs. An alternative notation of the BMJ operator using an arbitrary (user-defined) partial order $\prec$ is $\tilde{\bowtie}_{\prec}$.

The BMJ-operator is not associative, i. e.,

$$(R \,\tilde{\bowtie}_{\prec}\, S) \,\tilde{\bowtie}_{\prec'}\, T \;\neq\; R \,\tilde{\bowtie}_{\prec}\, (S \,\tilde{\bowtie}_{\prec'}\, T).$$

It is important to note that the BMJ operator is different from the matching problem known in the context of graph theory. This problem is defined as finding a maximal subset of the edges—representing preferences for pairs of nodes—of a graph such that no edges of this subset share a single node as start and end point, respectively. In contrast to that, our BMJ operator computes the overall best matching pairs of objects, none of which being dominated by any other pair.

Of particular practical relevance are the outer-best-match join variants. The most interesting variant is the left-outer-BMJ. It does not only compute the overall best matching pairs, but produces all best matching pairs for each individual tuple of the left input.

Definition 4.2.5 (Left Outer Best-Match Join) *Let $\prec_{y_1,\ldots,y_d}$ be a partial order reflecting users preference as defined in Definition 4.2.3, $r \in R$, and $s, s' \in S$. Then, the left-outer-BMJ of R and S, symbolized by $R \,\tilde{⟕}_{y_1,\ldots,y_d}\, S$, is defined as*

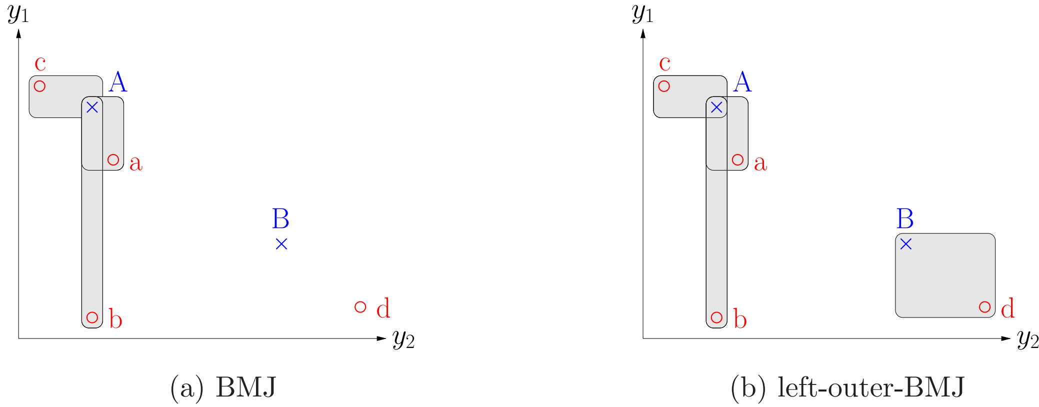
$$R \,\tilde{⟕}_{y_1,\ldots,y_d}\, S := \big\{(r \times s) \in (R \times S) \big| \neg\exists s' \in S : (r \times s') \prec_{y_1,\ldots,y_d} (r \times s)\big\}.$$

This left-outer-BMJ operator determines for every $r \in R$ the best-matching partners in S. Of course, the right-outer-BMJ is defined analogously. A last variant of the basic BMJ is the full-outer-BMJ. It computes for each left *and* right input tuple its best pairs.

Definition 4.2.6 (Full Outer Best-Match Join) *Let $\prec_{y_1,\ldots,y_d}$ be a partial order reflecting users preference as defined in Definition 4.2.3, $r, r' \in R$, and $s, s' \in S$. Then, the full-outer-BMJ of R and S, symbolized by $R \,\tilde{⟗}_{y_1,\ldots,y_d}\, S$, is defined as*

$$\begin{aligned} R \,\tilde{⟗}_{y_1,\ldots,y_d}\, S := \big\{(r \times s) \in (R \times S) \;\big|\; &(\neg\exists s' \in S :\; (r \times s') \prec_{y_1,\ldots,y_d} (r \times s)) \vee \\ &(\neg\exists r' \in R :\; (r' \times s) \prec_{y_1,\ldots,y_d} (r \times s))\big\}. \end{aligned}$$

Let us show by means of the following two abstract examples how the BMJ and the left-outer-BMJ variants work.



(a) BMJ (b) left-outer-BMJ

Figure 4.1: Examples of Best-Match Join Computations

Example 4.2.7 *Figure 4.1 (a) depicts the result of the BMJ* $R \bowtie_{y_1,\ldots,y_d} S$*. It shows two datasets of two-dimensional points, which are symbolized by crosses (representing* R*) and circles (representing* S*). The two dimensions are used as join attributes with minAttrDist as order on each dimension. The three pairs marked by boxes in Figure 4.1 (a) are the result of the BMJ* $R \bowtie_{y_1,y_2} S$*. These pairs are incomparable against each other, because the join pair* (A, a) *is better than* (A, c) *with respect to join attribute* y_2 *and* (A, c) *is better than* (A, a) *with respect to join attribute* y_1*. Similar thoughts hold for* (A, b)*. All other pairs are dominated by one of these three pairs, e. g.,* (B, d) *is dominated by* (A, a)*, because the distances of the points of both pairs with respect to join attribute* y_1 *are equal, but* (A, a) *is better than* (B, d) *with respect to join attribute* y_2*. This result can be achieved by, for instance, executing the SQL query:*

select $*$
from $R\ r,\ S\ s$
where **not exists** (
  select $*$
  from $R\ r',\ S\ s'$
  where $abs(r'.y_1 - s'.y_1) <= abs(r.y_1 - s.y_1)$ **and**
  $abs(r'.y_2 - s'.y_2) <= abs(r.y_2 - s.y_2)$ **and**
  $(r.y_1 <> r'.y_1$ **or** $r.y_2 <> r'.y_2$ **or** $s.y_1 <> s'.y_1$ **or** $s.y_2 <> s'.y_2))$;

□

Example 4.2.8 *Figure 4.1 (b) shows the result of the left-outer-BMJ* $R \bowtie_{y_1,y_2} S$ *on the same data. For the first tuple* A *of* R *the same conditions hold as before. Hence, the same pairs* (A, a)*,* (A, c)*, and* (A, b) *are generated. Next, the data point* B *is examined. The pair* (B, d) *dominates all other pairs with* B *as the first point and thus it is contained in the result. There are no more tuples in* R*, so the result of the left-outer-BMJ contains the marked pairs. Analogously to the previous example, these results can, for instance, be computed using the SQL query:*

```
select    *
from      R as r, S as s
where     not exists (
  select    *
  from      S as s'
  where     abs(r.y1 - s'.y1) <= abs(r.y1 - s.y1) and
            abs(r.y2 - s'.y2) <= abs(r.y2 - s.y2) and
            s.y1 <> s'.y1 and s.y2 <> s'.y2);
```

□

4.2.2.2 Query Language Integration

In this section, we propose some extensions to common query languages for being able to conveniently specify BMJ computations.

In order to specify BMJ operators in SQL queries, we propose to extend SQL's **from**-clause as follows:

```
select   ...
from     R [ ( left | right | full ) outer ] bestmatch join S
           on order1, order2, ...
where    ...
```

order1, *order2*, ... denote the user-defined comparison functions to be used as an order on the particular join attributes which are then composed to obtain the entire partial order for comparing pairs. There are two possibilities for specifying these individual orders. Type I orders using functions f_i that can be formulated in SQL are conveniently expressed as:

$$(f_i(R.y_i, S.y_i)) \text{ [} \textbf{min} \mid \textbf{max} \text{]}$$

This specification is translated into an order

$$(r \times s) \leq_{y_i} (r' \times s') \Leftrightarrow f_i(r.y_i, s.y_i) \leq f_i(r'.y_i, s'.y_i)$$

in the case of "**min**" and in

$$(r \times s) \leq_{y_i} (r' \times s') \Leftrightarrow f_i(r.y_i, s.y_i) \geq f_i(r'.y_i, s'.y_i)$$

in the case of "**max**".

Example 4.2.9 *The order minAttrDist on an attribute y_1 is specified by:*

$$(abs(R.y_1 - S.y_1)) \ \textbf{min}$$

□

Type I orders using functions that cannot be directly formulated in SQL and Type II orders, e. g., *subset*, are handled similarly by specifying the join attributes and the name of the user-defined comparison function:

$$(R.y_i, S.y_i)\ \mathit{function}$$

The difference of the variants is as follows: In the former, the computational logic is completely specified in SQL inside the parenthesis. The comparison operation, i. e., $\geq$ or $\leq$, is specified using the **min** and **max** keyword, respectively. In the latter, only the join attributes are specified in the parenthesis. The whole computation and comparison of these join attributes is done inside the code of the comparison function.

Example 4.2.10 *A best-match join* $R \bowtie_{y_1,y_2} S$ *using minAttrDist on* y_1 *and subset on* y_2 *(*y_2 *being a set-valued attribute) is written as:*

select ...
from R **bestmatch join** S
on (abs($R.y_1 - S.y_1$)) **min**, ($R.y_2$, $S.y_2$) **subset**;

□

Note that it has to be distinguished between the number of parameters of an order in our SQL syntax and the number of parameters of the user-defined comparison functions. In our SQL syntax, an order is specified by means of two (or at least one) join attribute. The implemented comparison functions always assume four parameters, where two at a time constitute one pair. For instance, these comparison functions could have the following signature in Java:

```
public int compareTo(Object r, Object s, Object r_prime, Object s_prime);
```

Unused parameters either refer to the same data object, e. g., in the case of a left-outer-BMJ "`r = r_prime`", or are `null`.

We propose a similar syntax to that of our SQL extensions for specifying BMJ computations in XQuery as follows.

for $\$r$ **in** π_R
for $\$s$ **in** π_S
where $\$r$ [(**left** | **right** | **full**) **outer**] **bestmatch join** $\$s$
on *order1*, *order2*, ...
return ...

As before, *order1*, *order2*, ... denote the user-defined comparison functions to be used as an order on the particular join attributes which are then composed to obtain the entire partial order for comparing pairs.

Example 4.2.11 *Consider again the example given in Example 4.2.10. Here, let* $\$r$ *and* $\$s$ *be the left and right inputs defined by the paths* π_R *and* π_S*, respectively, on which the best-match join shall be computed. Further,* y_1 *shall be reachable by the path* $\$r/\pi_{y_1}$ *in* R *and by* $\$s/\pi'_{y_1}$ *in* S*, respectively. Analogously,* y_2 *is reachable by the paths* $\$r/\pi_{y_2}$ *and* $\$s/\pi'_{y_2}$*, respectively. Then,* $R \bowtie_{y_1,y_2} S$ *using minAttrDist on* y_1 *and subset on* y_2 *is phrased in XQuery using our syntactical extensions as follows:*

> **for** $\$r$ **in** π_R
>   **for** $\$s$ **in** π_S
>   **where** $\$r$ **bestmatch join** $\$s$
>   **on** (**fn:abs**($\$r/\pi_{y_1}$ - $\$s/\pi'_{y_1}$)) **min**, ($\$r/\pi_{y_2}$, $\$s/\pi'_{y_2}$) **subset**
>   **return** ...

Note that the paths $\$r/\pi_{y_1}$*,* $\$s/\pi'_{y_1}$*,* $\$r/\pi_{y_2}$*, and* $\$s/\pi'_{y_2}$ *refer to the pair of elements currently being processed in the two nested* **for***-loops. This pair qualifies, if, for instance, the distance in the first dimension, i. e.,* **fn:abs***(*$\$r/\pi_{y_1}$ - $\$s/\pi'_{y_1}$*), is minimal, which is denoted by the* **min** *keyword. The computation of this minimum is hidden inside the* **bestmatch join** *operator. Also note that the comparison function* **subset** *has to be defined in XQuery or as an external function as described above in the case of SQL.* □

4.2.2.3 Basic Evaluation Strategy

Obviously, there is a close relationship between the BMJ operator and the skyline operator introduced in [BKS01]. The skyline of a single set of points (objects) is defined as those points which are not dominated by any other points of the set. Using the above defined partial order $\prec_{y_1,\dots,y_d}$, the BMJ can be ascribed to the computation of a skyline of the set $(R \times S)$ as

$$R \bowtie_{y_1,\dots,y_d} S = skyline_{\prec_{y_1,\dots,y_d}}(R \times S).$$

Therefore, a first approach for computing the BMJ consists of first materializing the Cartesian product $(R \times S)$ and then computing the skyline. Theoretically, by changing the original order $\prec_{y_1,\dots,y_d}$ to

$$(r \times s) \prec'_{y_1,\dots,y_d} (r' \times s') \Leftrightarrow (r = r') \wedge ((r \times s) \prec_{y_1,\dots,y_d} (r' \times s'))$$

the left-outer-BMJ can also be ascribed to the skyline computation as

$$R \ltimes\!\!\!\bowtie_{y_1,\dots,y_d} S = skyline_{\prec'_{y_1,\dots,y_d}}(R \times S).$$

Similarly, the other outer-variants can be handled.

In practice, not a single skyline is computed to get the results of the outer-variants, but for each left (or right) tuple a skyline is computed. Changing the partial order as described above exactly leads to this behavior. However, the approach of adapting the partial order for computing the outer-variants is not relevant in practice, since it has a

Figure 4.2: Nested-Loops Left-Outer-BMJ

Input : R, S, partial order $\prec_{y_1,\ldots,y_d}$
Output: $R \;⟖_{y_1,\ldots,y_d}\; S$

```
1 foreach r ∈ R do
2     foreach s ∈ S do
3         dominated ← false;
4         foreach s′ ∈ S do
5             if (r × s′) ≺_{y1,...,yd} (r × s) then dominated ← true;
6         endforeach
7         if ¬dominated then Output (r × s);
8     endforeach
9 endforeach
```

higher complexity than, e. g., the simple nested-loops approach for computing the left-outer-BMJ shown in Figure 4.2[2]. The algorithms of the other outer-BMJ variants can be constructed analogously. Obviously, ascribing the computation of BMJ operators to the computation of a skyline is rather inefficient because of the Cartesian product to combine the two argument sets and cannot be applied on streaming data, as it is impossible to materialize the Cartesian product.

4.2.3 Constrained Best-Match Joins

Figure 4.1 shows that "extreme pairs" such as (A, b) may be contained in the result of best matching pairs. This pair matches very well in one dimension, i. e., y_2, but poorly in the other dimension, i. e., y_1. This behavior is often not desirable in practice, because a single well-matching dimension might not balance the poor matching in other dimensions. Therefore, we propose constraints such that only pairs $(r \times s) \in (R \times S)$ are considered which do not exceed an individual maximum distance in each constrained dimension. This maximum distance on a dimension y_i, which is required for a pair to be considered as a part of the result of a constrained BMJ, is denoted as ϵ_i. The distance of two tuples r and s in dimension y_i is symbolized by $d_i(r, s)$. The situation that $d_i(r, s) \leq \epsilon_i$ for every $i \in \{1, \ldots, d\}$ is denoted as $d(r, s) \leq \epsilon$ (ϵ symbolizes the vector of the individual ϵ_i). So, only those $(r \times s) \in (R \times S)$ have to be considered as candidates for best matching pairs, for which $d(r, s) \leq \epsilon$ holds. With this, the constrained BMJ operator, denoted as $⋈^{\epsilon}_{y_1,\ldots,y_d}$, is formally defined as follows.

Definition 4.2.12 (Constrained Best-Match Join) *Let $\prec_{y_1,\ldots,y_d}$ be a partial order reflecting users preference as defined in Definition 4.2.3, ϵ a vector of constraints, d_i the distance metrics for each dimension, $r, r' \in R$, and $s, s' \in S$. Then, the constrained BMJ*

[2]It can easily be seen that the first approach has a complexity of $\mathcal{O}(n^4)$, whereas the algorithm in Figure 4.2 has a complexity of $\mathcal{O}(n^3)$.

of R and S, symbolized by $R \bowtie^{\epsilon}_{y_1,\ldots,y_d} S$*, is defined as*

$$R \bowtie^{\epsilon}_{y_1,\ldots,y_d} S := \big\{(r \times s) \in (R \times S) \;\big|\; d(r,s) \leq \epsilon \wedge \neg\exists(r' \times s') \in (R \times S) : \\ (d(r',s') \leq \epsilon \wedge (r' \times s') \prec_{y_1,\ldots,y_d} (r \times s))\big\}.$$

Analogously, the constrained left-outer-BMJ operator $⟕^{\epsilon}_{y_1,\ldots,y_d}$ is formalized in the following definition.

Definition 4.2.13 (Constrained Left Outer Best-Match Join) *Let* $\prec_{y_1,\ldots,y_d}$ *be a partial order reflecting users preference as defined in Definition 4.2.3,* ϵ *a vector of constraints,* d_i *the distance metrics for each dimension,* $r \in R$*, and* $s, s' \in S$*. Then, the constrained BMJ of R and S, symbolized by* $R \bowtie^{\epsilon}_{y_1,\ldots,y_d} S$*, is defined as*

$$R ⟕^{\epsilon}_{y_1,\ldots,y_d} S := \big\{(r \times s) \in (R \times S) \;\big|\; d(r,s) \leq \epsilon \wedge \neg\exists s' \in S : \\ (d(r,s') \leq \epsilon \wedge (r \times s') \prec_{y_1,\ldots,y_d} (r \times s))\big\}.$$

Of course, the constrained right-outer-BMJ and the constrained full-outer-BMJ are defined similarly. Constrained BMJ operators are expressed in SQL/XQuery using our new notation for the BMJ operators. To phrase a constrained BMJ, the maximum distance constraints are simply written after the corresponding comparison functions in the **on**-clause of the BMJ as follows

select ...
from R [(**left** | **right** | **full**) **outer**] **bestmatch join** S
  on *order1* ϵ_1, *order2* ϵ_2, ...
where ...

As before, the join attributes and the names of the user-defined comparison functions are specified after the **on**-keyword for each dimension. Additionally, for every join attribute the maximum distance is given by ϵ_i. Constraints are handled analogously in our extensions of the XQuery syntax.

The approach of computing the basic BMJ operators by ascribing them to the computation of one (or more) skyline(s) applies as well to the constrained BMJ operators. For instance, in the case of the constrained BMJ operator it has to be done as follows: Instead of using $R \times S$ as an intermediate relation, all pairs

$$\{(r \times s) \in (R \times S) \mid d(r,s) \leq \epsilon\}$$

are materialized. On this temporary relation an existing skyline algorithm can be applied as explained in Section 4.2.2 to determine the result of the constrained BMJ. The other variants of the constrained BMJ operator can be computed analogously by restricting the temporary relation to those pairs, which do not exceed the distance limits.

Example 4.2.14 (Running Example) *Using this definition of constrained best-match joins, the problem of the weather agency introduced in Example 4.1.3 can be solved on*

materialized data. The temperature and humidity data, respectively, shall be contained in the relations Temp and Hum, respectively. Each relation consists of the attributes t*,* x*,* y*, and* v *storing the time of measurement, the* x*- and* y*-coordinates of the sensor, and the sensor reading* v*. The attributes* t*,* x*, and* y *are used as join attributes. For comparing pairs of sensor data a partial order* $\prec_{t,x,y}$ *is constructed using minAttrDist on each dimension as an order. The given restrictions can be employed by setting* $\epsilon_t = 10min$*,* $\epsilon_x = \epsilon_y = 100m$*. Then, the expected pairs of sensor data can be obtained by evaluating the query*

$$Temp \ltimes^{\epsilon}_{t,x,y} Hum,$$

which can be written in our SQL extension as (we refrain from considering the correct units of the attributes):

```
select   *
from     Temp T constrained left outer bestmatch join Hum H
            on (abs(T.t - H.t)) min 10,
               (abs(T.x - H.x)) min 100,
               (abs(T.y - H.y)) min 100;
```

The result contains pairs of sensor data consisting of the best matching humidity measurements for each temperature sensor. If for a certain temperature sensor no humidity measurement satisfying the specified requirements is available, no pair having this particular sensor as a join partner will be contained in the result. To compute the results, the algorithm shown in Figure 4.2 can be employed. It has to be extended in a straightforward way to compare only those pairs satisfying the given maximum distance constraints.

Of course, the right-outer-BMJ could be executed as well to get for each humidity sensor the best matching temperatures. □

4.3 Evaluating Best-Match Joins on Data Streams

Naturally, Internet data sources provide their data as data streams. These data streams are either infinite or very huge. In both cases it is not feasible to buffer them locally and process the materialized data. Thus, methods for evaluating the constrained BMJ operators on streaming data are needed.

As already described in Section 3.8.2, a data stream R (or, S) is considered to be an ordered sequence of data objects $(r_1, r_2, \dots)$. The data objects can only be read sequentially, i. e., they are read in the sequence $r_1, r_2, \dots$ If r_i has been read, the data objects r_j with $j \leq i$ cannot be accessed anymore unless they are buffered locally. The problems arising from computing blocking operators, like the BMJ operators, on such data streams are discussed in the next section.

4.3.1 Best-Match Joins and Data Streams

To clarify the problems of evaluating BMJ operators on data streams we first consider the standard join operator. Generally, avoiding a blocking and enabling a pipelined execution is

achieved by utilizing suitable join algorithms, e. g., the double pipelined hash join [WA91]. That is, for every read input data object the join partners can be computed (on the basis of the inputs known up to that time) and delivered. To be able to compute the correct result, i. e., no join partners are missed, all consumed inputs have to be buffered. Hence, working on infinite data streams, a pipelined execution is possible, but the size of the state of a join operator grows beyond limits.

In contrast, the BMJ operators are in their basic form inherently blocking and hence not applicable on infinite data streams. This is due to the fact that temporary results may be invalidated by later arriving, better matching pairs. In the extreme, the best matching pairs of two (finite) data streams might be the last two data objects of the data streams. Therefore, in contrast to standard join operators, correct results cannot be delivered before the entire streams have been processed. If the BMJ operators are forced to deliver any results before, these results are approximative intermediate results. That is, they are correct with respect to the data processed up to that time, but some pairs might get dominated by newer (better matching) pairs and hence would be invalidated. Thus, dealing with infinite data streams, only approximative results can be continuously propagated. However, these approximative results are the best we can expect, since a “final” result does not exist. Like for the conventional join, the state of the BMJ operators in this case has an unlimited size, because all input data has to be buffered to be able to consider all pairs. The challenge in developing a BMJ algorithm that scales to infinite data streams is thus to deliver results as early as possible and to get by with a state of limited size.

We have introduced the constrained BMJ operators to enhance the quality of results by discarding non-interesting extreme pairs. In the worst case, these maximum distance constraints for pairs do neither bound the size of the state of the constrained BMJ operators, nor enable a pipelined execution. A best matching pair still might consist of data objects being delivered at the beginning of one and at the end of the other data stream. Again, the entire streams have to be buffered and processed to compute precise results. However, many data streams, e. g., sensor data, stock quotes, etc., do not deliver their data in a random fashion. Practically all real data streams consist of data objects which carry some kind of ordering information in special attributes, e. g., a time-stamp, sequence numbers, location neighborhood, etc. The data stream will naturally deliver its data objects sorted according to this attribute. In the remainder we focus on the data stream being strictly sorted in ascending order with respect to this attribute. We show techniques for relaxing this premise in Section 4.3.3. We refer to such an ordering of the data stream as a *physical property* and denote the special attribute as y_1 in the remainder. Furthermore, we assume that many data sources are somewhat “intelligent”, so that they can be told to deliver their data streams sorted according to a certain attribute y_1. In practice, y_1 is one of the constrained join attributes of a BMJ operation. Now, we can apply the constraint on y_1 as a *window predicate* to restrict the required input data to a small (contiguous) fraction of the data stream. Thus, the size of the state of the BMJ operator has an upper bound: For any data object $r \in R$ only those data objects $s \in S$ have to be buffered which are contained in the contiguous interval $r.y_1 - \epsilon_1 \leq s.y_1 \leq r.y_1 + \epsilon_1$ of the data stream S and vice versa (note that R and S are sorted according to y_1). These contiguous intervals

correspond to sliding windows of height ϵ_1 on the data streams and restrict the number of data objects that have to be buffered for computation.

In case of the practically more relevant outer-BMJ operators the combination of constraints and physical properties even enables a pipelined execution. Consider the left-outer-BMJ $R \ltimes^{\epsilon}_{y_1,\ldots,y_d} S$ as an example. For any data object $r \in R$ we are able to determine whether all interesting join partners $s \in S$ have been processed yet. This is the case, if all data objects s of the relevant contiguous interval of S have been read. As soon as this holds, the precise results for this r can be delivered without processing the remainder of the streams[3] and therefore a pipelined execution is achieved. This applies analogously for all outer-BMJ operators. The constrained BMJ is more complex, since it not only has to find all best matches for a single join partner, but the overall best matches. Thus, with respect to the pipelining behavior it cannot benefit from the combination of physical properties and constraints. However, approximative intermediate results, which converge to the precise result in case of finite data streams, can be delivered using size-limited sliding windows.

4.3.2 The Window-Based Approach

In the previous section, we have shown that it is possible to compute the constrained BMJ operators in a non-blocking fashion on data streams being sorted according to a constrained join attribute. The constraints define a multi-dimensional window of "interesting data" on the domain of a data stream. The idea of our algorithms is to "slide" this window along the sorted dimension over the data streams.

Our MatWin algorithm works as follows. We focus on the constrained left-outer-BMJ as an example in the following. Of course, the other variants (i. e., the constrained BMJ, right-outer-BMJ, and full-outer-BMJ) may be employed analogously. We further assume that both data streams are sorted according to y_1, that all distances of pairs $d_i(r, s)$ are normalized to the interval $[0; 1]$, and therewith all ϵ_i are within the interval $]0; 1]$. For the input data streams R and S, two data windows W_R and W_S are maintained and synchronously shifted over the data streams. Because each data stream is sorted according to y_1, it can be assured that all data objects belonging to a certain interval with respect to y_1 have been read from the data streams. The upper and lower bounds of this interval are denoted by $maxRS$ and $minRS$, respectively. $maxRS - minRS$ is denominated as the height of the data windows. $min_{y_1}(W_R)$ denotes the minimal y_1-value of all data objects contained in W_R. The data windows are maintained such that their height equals ϵ_1. Hence, the contents of the data windows given $minRS$ and $maxRS$ are

$$W_R = \{r_i \in R \mid minRS \leq r_i.y_1 \leq maxRS\},$$
$$W_S = \{s_j \in S \mid minRS \leq s_j.y_1 \leq maxRS\}.$$

During the computation the data windows are moved along the dimension y_1 by increasing $minRS$ and $maxRS$. That is, all data objects $r_i \in W_R$ with $r_i.y_1 < minRS'$ are deleted

[3]Of course, we continue to process the remainder of the stream to compute the left-outer-BMJ for all subsequent r', but this data is not relevant for r anymore.

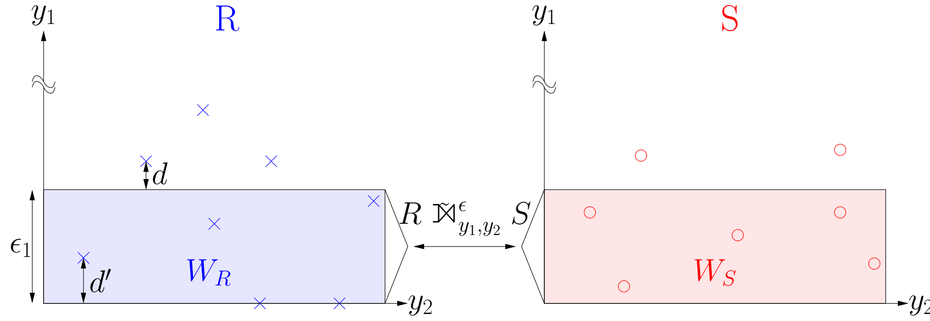


Figure 4.3: Constrained Left-Outer-BMJ: Initial Computation Step

from W_R and new data objects $r'_i \in R$ are read and stored in W_R as long as $r'_i.y_1 \leq maxRS'$, with $minRS'$ and $maxRS'$ being the new lower and upper bound of the data windows. Analogously, W_S is maintained. Thus, it can be assured that for a specific $r_i \in R$ all needed join partners $s_j \in S$ are contained in W_S at the same time: r_i enters W_R, when $r_i.y_1 = maxRS$. At this time W_S contains all $s_j \in S$, for which

$$r_i.y_1 - s_j.y_1 \leq \epsilon_1$$

holds. Analogously, r_i is deleted from W_R, if $r_i.y_1 < minRS$. Before that, W_S contained all $s_j \in S$ with

$$s_j.y_1 - r_i.y_1 \leq \epsilon_1.$$

Hence, all interesting join partners are contained in W_R and W_S while the data windows are shifted over the data streams.

The number of data objects contained in a data window is determined by ϵ_1 and the distribution of the values of join attribute y_1. This number may be potentially unlimited, but in real application domains the required size of the data windows will be limited. However, the maximum size of the data windows might be large and thus we might not be able to keep the data windows completely in main memory. All windowing techniques for joins described in the literature—to the best of our knowledge—limit the size of the data window to the size of the available main memory and discard the rest of the data objects. Hence, only approximative results can be computed, because interesting combinations of data objects might not be considered. In contrast to that, our approach materializes the data windows on secondary storage in fixed sized pages and employs efficient techniques for updating and processing these materialized data windows. Therefore, correct and complete results can be efficiently computed independently of the size of available main memory and the size of the data windows. We present efficient materialization and I/O-scheduling strategies for computing the constrained BMJ operators on these materialized data windows in Section 4.3.4.

Figures 4.3 to 4.5 show in detail how the MatWin algorithm computes the left-outer-BMJ $R ⟕^{\epsilon}_{y_1,y_2} S$. In these figures two join attributes y_1 and y_2 are used. The data streams are sorted according to y_1 and therefore the data objects are delivered from the bottom up.

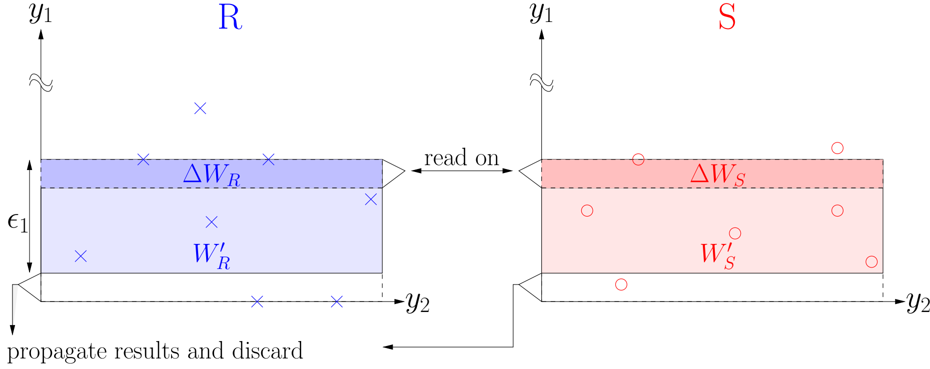


Figure 4.4: Constrained Left-Outer-BMJ: Moving Data Windows

Figure 4.3 shows the initial computation step. The initial data windows W_R and W_S are read (with $minRS = 0$, $maxRS = \epsilon_1$), materialized, and the left-outer-BMJ is computed on all pairs of $W_R \times W_S$ using the nested-loops algorithm shown in Figure 4.2. Not all needed pairs are processed at this time, so the computed best pairs are held in memory as an intermediate result O. Next, the data windows have to be moved. In a first step, all data objects $r_i \in W_R$ are removed, for which $r_i.y_1 = minRS$ holds. To calculate the new bounds of the data windows the two distances d and d' have to be considered. d is the distance with regard to y_1 of the first data object outside the current data window to $maxRS$. d' is defined as $d' = min_{y_1}(W_R) - minRS$. With this, $minRS$ and $maxRS$ are increased by $min(d, d')$ and the data windows are moved as described above. The new situation is depicted in Figure 4.4. The remaining parts of the data windows are denoted by W'_R and W'_S and the newly read parts are marked as ΔW_R and ΔW_S. Since all discarded data objects of the data window of R have been completely processed, all best matching pairs they are involved in can be propagated to the output data stream and deleted from O. The remaining current best pairs have to be updated with the pairs of $(W'_R \cup \Delta W_R) \times (W'_S \cup \Delta W_S)$. This is done by comparing the current best pairs O to pairs of the new data windows using $\prec_{y_1,\ldots,y_d}$. If a pair in O is dominated by a new pair, this new pair is inserted into O and the dominated pair is deleted. $(W'_R \cup \Delta W_R) \times (W'_S \cup \Delta W_S)$ can be rewritten as

$$(W'_R \times W'_S) \cup (W'_R \times \Delta W_S) \cup (\Delta W_R \times (W'_S \cup \Delta W_S)).$$

Obviously, not all pairs of the new data windows have to be compared with O, because the pairs of $W'_R \times W'_S$ have already been processed in the previous step. Hence, only the pairs of $W'_R \times \Delta W_S$ and $\Delta W_R \times (W'_S \cup \Delta W_S)$ have to be compared with all pairs contained in O. This situation is depicted in Figure 4.5. The movement of the data windows, the propagation of final results, and updating O continues until no more data can be read from the data streams.

It might seem obvious that the BMJ operators—being theoretically only an application of the skyline operation—could be computed by more sophisticated algorithms proposed in

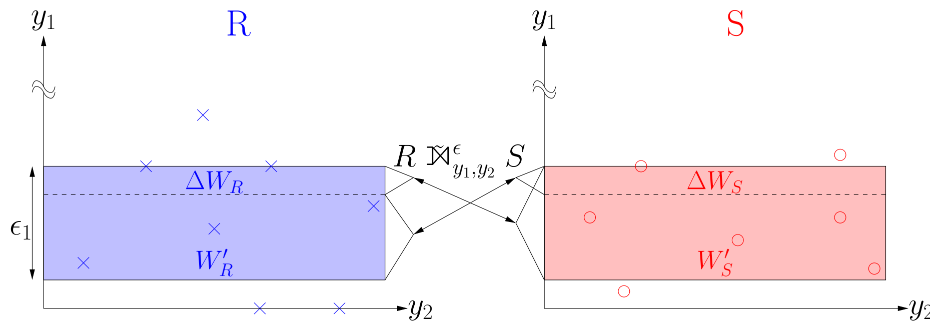


Figure 4.5: Constrained Left-Outer-BMJ: Updating Current Best Pairs

the literature, e. g., using the algorithms described in [BKS01, TEO01, KRR02, CGGL03]. But, the BMJ operators are intrinsically nested-loops operations because of having to compare *pairs* of data objects. While generating all pairs of data objects of both streams, the domination of the new pairs and the current best pairs can easily be checked as described above. Further, Kossmann et. al. [BKS01] have shown the nested-loops algorithm to be a good choice, since the number of best matching pairs is supposed to be small. All other algorithms are not applicable in the context of data streams: First, they work on indices of the entire data set, which cannot be computed on the data stream. Second, none of these algorithms is able to update an existing intermediate result by means of new data as it is needed here.

The details of the MatWin algorithm are shown in Figure 4.6. Data objects are read from the stream and materialized on secondary storage by the method MatWindows(. . .) (line 7). Its details are presented in Section 4.3.4.3. The method Update(. . .) used in lines 9, 11, and 12 updates the current best pairs of the left-outer-BMJ by the given parts of the data windows using a nested-loops algorithm as described in Section 4.2.2.

Of course, instead of the constrained left-outer-BMJ the other constrained BMJ operators can be computed by this method as well. However, in the case of the constrained BMJ, the propagated pairs are approximative as described in the previous section.

4.3.3 Exploiting Fuzzy Orders on Data Streams

The premise of the data streams being ordered is needed for the MatWin algorithm to be able to determine whether for a given upper bound of the data windows all necessary data objects have been read. However, some data sources may not be able to produce a strictly ordered data stream but only a "somewhat ordered" data stream. Consider a sensor network managed by our StreamGlobe DSMS, which multiplexes data from individual sensors into a single stream, e. g., the sensor data of individual temperature and humidity sensors of our running example. It might not be able to deliver all sensor readings ordered strictly according to the time of measurement, because the connections of the individual sensors have variable latencies. Hence, on the whole the timestamps of the sensor readings increase, but some data objects may be slightly out of order. But, StreamGlobe might be able to

Figure 4.6: MatWin Algorithm

Input : data streams R and S; $\epsilon_1, \ldots, \epsilon_d$; $\prec_{y_1,\ldots,y_d}$
Output: output data stream of $R \bowtie^{\epsilon}_{y_1,\ldots,y_d} S$

```
1  O ← ∅; oldMaxRS ← 0 ;
2  W_R ← ∅; W_S ← ∅ ;
3  r ← R.next(); s ← S.next() ;
4  minRS ← max(0, r.y1 − ϵ1) ;
5  maxRS ← minRS + ϵ1 ;
6  while (r ≠ null) ∨ (W_R ≠ ∅) do
7      {W_R, W_S} ← MatWindows(R, S, r, s, minRS, maxRS);
8      if oldMaxRS = 0 then
9          Update(O, W_R, W_S, ≺_{y1,...,yd});
10     else
11         Update(O, W'_R, ΔW_S, ≺_{y1,...,yd});
12         Update(O, ΔW_R, W'_S ∪ ΔW_S, ≺_{y1,...,yd});
13     endif
14     oldMaxRS ← maxRS ;
15     W_R.delete({r' ∈ W_R | r'.y1 = minRS}) ;
16     if (r = null) ∧ (W_R = ∅) then
17         break ;
18     else if (r = null) ∧ (W_R ≠ ∅) then
19         minRS ← min_{y1}(W_R) ;
20     else if (r ≠ null) ∧ (W_R ≠ ∅) then
21         minRS ← min(min_{y1}(W_R), r.y1 − ϵ1) ;
22     else
23         minRS ← r.y1 − ϵ1 ;
24         oldMaxRS ← 0;
25     endif
26     maxRS ← minRS + ϵ1;
27     W_S.delete({s' ∈ W_S | s'.y1 < minRS});
28     output and delete {(r' × s') ∈ O | r'.y1 < minRS};
29 endwhile
30 output {(r × s) ∈ O};
```

assure that, e. g., all measurements up to $t - 5s$ have been delivered at time t. We denote data streams satisfying such relaxed ordering properties as *fuzzy ordered data streams*. The MatWin algorithm can easily be adapted to work with such relaxed physical properties, because it still can be determined if all needed data has been read from a data stream.

We distinguish two types of constraints for fuzzy ordered data streams: *static* and *dynamic constraints*. A static constraint is a fixed property of a data stream known in

advance. For instance, static constraints are:

- **Value constraint.** Let r_i be the last read data object of a data stream R and let $max_{y_1} := \max\{r_j.y_1 \mid 1 \leq j \leq i\}$. Then, this constraint assures that all data objects r' with $r'.y_1 \leq max_{y_1} - c$ have been delivered up to the current point of time, with c being a characteristic constant of a given data stream.
- **Read-ahead constraint.** The read-ahead constraint defines the number c of data objects which have to be read from the data stream until it can be determined that all needed data objects have been read yet. For instance, if we want to read all data objects r with $r.y_1 \leq maxRS$, we have to read until c consecutive data objects with y_1-values greater than $maxRS$ have been encountered.

In addition, dynamic constraints can be delivered by the data source in the data stream as metadata interspersed with the data objects. An example for such a dynamic constraint is the concept of punctuations of data streams proposed by Tucker et. al. [TMSF03]. A punctuation is a predicate, which is delivered in the data stream like a normal data object. It means that after a punctuation has been read, no data objects will be contained in the data stream which satisfy that predicate. For instance, after the punctuation "$y_1 \leq c$" has been read, no more data objects with y_1-values less than or equal to c are delivered. Punctuations of a data stream can be arbitrary predicates on any attributes. For our purpose, punctuations concerning the join attribute y_1 in the form of "$y_1 \leq c$" are useful.

4.3.4 I/O-Scheduling Using the ϵ-Grid-Order

In our approach, we buffer data windows that do not fit into main memory on secondary storage. For an efficient computation of the constrained BMJs on the materialized data windows under main memory limitations we adapt the Epsilon Grid Order developed by Böhm et. al. [BK01]. It has been designed to efficiently compute the similarity join

$$\{(p \times q) \in (P \times Q) \mid ||p - q|| \leq D\}$$

of two data sets P and Q given a maximum distance D of the pairs of points. To suit our needs, we extend this order to consider a maximum distance ϵ_i on each dimension instead of using a single distance D. We denote our extended version of the Epsilon Grid Order as ϵ-Grid-Order. Further, the scheduling algorithms of [BK01] exploiting the Epsilon Grid Order are restricted to the similarity *self*-join; we adapted them to work on two inputs instead of one. Note that the ϵ-Grid-Order is used as an indexing technique for buffering the data windows on secondary storage. The order in which data objects arrive on the data stream is independent from the ϵ-Grid-Order, which only affects the materialized data windows.

4.3.4.1 Definition and Properties of the ϵ-Grid-Order

In this section, we define the ϵ-Grid-Order and show an important property for utilizing it in the context of constrained BMJ computations. For presentation purposes, we define the

ϵ-Grid-Order, symbolized by $<_\epsilon$, and present its properties using d-dimensional vectors p and q. The dimensions of these vectors correspond to the join attributes of a BMJ.

Definition 4.3.1 (ϵ-Grid-Order) *Let p and q be d-dimensional vectors. The predicate $p <_\epsilon q$ is true if there exists a dimension i such that the following condition holds:*

$$\left(\left\lfloor\frac{q_i}{\epsilon_i}\right\rfloor < \left\lfloor\frac{p_i}{\epsilon_i}\right\rfloor\right) \quad \wedge \quad \left(\left\lfloor\frac{q_j}{\epsilon_j}\right\rfloor = \left\lfloor\frac{p_j}{\epsilon_j}\right\rfloor \quad \forall j < i\right)$$

The i-th dimension of the vectors p and q are denoted by p_i and q_i.

Böhm et. al. show that the original Epsilon Grid Order is an irreflexive order. The proofs are similar for the ϵ-Grid-Order and are therefore not carried out here. The ϵ-Grid-Order divides the d-dimensional space into a grid. The edges of every cell of this grid have a length of ϵ_i in the dimension i. The data objects contained in each cell are equal with regard to $<_\epsilon$.

The following properties of the ϵ-Grid-Order enable an efficient computation of constrained BMJ operators. Let p, p', q be d-dimensional vectors. Given a fixed p, the following condition holds: If $q <_\epsilon p - (\epsilon_1, \ldots, \epsilon_d)$, there exists a dimension $i \in \{1, \ldots, d\}$ with $p_i - q_i > \epsilon_i$ and therewith

$$q \notin [p_1 - \epsilon_1] \times \cdots \times [p_d - \epsilon_d].$$

Additionally, for all p' with $p <_\epsilon p'$ there exists a dimension $j \in \{1, \ldots, d\}$ with $p'_j - q_j > \epsilon_j$ and therewith

$$q \notin [p'_1 - \epsilon_1] \times \cdots \times [p'_d - \epsilon_d].$$

Hence, a point q cannot be a join partner of p if $q <_\epsilon p - (\epsilon_1, \ldots, \epsilon_d)$. Analogously, it can be derived that a point q cannot be a join partner of p if $p + (\epsilon_1, \ldots, \epsilon_d) <_\epsilon q$.

These two properties can be utilized for an efficient computation of $R \bowtie^{\epsilon}_{y_1,\ldots,y_d} S$ (and the other variants) as follows. We assume that S is sorted on the join attributes according to the ϵ-Grid-Order. We define

$$r - \epsilon := [r.y_1 - \epsilon_1, \ldots, r.y_d - \epsilon_d, r.z_1, \ldots, r.z_m]$$

and analogously $r + \epsilon$. For a specific $r \in R$ only those $s \in S$ are interesting join partners which are contained in the interval

$$[r - \epsilon,\ r + \epsilon]$$

of the sequence of S sorted according to $<_\epsilon$. Hence, not all pairs $\{(r \times s) \mid s \in S\}$ have to be examined for computing the best matching pairs.

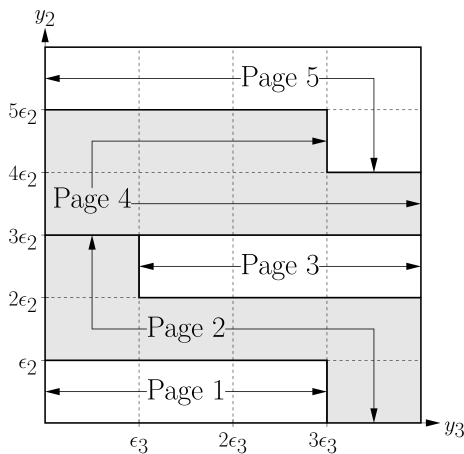


Figure 4.7: Pages in the Data Space

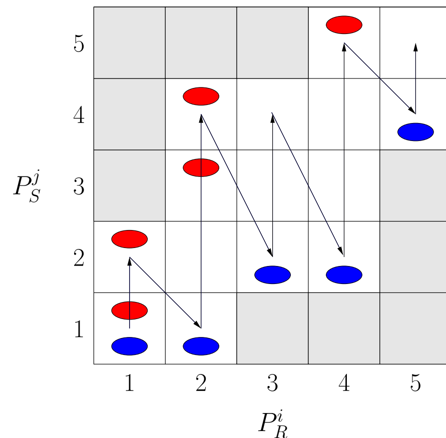


Figure 4.8: Pairs of Pages to be Processed

4.3.4.2 I/O-Scheduling

As mentioned before, the current data windows W_R and W_S are stored in pages on secondary storage by our MatWin algorithm. The pages of W_R and W_S are denoted as P_R^i and P_S^j, respectively, where i and j range between 1 and the maximum number of pages of W_R and W_S. To be able to exploit the benefits of the ϵ-Grid-Order these data windows have to be stored sorted according to $<_\epsilon$. That is, every page contains a sorted sequence of data objects. Further, the last data object of a page P_R^i has to be less than or equal to the first data object of any other page $P_R^{i'}$ with $i < i'$ (analogously for the pages of W_S). Two approaches for storing the data objects in that manner are presented in the next section. The pages of W_R and W_S have to be loaded into main memory to perform the computation of the constrained BMJ operators. For that purpose a buffer for m pages shall be available. The task of the scheduling algorithm is to load the pages of W_R and W_S into main memory during the computation of the constrained BMJ operators such that the number of disk accesses is minimized.

Figure 4.7 depicts an example of an assignment of data objects to pages with respect to the join attributes y_2 and y_3. The join attribute y_1 is omitted, because the height of the data windows is equal to ϵ_1. Hence, with regard to y_1 the data objects of the data windows are all equal or at most divided into two parts—if $minRS$ and $maxRS$ are not a multiple of ϵ_1—according to $<_\epsilon$. Therefore, to be able to present the assignment of data objects to pages in more detail we only show the join dimensions y_2 and y_3. Note that in the current situation of the data streams being sorted according to y_1 it would not be necessary to include y_1 in the ϵ-Grid-Order definition, because the sliding windows of the MatWin algorithm ensure that the constraints on y_1 are satisfied. To exploit physical properties that do not impose that the data streams are strictly sorted according to y_1 (as shown in Section 4.3.3), y_1 has to be included in the ϵ-Grid-Order definition. For presentation purposes we assume in the remainder that the layout of pages of both data windows equals

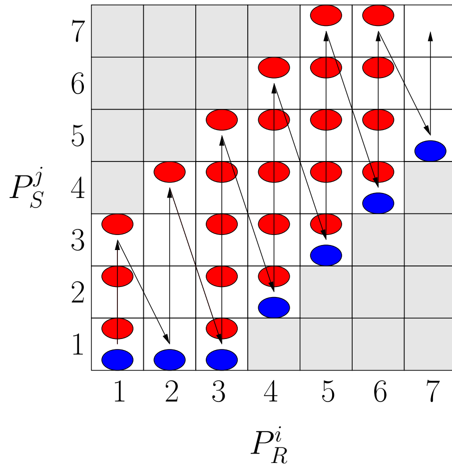


Figure 4.9: Gallop Mode: Thrashing

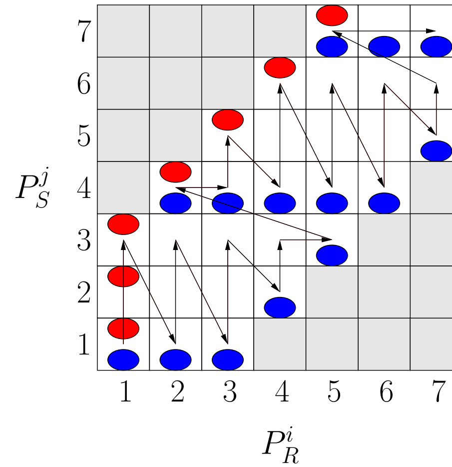


Figure 4.10: Crabstep Mode

that of Figure 4.7. Of course, in reality the assignment of data objects to pages might not be identical for W_R and W_S, which does not affect the algorithm.

To perform the update of the current best matching pairs O with new data windows the pages of W_R and W_S have to be loaded into the memory buffers and all interesting pairs have to be examined. Because of the observations of the previous section, not all pairs of pages have to be considered for computing the constrained BMJ. Figure 4.8 shows which pairs of pages have to be considered using the page layout of Figure 4.7. Each cell in the matrix stands for a pair of pages. Obviously, the pair (1,1), i. e., P_R^1 and P_S^1, has to be processed. The pair (1,2) has to be considered, because P_R^1 may contain data objects which have interesting join partners stored in P_S^2. The grey shaded cells need not to be considered. For instance, the pair (1,3) does not have to be processed, because P_S^3 cannot contain any interesting join partner s for any $r \in P_R^1$.

Now, all interesting pairs of pages have to be loaded into the main memory buffers. The goal is to induce a minimum number of disk accesses. Assuming $m = 4$ the naive column-by-column scheduling method is optimal. One buffer is reserved for storing the current page of W_R. The remaining $m-1$ buffers are available for loading pages of W_S. If $m-1$ pages of W_S are contained in the buffers and a new page of W_S has to be loaded, a page of W_S is discarded from memory using the LRU strategy. In the figure, a disk access of a page of W_R is symbolized by a blue oval and a disk access of a page of W_S is symbolized by a red oval. First, P_R^1 and P_S^1 are loaded and O is updated with all pairs of $P_R^1 \times P_S^1$. Next, P_S^2 is loaded and the pairs of $P_R^1 \times P_S^2$ are processed. No other pairs of pages in this column have to be processed. This can be determined by comparing the last and first data objects of the corresponding pages using $<_\epsilon$. The values of the join attributes of the first and last data object of a page can be held in main memory to avoid additional disk accesses. Proceeding to the next column is done by discarding P_R^1 and loading P_R^2. This scheduling is performed until all pairs of pages have been processed. In the style of [BK01] this scheduling method is called *gallop mode*.

Figure 4.9 depicts another example using $m = 4$ and the gallop mode. Up to column 2 this method is optimal. Then, thrashing occurs, because the number of pages needed for storing the whole interesting interval of data objects exceeds the number of memory buffers. In this case we switch to the *crabstep mode* (according to [BK01]) as shown in Figure 4.10. After processing the pair (2,3), P_S^4 has to be loaded, but there is no available buffer. Hence, the loaded pages of W_S are pinned to the buffers. The pages $P_R^3, \ldots, P_R^5$ are consecutively loaded into the single buffer for pages of W_R and processed. After (5,3) has been computed, all buffers are freed and the scheduling is continued at (2,4). It is again started with the gallop mode and switched to the crabstep mode if needed. Thus, the number of disk accesses can be greatly reduced—in this example from 30 to 21.

Another reason for utilizing this grid structure to store the data windows on secondary storage is that an efficient "lazy update" strategy can be used while sliding the materialized windows over the data streams. This lazy update works as follows. While shifting the data windows, data objects that are not needed any longer (i. e., data objects having an y_1-value less than $minRS$) are not explicitly deleted from the corresponding pages. Because of the fact that traversing all grid cells according to the ϵ-Grid-Order is done in a lexicographical fashion, a whole page can be discarded if the "maximum" data object of this page according to y_1 has a y_1-value less than $minRS$. Therefore, while shifting the data windows, no expensive deletions of individual data objects are performed, but whole pages are simply dropped if they are found not to be needed anymore. Hence, no additional sophisticated and possibly expensive "bulk-delete" strategies have to be employed for performing the updates of the materialized sliding windows.

As explained in Section 4.3.2, the MatWin algorithm does not have to process all pairs of $W_R \times W_S$ but only parts of the whole data windows, e. g., $\Delta W_R \times (W_S \cup \Delta W_S)$. The data objects of these parts, e. g., ΔW_R, are spread over all pages of the data windows. Of course, while processing a pair of pages the data objects not contained in these interesting parts of the data windows are ignored. To efficiently process only the interesting parts of the pages the following indexing technique can be applied. A page contains an ordered sequence of cells of the ϵ-Grid-Order. In every cell, all data objects are equal according to $<_\epsilon$ and hence the order in which the data objects arrive can be preserved in each cell (note that the stream is sorted according to y_1 and hence the data objects of each cell are ordered accordingly, too). In each page a list of references to the first data objects of all contained cells is stored. The first data object of a page having a y_1-value greater than a given boundary can be determined by performing a binary search over this list. Analogously, all data objects of a page with a y_1-value lower than a given boundary are processed by proceeding sequentially through each cell until a data object with a y_1-value greater than the limit is reached.

4.3.4.3 Materializing the Data Windows

In this section, we show how to materialize the current data windows W_R and W_S of the data streams R and S sorted according to $<_\epsilon$.

We start by again assuming that the data streams are strictly ordered with respect to y_1.

Figure 4.11: MatWindows Algorithm

```
Input  : data streams R and S; r, s, minRS, maxRS
Output: materialized windows W_R and W_S
1  while ((r.y_1 ≤ maxRS) ∧ (r ≠ null)) ∨ ((s.y_1 ≤ maxRS) ∧ (s ≠ null)) do
2      if (r.y_1 ≤ maxRS) ∧ (r ≠ null) then
3          if (|M_R| = m_R − 1) ∧ (last page in M_R is full) then
4              Materialize(M_R);
5              M_R ← ∅;
6          endif
7          M_R.add(r);
8          r ← R.next();
9      endif
10     if (s.y_1 ≤ maxRS) ∧ (s ≠ null) then
11         if s.y_1 ≥ minRS then
12             if (|M_S| = m_S − 1) ∧ (last page in M_S is full) then
13                 Materialize(M_S);
14                 M_S ← ∅;
15             endif
16             M_S.add(s);
17         endif
18         s ← S.next();
19     endif
20 endwhile
21 Materialize(M_R);
22 Materialize(M_S);
23 return {W_R, W_S};
```

The common part of both methods is depicted in Figure 4.11. The memory buffers, which were used in the previous section for computing/updating the constrained BMJ operators, are divided between the two input stream for sorting. For R and S, respectively, they are denoted by M_R and M_S. Further, m_R and m_S pages can be held in main memory ($m_R + m_S = m$). m_R and m_S may be dynamically adapted to improve the efficiency of the materialization, e. g., if W_R is completely materialized before W_S, the entire buffer is used to materialize W_S. The data objects of the data streams are read into the corresponding buffer. If the buffer of a data window is full or the data window contains all necessary data objects, e. g., W_R is completely filled if a r with $r.y_1 > maxRS$ has been read, the buffer is sorted (in memory) and written to disk using the Materialize()-method (lines 4, 13, 21, and 22). This is repeated until both data windows are completely materialized. The Materialize()-method implements one of the following two strategies.

The first materialization approach works like external sorting. If the memory buffers of a data stream are full, the pages are sorted and written to secondary storage as a run.

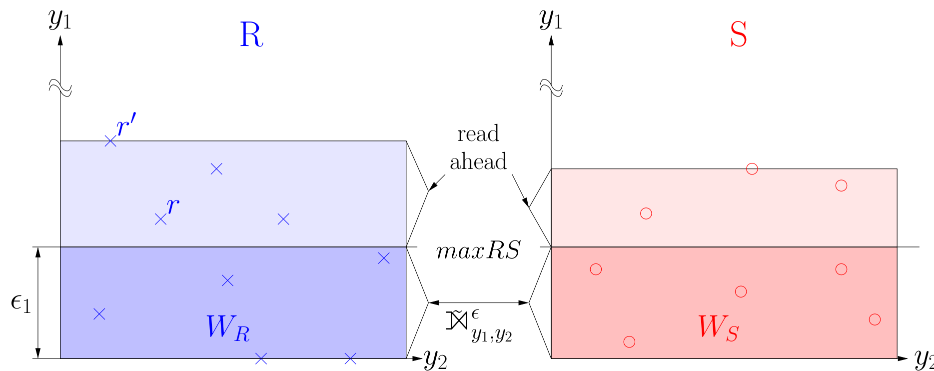


Figure 4.12: Filling Data Windows on Fuzzy Ordered Data Streams

Eventually, all created runs and the old data window (W'_R and W'_S, respectively) of the previous step are merged to create the current sorted data window. This approach is denoted as *standard materialization*.

The second approach, denoted as *grid materialization*, utilizes the fact that the ϵ-Grid-Order divides the data space into cells. This grid is held in memory as an index. Every cell of the grid contains a list of pages. If the memory buffers of a data stream are full, the data objects are sorted in memory. Using the grid structure, all data objects belonging to a single cell are materialized by reading the last page of the list of pages belonging to this cell and writing the data objects to it. If a page is full, a new one is appended to the corresponding list.

Both approaches have assets and drawbacks. The standard materialization has to create and merge the runs. Hence, all pages of the old data windows and the current runs are read and written. In contrast to that, the grid materialization has to read and write only the last existing page of each cell for inserting new data objects. Hence, the grid materialization will in most cases generate less disk accesses. But, the grid structure will grow rapidly with decreasing ϵ_i and an increasing number of dimensions. Additionally, the pages might be populated sparsely as for each cell a page is reserved.

Only minor changes to the MatWindows algorithm have to be made to accommodate it to fuzzy ordered data streams. The MatWin algorithm needs the y_1-value of the "next" data object outside the current data window for the computation of the new window bounds. Hence, the materialization algorithm has to continue reading data objects from the stream until this data object is definitely known. How many data objects have to be read ahead is determined by considering the given constraint of the fuzzy ordered stream. Figure 4.12 shows this situation for the example of the initial computation step of the MatWin algorithm. R and S shall be fuzzy ordered data streams with the static read-ahead constraint $c = 2$. While filling the data window W_R the data object r with $r.y_1 > maxRS$ is read. Because of the fuzzy ordered stream, the MatWindows()-method continues reading data objects until the read-ahead constraint is satisfied, i. e., until r' is known. Now, it is guaranteed that all necessary data objects for W_R have been delivered and r is the "next"

data object outside the data window. W_S is handled analogously. Dynamic constraints such as punctuations can be exploited similarly: For filling a data window with a given upper bound $maxRS$ the data stream is read until a punctuation "$y_1 \leq c$" with $c \geq maxRS$ is received. Then, all needed data objects are known and the computation can be done as already presented. Obviously, only the conditions in the lines 2 and 10 have to be adapted according to the given physical property. Of course, the data windows may contain data objects lying above the upper bound of the data windows. To avoid multiple processing of these data objects only the data objects within the upper and lower bound of the data windows have to be considered for computing the constrained BMJ operators.

Further, it might be possible that the data sources can be told to already deliver the data streams ordered according to the ϵ-Grid-Order. Obviously, such streams are fuzzy ordered data streams and therefore usable by the MatWin algorithm. In this case the sorting steps during the materialization of the data windows can be avoided and the MatWin algorithm becomes even more efficient.

4.3.5 Dealing with Time-Stamped Data Streams

In the previous sections, we have shown how the constrained BMJ operators can be efficiently processed on (fuzzy) ordered data streams. In this section, we analyze some example application scenarios for computing constrained BMJ operators on special data streams which we denote as *time-stamped data streams*. Time-stamped data streams are data streams where each data object has an associated time-stamp t and which are sorted, or at least fuzzy ordered, according to this dimension t. We distinguish two cases with regard to the computation of constrained BMJ operators on such streams: t is not used as a join attribute and t is a join attribute.

4.3.5.1 Time as a Non-Join-Attribute

Here, the most common task is to consider all input data given a particular maximum age, but the time-stamps of the data objects do not affect the computation of best matching pairs. For instance, the problem *"Compute $R \rtimes\!\!\!\ltimes^{\epsilon}_{y_1,\ldots,y_d} S$ of the data streams R and S using an order $\prec_{y_1,\ldots,y_d}$ and do not consider any input data older than T"* has to be solved.

For that purpose, we employ a simple window-based (not the MatWin algorithm) approach as shown in Figure 4.13, which is similar to window joins [KNV03, GÖ03]. For each data stream a data window is maintained such that for every point in time all previous data objects of period T are contained. The upper and lower bounds with regard to the time-stamps of the data objects of both data windows are equal. Every time a new data object with an associated time-stamp t is read from a data stream, all data objects with a time-stamp older than $t - T$ are discarded from both data windows. The data windows are either materialized, if they are too big for being stored in main memory, or held in main memory. Whenever the data windows change, $W_R \rtimes\!\!\!\ltimes^{\epsilon}_{y_1,\ldots,y_d} W_S$ is computed, the best matching pairs are annotated with the current time, and propagated to the output data stream. For the computation of the constrained left-outer BMJ operator any algorithm

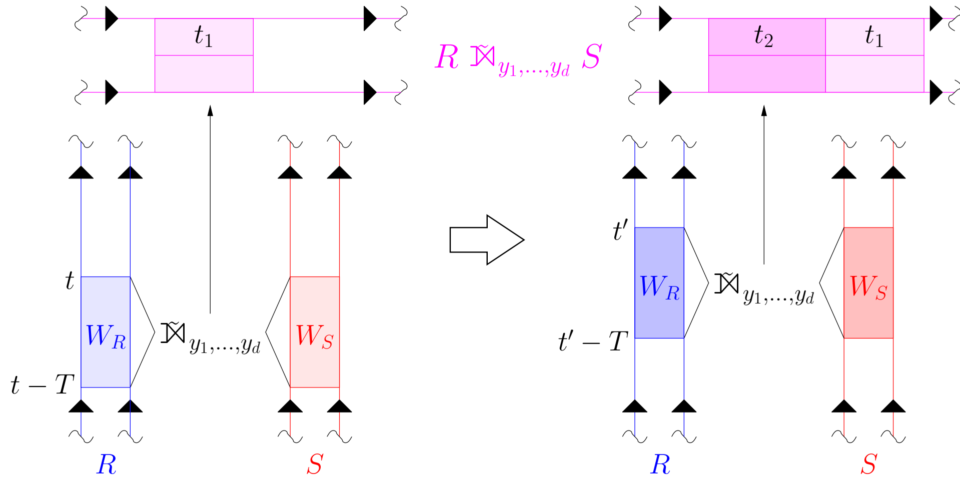


Figure 4.13: Fixed Time Window Evaluation

can be used, e. g., the nested-loops algorithm utilizing the ϵ-Grid-Order. We propose this re-evaluation approach, because an algorithm continuously updating the current result of best matching pairs does not produce satisfying results in this situation: Consider a very well matching pair $(r \times s)$ which dominates a pair $(r' \times s')$, where r' and s' are younger than both r and s. At a particular point in time, r, s, or both of them are discarded from the data windows, because they are too old. At this time the pair $(r' \times s')$ should be contained in the result, as $(r \times s)$ (which dominated $(r' \times s')$) is not known any longer. An approach which continuously updates the current best pairs will not generate the pair $(r' \times s')$ after $(r \times s)$ has been discarded, because $(r' \times s')$ already has been processed and it will not be considered again.

Of course, fuzzy ordered data streams can be handled analogously as explained in Section 4.3.3 for the MatWin algorithm.

4.3.5.2 Time as a Join-Attribute

Now, the time dimension is a join attribute and hence influences the computation of best matching pairs. We show how to handle such a situation by means of the following example: *"Compute for each data object of the data stream R all best matching join partners of the data stream S. Best matching pairs shall be those pairs which match best given a partial order $\prec_{y_1,\ldots,y_d}$ not considering the time dimension, as well as pairs being very young, even if they do not match perfectly according to $\prec_{y_1,\ldots,y_d}$".*

To solve this problem, we define the age of a pair $(r \times s)$ as

$$age(r,\ s) := max(r.t,\ s.t),$$

i. e., a pair is as old as the youngest join partner[4]. In combination with the given partial

[4]Depending on the application scenario, *min* might also be of interest. It can be dealt with analogously.

order $\prec_{y_1,\ldots,y_d}$ a single partial order $\prec$ for comparing pairs of $(R \times S)$ can be constructed as follows to solve our problem using a left-outer-BMJ operator:

$$(r \times s) \prec (r \times s') \Leftrightarrow (age(r,s) \geq age(r,s')) \;\wedge ((r \times s) \prec_{y_1,\ldots,y_d} (r \times s'))$$

Because of the definition of the age of a pair, a pair consisting of a young and a very old data object is considered to be young. Again, such pairs are not interesting in real-world applications, because in general we are only interested in young pairs consisting of data objects of roughly the same age. So, we only want to consider pairs $(r \times s)$ which differ no more than a given T in their age, i. e., to be an interesting pair the condition $|r.t - s.t| \leq T$ must hold. Since the data streams are (fuzzy) ordered according to the time-dimension and pairs to be considered have a given maximum distance T with regard to this dimension, the MatWin algorithm is applicable for efficiently computing this constrained left-outer-BMJ. Therefore, our problem is solved and the result contains all pairs for which no younger and better matching pair exists, as well as older pairs which match better than the younger pairs according to $\prec_{y_1,\ldots,y_d}$.

4.4 Performance Evaluation

In this section, we present the results of the most relevant benchmark experiments to assess our algorithms. All tests have been performed using the prototype Java (JDK 1.4) implementation in our StreamGlobe system. I/O operations are based on the `java.nio`-package. In our experiments, we have measured the query $R \ltimes\!\!\rtimes^{\epsilon}_{y_1,\ldots,y_d} S$ as an example. As input data we have used data streams of a simple relational schema

$$\big\{[data : string, y_1 : double, \ldots, y_d : double]\big\}$$

for both R and S. As before, $y_1, \ldots, y_d$ have been the join attributes of the constrained left-outer-BMJ. On each dimension the order *minAttrDist* has been utilized for comparison. The values of the join attributes have been randomly generated in the range $[0;1]$ using uniform, correlated, and anti-correlated distributions [BKS01] and a normal distribution.

All experiments have been carried out on a Sun Enterprise 450 with four 400 MHz processors and 4 GB of main memory. The implementation is not optimized for parallel processing, so only one processor has been utilized. In our tests, we have varied the size of the inputs, the number of dimensions, the distribution of the join attributes, and the maximum distances ϵ_i.

At first, we have measured the total execution time of the MatWin algorithm in comparison with the nested-loops algorithm to be able to judge the overall efficiency of the MatWin algorithm. We have used the 2-dimensional constrained left-outer-BMJ with $\epsilon_i = 0.1$, uniformly distributed values of the join attributes, and have varied the size of the input data streams. For the computation of the constrained left-outer-BMJ, the MatWin algorithm with/without utilizing the ϵ-Grid-Order using the standard materialization (MatWin/Standard, MatWin/Standard/EGO) and the grid materialization utilizing the ϵ-Grid-Order (MatWin/Grid/EGO) have been employed. The left diagram of

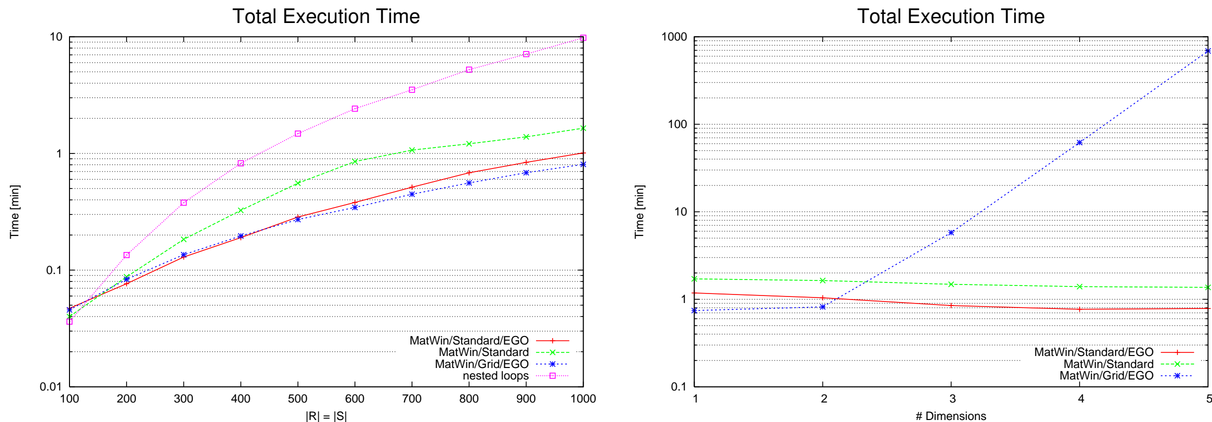


Figure 4.14: Overall Performance of the MatWin Algorithm

Figure 4.14 shows the results of this experiment. All measured variants of the MatWin algorithm perform significantly better than the nested-loops algorithm. Utilizing the ϵ-Grid-Order, the efficiency can be further increased. The grid materialization performs slightly better than the standard materialization. The right diagram of Figure 4.14 shows the results of the next experiment. In this test the three MatWin-variants have been measured using the same parameters as before and varying the number of dimensions. The size of the input has been fixed to 1000 data objects each. Obviously, the grid materialization performs better with less dimensions. With an increasing number of dimensions the growth of the grid prevails its benefits and it performs worse. The performance of the standard materialization is not affected by the number of dimensions.

Considering the overall efficiency, the MatWin/Standard/EGO algorithm outperforms the nested-loops algorithm. Other tests show, that these observations hold varying the distribution of the values of the join attributes and the maximum distances ϵ_i. Of course, the total time of execution can only be determined if the data streams are finite. But it has to be kept in mind that the nested-loops algorithm is not applicable to never-ending data streams and thus a comparison of the overall performance is only feasible in the case of finite data streams.

To determine the performance of the MatWin algorithm on infinite data streams, the following experiments investigate how continuously results are delivered. For that purpose, the constrained left-outer-BMJ has been computed on data streams of a fixed size of 1000 data objects each with two join attributes. The percentage of the result, which had been propagated at a certain time, has been measured. The remaining parameters have been equal to those mentioned above. The left diagram of Figure 4.15 depicts the results of this experiment varying the distribution of the values of the join attributes. It can be observed that the distribution of the join attributes has no influence on how continuously the results are propagated. The difference in total execution time is due to the fact that the result consists of more pairs using normally distributed join attributes. Thus, more pairs have to be compared during the update of intermediate results yielding a higher execution time.

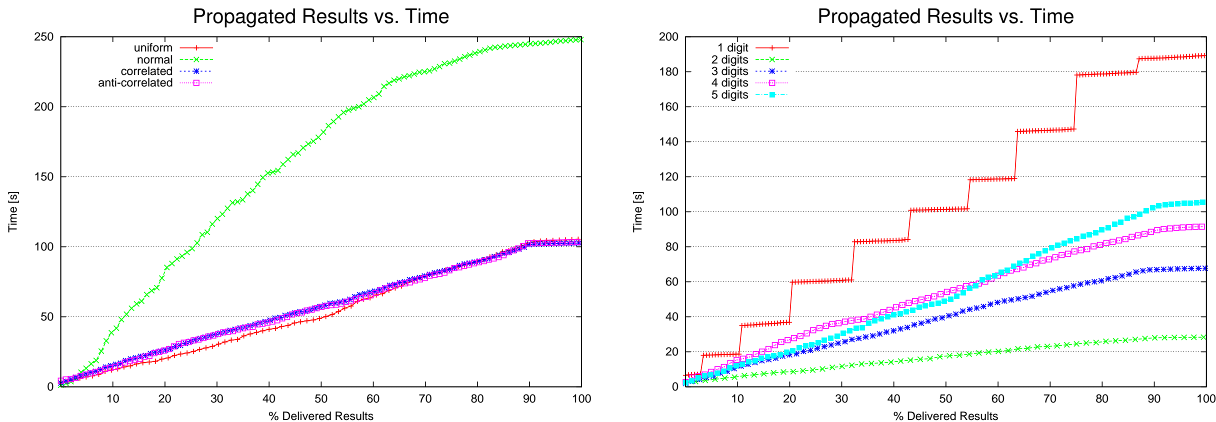


Figure 4.15: Streaming Behavior of the MatWin Algorithm

The right diagram of Figure 4.15 shows the results of the tests varying the number of distinct values of the join attribute y_1. The precision of the values of y_1 has been varied from one to five decimal places to investigate whether the granularity of y_1 influences the propagation of results. It can be observed that the granularity does not influence the continuous propagation of results except the measurement with one decimal place. In this case, the precision is equal to $\epsilon_1 = 0.1$. Hence, the data objects are always located exactly on the boundaries of the data windows. Whenever the data windows are moved, all data objects of the lower boundary are completely discarded and the results, they are involved in, are propagated all at once. This explains the steps in the measurement.

Concluding, it can be stated that the MatWin algorithm shows a good overall performance and pipelined behavior in all our experiments.

4.5 Related Work

Our work on best-match joins is related to three subareas of query evaluation: evaluation techniques for streaming data, window-based processing of joins, and finding optimal (best-matching) pairs of objects. We will focus on related work in these three areas in the remainder of this section.

Query Processing on Data Streams

We will only show some directly related work to our best-match joins in this paragraph. Most of the related work of this area has already been covered in Sections 2.5 and 3.10.

The streaming query engine of STREAM [ABB+03, MWA+03, BW01] processes data by mapping streams to relations using window constraints, applying relational operators, and converting the result back to streams. We employ a similar processing approach in our streaming algorithms.

Other relevant work is generating approximative intermediate results [RH02]. This is related to computing an unconstrained best-match join (i. e., all *overall* best matching pairs) on data streams, where approximate results are the best that can be expected.

Window-Based Join Processing

To enable pipelined BMJ processing of streaming data we have developed windowed BMJ processing techniques. Window-based processing of joins is one approach to overcome the fact that many join implementations are pipeline breakers. One noble exception is the double pipelined hash join [WA91]. It has been extended to the XJoin [UF99] enabling arbitrarily large buffers which are swapped to secondary storage. We employ a similar idea in our best-match join by buffering data windows on disk if they exceed main memory capacity. To overcome the fact that processing a join over unbounded data streams requires unbounded memory, in [KNV03] window joins are investigated. In [GÖ03] different algorithms and optimization techniques for efficiently processing sliding window multi-joins are described. Various processing strategies for shared window joins aiming at the optimization of the response time of all processed queries are proposed in [HFAE03]. The problem of approximating sliding window joins over data streams and quality measures of generated results is investigated in [DGR03]. In contrast to our work, these window join algorithms are all main memory techniques.

Computation of Best-Matching Pairs

Semantically, the variants of the best-match join operators conform to the skyline computation—albeit over the cross product of the arguments and therefore not being applicable on data streams. Finding the best data objects of a single set according to multi-dimensional criteria is the goal of much research effort. Kung et. al. [KLP75, PS85] describe the maximum vector problem to find all maximum ("best") vectors with respect to a partial order. Meanwhile, several other algorithms have been proposed for solving the maximum vector problem. However, these algorithms are all main memory techniques. [BKS01] was the first work to tackle this problem in a database context. The authors have introduced the skyline operator as an extension of the work of Kung et. al., working much better in the database field, proposed new algorithms, and showed how to integrate this operator into a database system. Since then, various algorithms have been devised for efficiently computing the skyline in a database environment, especially for computing the skyline progressively. That is, first skyline points are delivered before the entire data is read. This is accomplished by the use of various indexing techniques, e.g., sorting [CGGL03], bitmap indices and B-Trees [TEO01], and nearest neighbor search using R-Trees [KRR02, PTFS03, RKV95]. These algorithms are able to progressively deliver skyline points, but beforehand the entire data has to be processed in order to compute the required indices. Thus, these algorithms are not applicable in the context of data streams. Further, they only work on a single set as input data.

From a more theoretical point of view the work on preference queries, which are a generalization of the skyline problem, is also of interest. In [Kie02] Kießling presents a general framework for preference queries and a preference algebra using partial orders, which are also used in our work for specifying best matches. Chomicki [Cho03] studies properties of preference queries by means of a general framework using preference formulas and binary preference relations.

The area of similarity joins is also semantically related. In [BK01] efficient indexing techniques for the similarity self-join are presented. We have extended these techniques to be able to process two inputs and employ them for efficiently buffering data windows on secondary storage.

4.6 Discussion

In this chapter, we have introduced the novel class of best-match join operators and the practically more relevant family of constrained BMJ operators to compute best matching pairs of two data sets based on user-defined multi-dimensional criteria. The constraints in combination with certain physical properties of the data streams overcome the blocking nature of the BMJ operators and enable our new pipelined sliding-window BMJ algorithms to process infinite, (fuzzy) ordered data streams. These algorithms are based on synchronously sliding data windows over the data streams. To handle arbitrarily large data windows we have developed techniques for efficiently materializing the data windows on secondary storage. We have discussed efficient I/O-scheduling methods for processing the materialized data windows based on a particular order of the materialized data. Our experiments, which have been carried out within our Java-based StreamGlobe implementation, have shown a good overall performance and pipelining behavior of this approach.

Chapter 5

Conclusion

In this thesis, we have investigated three aspects of a general-purpose data stream management system: its design and common optimization goals, a streaming query engine for processing subscriptions, and the application of novel information retrieval operations on data streams.

We have presented the architecture of our distributed data stream management system StreamGlobe, which has been designed to meet the challenges that arise in processing data streams in an information retrieval network. StreamGlobe employs XML, being the de facto standard for information exchange, as the data model for data streams and, correspondingly, XQuery for specifying subscriptions. Further, StreamGlobe is built on top of P2P networking and Grid techniques to provide users the ability to flexibly join the network and to fit into existing e-science platforms. StreamGlobe differs from other upcoming data stream systems in not only efficiently locating and querying data streams, but at the same time optimizing the data flow in the network by means of data stream sharing. Data stream sharing is based on exploiting in-network query processing capabilities by pushing operators for subscription evaluation into the network. These optimizations, aiming at a parsimonious usage of both networking and computational resources, are a crucial requirement for scalable large-scale distributed data stream management systems.

For the evaluation of XQuery subscriptions, we have presented our streaming XQuery engine, FluX. Main memory is probably the most critical resource in (streamed) query processing. Keeping main memory consumption low is vital to scalability and has an impact on query engine performance in terms of runtime. We have introduced the FluX language as an extension of the XQuery language. It supports event-based query processing and provides a strong intuition for buffer-conscious query processing on structured data streams. Further, we have presented an algorithm for automatically translating a significant fragment of XQuery into equivalent FluX queries. This algorithm uses schema information to schedule FluX queries so as to reduce the use of buffers. We have described the core concepts of the prototype implementation of our query engine. As evidenced by our experiments, our approach indeed dramatically increases the scalability and performance of main memory XQuery engines.

StreamGlobe enables the execution of user-defined operators to carry out expressive

information retrieval tasks beyond the scope of subscriptions phrased in XQuery. As an example, we have introduced and formally defined the novel class of best-match join operators addressing the problem of finding best matching pairs of data objects with regard to multiple dimensions. To overcome the inherently blocking nature of these operators and to improve the quality of the results we have employed constraints on join dimensions. These constraints in combination with physical properties of data streams have enabled our pipelined sliding-window best-match join algorithms. Further, we have discussed efficient I/O-scheduling techniques for processing materialized data windows. Our experimental evaluation has proven the good overall performance and pipelining behavior of these algorithms.

Appendix A

Complete XQuery$^-$ Grammar

RelPath ::= **QName** (“/” **QName**)*

FixedPath ::= **VarRef** (“/” *RelPath*)?

WinPath ::= *FixedPath* “|” *WinSpec* “|”

WinSpec ::= (“`count`” **IntegerLiteral** (“`step`” **IntegerLiteral**)?
| (*RelPath*)? “`diff`” **NumericLiteral** (“`step`” **NumericLiteral**)?)

AtomicClause ::= (“`fn:true()`” | “`fn:false()`”)

Function ::= (“`fn:`” (“`count`” | “`avg`” | “`max`” | “`min`” | “`sum`”)
| “`udf:`” **QName**) “`(`” *FixedPath* “`)`”

ValExpr ::= (**Literal**
| *FixedPath*
| *Function*
| *ValExpr* (“`+`” | “`-`”) *ValExpr*
| *ValExpr* (“`*`” | “`div`” | “`idiv`” | “`mod`”) *ValExpr*)

AtomicCondition ::= (*ValExpr* **GeneralComp** *ValExpr*
| *AtomicClause*
| (“`fn:empty`” | “`fn:exists`”) “`(`” *FixedPath* “`)`”
| “`fn:not(`” *AtomicCondition* “`)`”
| *AtomicCondition* “`and`” *AtomicCondition*
| *AtomicCondition* “`or`” *AtomicCondition*)

Expression ::= (“`()`”
| **Literal**
| *Expression* (*Expression*)+
| “`{`” “`for`” **VarRef** “`in`” *FixedPath* (“`where`” *AtomicCondition*)?
“`return`” *Expression* “`}`”
| “`{`” “`let`” **VarRef** “`:=`” *Function* (“`where`” *AtomicCondition*)?
“`return`” *Expression* “`}`”
| “`{`” “`for`” **VarRef** “`in`” *WinPath* (“`where`” *AtomicCondition*)?
“`return`” *Expression* “`}`”
| “`{`” *FixedPath* “`}`”
| “`{`” **VarRef** “`}`”
| “`{`” *Function* “`}`”
| “`{`” “`if`” “`(`” *AtomicCondition* “`)`” “`then`” *Expression* “`}`”)

Appendix B

Translating XQuery$^-$ into FluX: The Rewrite Algorithm

The complete algorithm for rewriting normalized XQuery$^-$ queries into FluX is shown on the next page.

Function rewrite(*Variable* $x, *Set*⟨Σ⟩ *H*, *XQuery*⁻ β) **returns** *FluXQuery*

```
1  begin
2      if {$x} ⪯ β then
3          if β is simple then
4              return β ;
5          else
6              return {ps $x:  on-first past(*) return β} ;
7          endif
8      else
9          if β is simple and H = ⊥ then
10             return β ;
11         else if β = β1 β2 then
12             if H = ⊥ then H := ∅ ;
13             β'1 := rewrite ($x, H, β1) ;
14             match ζ1 such that β'1 = {ps $x:  ζ1 } ;
15             β'2 := rewrite ($x, H ∪ hsymb(ζ1), β2) ;
16             match ζ2 such that β'2 = {ps $x:  ζ2 } ;
17             return {ps $x:  ζ1; ζ2 }
18         else if β is of the form {for $y in $z/a return α} then
19             X := {b ∈ dep($x, α) ∪ H | ¬Ord_$x(b, a)} ;
20             if $z ≠ $x then
21                 return {ps $x:  on-first past(X) return β};
22             else if X ≠ ∅ then
23                 return {ps $x:  on-first past(X ∪ {a}) return β} ;
24             else
25                 α' := rewrite ($y, ⊥, α) ;
26                 return {ps $x:  on a as $y return α'} ;
27             endif
28         else if β is of the form {for $y in $z/a/π|ws| return α} then
29             X := {b ∈ dep($x, α) ∪ H | ¬Ord_$x(b, a)};
30             if ($z ≠ $x) ∨ (X ≠ ∅) then
31                 return {ps $x:  on-first past(X ∪ {a}) return β};
32             else
33                 if β is simple and |ws| is simple then
34                     return {ps $x:  on-window $z/a/π|ws| as $y return β} ;
35                 else
36                     return {ps $x:  on-each past $z/a/π|ws| as $y return β} ;
37                 endif
38             endif
39         else
40             return {ps $x:  on-first past(dep($x, β) ∪ H) return β} ;
41         endif
42     endif
43 end
```

Appendix C

FluX Benchmark Experiments

In this section, we present our benchmark experiments based on the XMark benchmark and the queries we have used in detail.

C.1 Modified XMark Benchmark Queries

As briefly sketched in Section 3.9, we have adapted selected queries of the XMark benchmark to suit the capabilities of our prototype implementation. In detail, we have made the following changes:

- Attributes have been converted into subelements of their parent element. For example, an element

  ```
  <person id = "..."> ... </person>
  ```

  has been converted to:

  ```
  <person>
    <person_id>
      ...
    </person_id>
    ...
  </person>
  ```

  The XMark queries and the schema have been adapted accordingly. While processing an XML stream (generated by the XMark xmlgen data generation tool), our XSAX parser converts attributes into subelements on-the-fly.

- XPath kind tests such as `text()` have been omitted; the whole element is printed instead.

- Aggregations such as `count($x)` have been omitted and the whole subtree bound to `$x` is written to the output instead. In some queries this caused runtime to be mainly determined by the time taken to produce the output, in which case we have further restricted the queries to printing a fraction of the whole subtree.

The following queries have been used in our experiments (for all systems):

Query 1

```
<query1b>
  { for $b in /site/people/person
    where $b/person_id = "person0"
    return
      <result>
        {$b/name}
      </result> }
</query1b>
```

Query 5

```
<query5>
  { for $i in /site/closed_auctions/closed_auction
    where $i/price >= 40
    return
      $i/price }
</query5>
```

Query 8

```
<query8>
  { for $p in /site/people/person return
    <item>
      <person>
        {$p/name}
      </person>
      <items_bought>
        { for $t in /site/closed_auctions/closed_auction
          where $t/buyer/buyer_person = $p/person_id
          return
            <result>
              {$t}
            </result> }
      </items_bought>
    </item> }
</query8>
```

Query 8b

```
<query8b>
  { for $t in /site/closed_auctions/closed_auction return
    <item>
      <auction>
        {$t}
      </auction>
      { for $p in /site/people/person
        where $t/buyer/buyer_person = $p/person_id
        return
          <buyer>
            {$p/name}
          </buyer> }
    </item> }
</query8b>
```

Query 11

```
<query11>
  { for $p in /site/people/person return
    <items>
      {$p/name}
      { for $o in /site/open_auctions/open_auction
        where $p/profile/profile_income > (5000 * $o/initial)
        return
          $o/open_auction_id }
    </items> }
</query11>
```

Query 13

```
<query13>
  { for $i in $ROOT/site/regions/australia/item return
    <item>
      <name>
        {$i/name}
      </name>
      <desc>
        {$i/description}
      </desc>
    </item> }
</query13>
```

Query 16

```
<query16>
  { for $a in /site/closed_auctions/closed_auction
    where fn:not(fn:empty($a/annotation/description/parlist/listitem/ ⤦
                           parlist/listitem/text/emph/keyword))
    return
       $a/seller/seller_person }
</query16>
```

Query 17

```
<query17>
  { for $p in /site/people/person
    where fn:empty($p/homepage)
    return
      <person>
        {$p/name}
      </person> }
</query17>
```

Query 20

```
<query20>
  { for $p in /site/people/person
    where fn:empty($p/person_income)
    return
      $p }
</query20>
```

C.2 Additional Benchmark Queries

Our additional benchmark queries are also based on the data of the XMark benchmark. As before, attributes are converted into elements to suit our prototype implementation.

Query A

List the most expensive items sold.

```
<query_agg>
  { for $a in /site/closed_auctions return
      for $c in $a/closed_auction
      where $c/price = fn:max($a/closed_auction/price)
      return
        $c }
</query_agg>
```

Query W[1|2|3]

List the maximum current price of the last Δ *open items every* μ *items.*

```
<query_win>
  { for $w in /site/open_auctions/open_auction|count Δ step μ|
    return
      <max_item>
        {fn:max($w/current)}
      </max_item> }
</query_win>
```

In our experiments the following values for Δ and μ have been used:

	Δ	μ
Query W1	10	10
Query W2	10	5
Query W3	20	5

C.3 Benchmark Results

Figure C.1 shows the results of our performance tests using the queries presented in Section C.1. The term "DNF" denotes the situation that the query could not be processed due to, e. g., an "Out Of Memory" error. The results of the benchmarks using the queries shown in Section C.2, which involve aggregations and windowed computations, are shown in Figure C.2. The benchmark setup is described in Section 3.9.

		FluX					Galax	
		Unoptimized		Optimized		Compiled		
		Time [s]	Memory	Time [s]	Memory	Time [s]	Time [s]	Memory
Q_1	5M	1.5	112 Byte	1.4	0 Byte	1.4	1.6	32.1 MB
	10M	2.1	109 Byte	1.8	0 Byte	1.8	3.5	64.1 MB
	50M	5	114 Byte	5	0 Byte	4.6	24.8	318.9 MB
	100M	8.4	113 Byte	8.1	0 Byte	7.9	69.9	638.2 MB
Q_5	5M	1.4	91 Byte	1.4	91 Byte	1.4	1.6	32.2 MB
	10M	1.8	91 Byte	1.8	91 Byte	1.9	3.5	64.3 MB
	50M	4.8	91 Byte	4.8	91 Byte	4.9	24.9	320.3 MB
	100M	8.1	91 Byte	8.2	91 Byte	8.4	70.3	640.8 MB
Q_8	5M	12	1.4 MB	7.1	1.4 MB	6.5	13.6	34.2 MB
	10M	40	2.9 MB	23.1	2.9 MB	20.3	52	68.4 MB
	50M	1045.3	14.9 MB	499.6	14.9 MB	466.1	1707.7	340.4 MB
	100M	3967	30.1 MB	2375.5	30.1 MB	1873.1	8821.3	681.4 MB
Q_{8b}	5M	5.9	1.4 MB	5.2	209.7 kB	4.3	9.2	32.9 MB
	10M	18.4	2.9 MB	13.2	412 kB	10.1	33.7	65.8 MB
	50M	382.8	14.9 MB	308.9	1.9 MB	257.9	850.8	327.3 MB
	100M	1523.8	30.1 MB	1208.5	3.9 MB	995.5	3853.4	655 MB
Q_{11}	5M	5.6	275.3 kB	4.7	275.3 kB	3.8	43.3	32.7 MB
	10M	15.3	545.4 kB	12.4	545.4 kB	9.2	177	65.8 MB
	50M	376.6	2.6 MB	282.4	2.6 MB	187.5	6087.5	351.1 MB
	100M	1495.8	5.3 MB	1097.3	5.3 MB	742.8	*DNF*	*DNF*
Q_{13}	5M	1.3	8.7 kB	1.3	0 Byte	1.4	1.6	32.2 MB
	10M	1.8	8.6 kB	1.7	0 Byte	1.7	3.5	64.3 MB
	50M	4.8	13.4 kB	4.6	0 Byte	4.5	26.2	320.1 MB
	100M	8.2	15.6 kB	8.2	0 Byte	7.9	72	640.5 MB
Q_{16}	5M	1.3	129 Byte	1.3	129 Byte	1.5	1.6	32.1 MB
	10M	1.8	129 Byte	1.7	129 Byte	2.1	3.4	64.2 MB
	50M	5	130 Byte	4.7	130 Byte	4.6	26.8	319.6 MB
	100M	8.2	130 Byte	8.2	130 Byte	8.1	70.4	639.5 MB
Q_{17}	5M	1.5	112 Byte	1.4	112 Byte	1.6	1.6	32.6 MB
	10M	2	109 Byte	1.9	109 Byte	2	3.5	65.2 MB
	50M	5	114 Byte	5	114 Byte	5	27.6	324.7 MB
	100M	8.5	113 Byte	8.5	113 Byte	8.5	71.2	649.7 MB
Q_{20}	5M	1.6	4.1 kB	1.5	0 Byte	1.5	2	32.3 MB
	10M	1.9	4.7 kB	1.8	0 Byte	1.8	4.2	64.6 MB
	50M	5.3	6.2 kB	4.7	0 Byte	4.8	32.3	321.7 MB
	100M	9.3	6.3 kB	8.3	0 Byte	8.2	88.2	643.7 MB

Figure C.1: XMark Benchmark Results

		FluX				Galax	
		Unoptimized		Optimized			
		Time [s]	Memory	Time [s]	Memory	Time [s]	Memory
Q_A	5M	1.3	1.2 MB	1.6	1.2 MB	5.1	34.9 MB
	10M	1.8	2.5 MB	2.1	2.5 MB	17.7	126.6 MB
	50M	5.1	13 MB	5	13 MB	386	884.9 MB
	100M	8.7	26.2 MB	9.1	26.2 MB	1571.7	11.4 GB
Q_{W1}	5M	1.3	0 Byte	1.6	0 Byte	2	40.1 MB
	10M	1.6	0 Byte	1.6	0 Byte	4.4	81.7 MB
	50M	4.6	0 Byte	4.6	0 Byte	37.6	472.3 MB
	100M	8.2	0 Byte	8.1	0 Byte	137.5	1 GB
Q_{W2}	5M	1.3	910 Byte	1.5	910 Byte	2.4	48 MB
	10M	1.9	910 Byte	1.7	910 Byte	5.5	99.2 MB
	50M	5.1	910 Byte	5	910 Byte	50.8	632.8 MB
	100M	8.6	910 Byte	8.5	910 Byte	202.6	1.5 GB
Q_{W3}	5M	1.3	1.7 kB	1.4	1.7 kB	3.2	61.9 MB
	10M	1.9	1.7 kB	1.8	1.7 kB	7.4	128.1 MB
	50M	4.9	1.7 kB	4.9	1.7 kB	72.6	776.6 MB
	100M	8.6	1.7 kB	8.5	1.7 kB	261.9	1.8 GB

Figure C.2: Extensions Benchmark Results

Appendix D

Translating XQuery$^-$ into FluX: The Rewrite System

Our rewrite algorithm for transforming normalized XQuery$^-$ into FluX (function "rewrite" in Section 3.5) is based on a corresponding rewrite system. The rules of that rewrite system are given in Figure D.1.

Our algorithm "rewrite" applies the rules of this rewrite system according to the following rewrite strategy: Given a DTD, for transforming a given normalized XQuery$^-$ query Q into a FluX query, the rewrite rules of Figure D.1 are applied in the following way: Do one top-down left-to-right pass over the query tree of Q, where the children of each node are processed from left to right. Each node v (or, more precisely, the subtree rooted at v) represents a subexpression α_v of Q. If α_v is *simple* and none of the rules in Figure D.1 can be applied to α_v, then α_v will *not* be further rewritten and the subtree rooted at node v will not be further processed. In all other cases, all possible rewrite rules are applied at node v. Inside "`on-first`"-handlers, no application of rewrite rules is allowed.

During the rewrite process intermediate results might occur that are not syntactically correct FluX queries. However, following the above strategy, the rewriting will always *terminate* with a syntactically correct FluX query, denoted $FluX_{DTD}(Q)$. Note, that one can easily define an extension FluX$^+$ of FluX such that the rewrite rules of Figure D.1 are in fact *equivalences* and all intermediate results that occur during the rewrite process belong to FluX$^+$.

For every XQuery$^-$ query Q in normal form, $FluX_{DTD}(Q)$ can be computed in $O(|Q|)$ rewrite steps, and $FluX_{DTD}(Q)$ is a safe FluX query that is equivalent to Q on all XML documents compliant with the DTD. In fact, $FluX_{DTD}(Q)$ is the FluX query produced by our rewriting algorithm via calling "rewrite($\$ROOT$, $\perp$, Q)".

Let us demonstrate the rewrite system by considering once again Example 3.5.3.

Example D.0.1 ([W3C05a], XMP-Q2) *Let Q'_2 be the normalized XQuery$^-$ query from Example 3.5.3. Rewriting Q'_2 into $FluX_{DTD}(Q'_2)$, we first apply the rule [For3] to rewrite the outermost* `for`*-loop into a "*`{ps $ROOT: on bib ···}`*"-expression. Then, we apply the rules [Mrg-HLists5] and [Mrg-HLists4] to move the strings "*`<results>`*" and "*`</results>`*"

$$\frac{\beta\texttt{:}\ \ \texttt{\{for }\$y\texttt{ in }\$x/a\texttt{ return }\alpha\texttt{\}}}{\begin{array}{l}\texttt{\{ps }\$r\texttt{:}\ \ \texttt{on-first past(}X\texttt{) return}\\ \quad\texttt{\{for }\$y\texttt{ in }\$x/a\texttt{ return }\alpha\texttt{\};\}}\end{array}} \quad \left(\begin{array}{l}\$r = \mathit{parentVar}(\beta),\ \$r \neq \$x,\ \{\$r\} \not\preceq \alpha,\\ X = \{b \in \mathit{dep}(\$r,\ \alpha) \mid \neg \mathit{Ord}_{\$r}(b,\ a)\}\end{array}\right) \quad \textit{[For1]}$$

$$\frac{\beta\texttt{:}\ \ \texttt{\{for }\$x\texttt{ in }\$r/a\texttt{ return }\alpha\texttt{\}}}{\begin{array}{l}\texttt{\{ps }\$r\texttt{:}\ \ \texttt{on-first past(}X \cup \{a\}\texttt{) return}\\ \quad\texttt{\{for }\$x\texttt{ in }\$r/a\texttt{ return }\alpha\texttt{\};\}}\end{array}} \quad \left(\begin{array}{l}\$r = \mathit{parentVar}(\beta),\ \{\$r\} \not\preceq \alpha,\\ X = \{b \in \mathit{dep}(\$r,\ \alpha) \mid \neg \mathit{Ord}_{\$r}(b,\ a)\},\\ X \neq \emptyset\end{array}\right) \quad \textit{[For2]}$$

$$\frac{\beta\texttt{:}\ \ \texttt{\{for }\$x\texttt{ in }\$r/a\texttt{ return }\alpha\texttt{\}}}{\texttt{\{ps }\$r\texttt{:}\ \ \texttt{on }a\texttt{ as }\$x\texttt{ return }\alpha\texttt{\}}} \quad \left(\begin{array}{l}\$r = \mathit{parentVar}(\beta),\ \{\$r\} \not\preceq \alpha,\\ \forall\ b \in \mathit{dep}(\$r,\ \alpha)\ :\ \mathit{Ord}_{\$r}(b,\ a)\end{array}\right) \quad \textit{[For3]}$$

$$\frac{\alpha}{\texttt{\{ps }\$x\texttt{:}\ \ \texttt{on-first past(*) return }\alpha\texttt{\}}} \quad \left(\begin{array}{l}\$x = \mathit{parentVar}(\alpha),\ \{\$x\} \preceq \alpha,\\ \mathit{dep}(\$x,\ \alpha) \neq \emptyset \text{ or } \alpha \text{ is not } \mathit{simple}\end{array}\right) \quad \textit{[Output1]}$$

$$\frac{\alpha\texttt{:}\ \ \texttt{\{if }\chi\texttt{ then }s\texttt{\}}}{\texttt{\{ps }\$x\texttt{:}\ \ \texttt{on-first past(}\mathit{dep}(\$x,\ \alpha)\texttt{) return }\alpha\texttt{\}}} \quad (\$x = \mathit{parentVar}(\alpha)) \quad \textit{[Output2]}$$

$$\frac{\begin{array}{l}\texttt{\{ps }\$r\texttt{:}\ \ \zeta\texttt{\}}\\ \texttt{\{ps }\$r\texttt{:}\ \ \texttt{on }a\texttt{ as }\$x\texttt{ return }\alpha\texttt{\}}\end{array}}{\texttt{\{ps }\$r\texttt{:}\ \ \zeta\texttt{; on }a\texttt{ as }\$x\texttt{ return }\alpha\texttt{\}}} \quad (\forall\ b \in \mathit{hsymb}(\zeta)\ :\ \mathit{Ord}_{\$r}(b,\ a)) \quad \textit{[Mrg-HLists1]}$$

$$\frac{\begin{array}{l}\texttt{\{ps }\$r\texttt{:}\ \ \zeta\texttt{\}}\\ \texttt{\{ps }\$r\texttt{:}\ \ \texttt{on }a\texttt{ as }\$x\texttt{ return }\alpha\texttt{\}}\end{array}}{\begin{array}{l}\texttt{\{ps }\$r\texttt{:}\ \ \zeta\texttt{; on-first past(}X \cup \{a\}\texttt{) return}\\ \quad\texttt{\{for }\$x\texttt{ in }\$r/a\texttt{ return }\alpha\texttt{\};\}}\end{array}} \quad \left(\begin{array}{l}X = \{b \in \mathit{hsymb}(\zeta) \mid \neg \mathit{Ord}_{\$r}(b,\ a)\},\\ X \neq \emptyset,\ \alpha \text{ is an XQuery}\end{array}\right) \quad \textit{[Mrg-HLists2]}$$

$$\frac{\begin{array}{l}\texttt{\{ps }\$r\texttt{:}\ \ \zeta\texttt{\}}\\ \texttt{\{ps }\$r\texttt{:}\ \ \texttt{on-first past(}X\texttt{) return }\alpha\texttt{\}}\end{array}}{\texttt{\{ps }\$r\texttt{:}\ \ \zeta\texttt{; on-first past(}X \cup \mathit{hsymb}(\zeta)\texttt{) return }\alpha\texttt{\}}} \quad \textit{[Mrg-HLists3]}$$

$$\frac{\texttt{\{ps }\$r\texttt{:}\ \ \zeta\texttt{\} }s}{\texttt{\{ps }\$r\texttt{:}\ \ \zeta\texttt{; on-first past(}\mathit{hsymb}(\zeta)\texttt{) return }s\texttt{\}}} \quad \textit{[Mrg-HLists4]}$$

$$\frac{s\ \texttt{\{ps }\$r\texttt{:}\ \ \zeta\texttt{\}}}{\texttt{\{ps }\$r\texttt{:}\ \ \texttt{on-first past() return }s\texttt{; }\zeta\texttt{\}}} \quad \textit{[Mrg-HLists5]}$$

Figure D.1: Rewrite Rules for Translation of XQuery into FluX (Using a DTD)

into this expression. With another application of rule [For3] we rewrite the subquery inside the "`on bib`*"-handler into a "*`{ps $bib: on book ...}`*"-expression.*

How the subquery inside the "`on book`*"-handler is rewritten into FluX depends on the particular DTD.*

When given the first DTD from Section 3.1, no order constraint on `title` *and* `author` *holds, and we have to rewrite the* `for`*-loop over titles using rule [For2]. To check that this rule can indeed be applied, note that* `$b` *is the parent variable of this* `for`*-loop and that* $X = \{\texttt{author}\}$ *is the set of dependencies for* `$b`*. Applying rule [For2], we obtain the*

following query $FluX_{DTD}(Q)$, which is exactly the FluX query F_2 from Example 3.5.3:

```
1  { ps $ROOT:
2    on-first past() return <results>;
3    on bib as $bib return
4      { ps $bib:
5        on book as $b return
6          { ps $b:
7            on-first past(author, title) return
8              { for $t in $b/title return
9                { for $a in $b/author return
10                 <result>
11                 {$t}
12                 {$a}
13                 </result> } }; }; };
14   on-first past(bib) return </results>; }
```

We will refer to the expression in lines 6–13 of F_2 as α'_2.

Let us now consider the case where we are given a different DTD with the production

```
<!ELEMENT book (author*, title*)>
```

in which the order constraint $Ord_{\texttt{book}}$(`author`, `title`) *is met. Then, rule [For3] can be applied to rewrite the* `for`*-loop over titles. Thus, an "*`on title`*"-handler is created instead of an "*`on-first past(author, title)`*"-handler. Considering the event expression of this newly created handler, we apply rule [Output1], as the parent variable* `$t` *is output inside the event expression. The resulting query $FluX_{DTD}(Q)$ differs from the solution above in the subexpression α'_2, which is replaced by*

```
{ ps $b:
  on title as $t return
    { ps $t:
      on-first past(*) return
        { for $a in $b/author return
          <result>
          {$t}
          {$a}
          </result> }; }; }
```

Note, that the resulting query $FluX_{DTD}(Q)$ now is precisely the query F'_2 from Example 3.5.3.
□

Bibliography

[AAB+05] D. J. Abadi, Y. Ahmad, M. Balazinska, U. Çetintemel, M. Cherniack, J.-H. Hwang, W. Lindner, A. Maskey, A. Rasin, E. Ryvkina, N. Tatbul, Y. Xing, and S. B. Zdonik. The Design of the Borealis Stream Processing Engine. In *Proc. of the Conf. on Innovative Data Systems Research*, pages 277–289, Asilomar, CA, USA, January 2005.

[ABB+02] A. Arasu, B. Babcock, S. Babu, J. McAlister, and J. Widom. Characterizing Memory Requirements for Queries over Continuous Data Streams. Technical report 2002-29, Stanford University, 2002.

[ABB+03] A. Arasu, B. Babcock, S. Babu, M. Datar, K. Ito, R. Motwani, I. Nishizawa, U. Srivastava, D. Thomas, R. Varma, and J. Widom. STREAM: The Stanford Stream Data Manager. *IEEE Data Engineering Bulletin*, 26(1):19–26, March 2003.

[ABS00] S. Abiteboul, P. Buneman, and D. Suciu. *Data on the Web*. Morgan Kaufmann, 2000.

[ABW03] A. Arasu, S. Babu, and J. Widom. The CQL Continuous Query Language: Semantic Foundations and Query Execution. Technical report 2003-67, Stanford University, 2003.

[ACGG+02] I. Avila-Campillo, T. J. Green, A. Gupta, M. Onizukaz, D. Raven, and D. Suciu. XMLTK: An XML Toolkit for Scalable XML Stream Processing. In *Workshop on Programming Language Technologies for XML*, Pittsburgh, PA, USA, October 2002.

[ACMD+03] K. Aberer, P. Cudré-Mauroux, A. Datta, Z. Despotovic, M. Hauswirth, M. Punceva, and R. Schmidt. P-Grid: A Self-organizing Structured P2P System. *ACM SIGMOD Record*, 32(3):29–33, 2003.

[ACMH03] K. Aberer, P. Cudré-Mauroux, and M. Hauswirth. The Chatty Web: Emergent Semantics Through Gossiping. In *Proc. of the Intl. World Wide Web Conference*, pages 197–206, Budapest, Hungary, May 2003.

[AF00] M. Altinel and M. J. Franklin. Efficient Filtering of XML Documents for Selective Dissemination of Information. In *Proc. of the Intl. Conf. on Very Large Data Bases*, pages 53–64, Cairo, Egypt, September 2000.

[Asc98] B. Aschenbach. Discovery of a Young Nearby Supernova Remnant. *Nature*, 396(6707):141–142, November 1998.

[BBD+02] B. Babcock, S. Babu, M. Datar, R. Motwani, and J. Widom. Models and Issues in Data Stream Systems. In *Proc. of the ACM SIGACT-SIGMOD-SIGART Symp. on Principles of Database Systems*, pages 1–16, Madison, WI, USA, June 2002.

[BCG+03] C. Barton, P. Charles, D. Goyal, M. Raghavachari, M. Fontoura, and V. Josifovski. Streaming XPath Processing with Forward and Backward Axes. In *Proc. of the IEEE Intl. Conf. on Data Engineering*, pages 455–466, Bangalore, India, March 2003.

[BDK+03] I. Brunkhorst, H. Dhraief, A. Kemper, W. Nejdl, and C. Wiesner. Distributed Queries and Query Optimization in Schema-Based P2P-Systems. In *Proc. of the Intl. Workshop On Databases, Information Systems and Peer-to-Peer Computing*, Berlin, Germany, September 2003.

[Bec04] O. Becker. *Serielle Transformationen von XML – Probleme, Methoden, Lösungen*. PhD thesis, Humboldt-Universität zu Berlin, 2004. http://edoc.hu-berlin.de/dissertationen/becker-oliver-2004-11-26/PDF/Becker.pdf.

[BF05] S. Bose and L. Fegaras. XFrag: A Query Processing Framework for Fragmented XML Data. In *Proc. of the Intl. Workshop on the Web & Databases*, pages 97–102, Baltimore, Maryland, USA, June 2005.

[BGK03] P. Buneman, M. Grohe, and C. Koch. Path Queries on Compressed XML. In *Proc. of the Intl. Conf. on Very Large Data Bases*, pages 141–152, Berlin, Germany, September 2003.

[BK93] A. Brüggemann-Klein. Regular Expressions into Finite Automata. *Theoretical Computer Science*, 120(2):197–213, 1993.

[BK01] C. Böhm and H.-P. Kriegel. A Cost Model and Index Architecture for the Similarity Join. In *Proc. IEEE Conf. on Data Engineering*, pages 411–420, Heidelberg, Germany, 2001.

[BKK+01] R. Braumandl, M. Keidl, A. Kemper, D. Kossmann, A. Kreutz, S. Seltzsam, and K. Stocker. ObjectGlobe: Ubiquitous query processing on the Internet. *The VLDB Journal*, 10(1):48–71, August 2001.

[BKK03] R. Braumandl, A. Kemper, and D. Kossmann. Quality of Service in an Information Economy. *ACM Transactions on Internet Technology*, 3(4):291–333, November 2003.

[BKS01] S. Börzsönyi, D. Kossmann, and K. Stocker. The Skyline Operator. In *Proc. IEEE Conf. on Data Engineering*, pages 421–430, Heidelberg, Germany, 2001.

[BKW98] A. Brüggemann-Klein and D. Wood. One-Unambiguous Regular Languages. *Information and Computation*, 142(2):182–206, 1998.

[BT99] C. Beeri and Y. Tzaban. SAL: An Algebra for Semistructured Data and XML. In *Proc. of the Intl. Workshop on the Web & Databases*, pages 37–42, Philadelphia, Pennsylvania, USA, June 1999.

[BW01] S. Babu and J. Widom. Continuous Queries over Data Streams. *ACM SIGMOD Record*, 30(3):109–120, March 2001.

[CBB+03] M. Cherniack, H. Balakrishnan, M. Balazinska, D. Carney, U. Çetintemel, Y. Xing, and S. B. Zdonik. Scalable Distributed Stream Processing. In *Proc. of the Conf. on Innovative Data Systems Research*, Asilomar, CA, USA, January 2003.

[CCD+03] S. Chandrasekaran, O. Cooper, A. Deshpande, M. J. Franklin, J. M. Hellerstein, W. Hong, S. Krishnamurthy, S. Madden, V. Raman, F. Reiss, and M. A. Shah. TelegraphCQ: Continuous Dataflow Processing for an Uncertain World. In *Proc. of the Conf. on Innovative Data Systems Research*, Asilomar, CA, USA, January 2003.

[CDTW00] J. Chen, D. J. DeWitt, F. Tian, and Y. Wang. NiagaraCQ: A Scalable Continuous Query System for Internet Databases. In *Proc. of the ACM SIGMOD Intl. Conf. on Management of Data*, pages 379–390, Dallas, TX, USA, May 2000.

[CF03] S. Chandrasekaran and M. J. Franklin. PSoup: a system for streaming queries over streaming data. *The VLDB Journal*, 12(2):140–156, 2003.

[CFGR02] C. Y. Chan, P. Felber, M. N. Garofalakis, and R. Rastogi. Efficient filtering of XML documents with XPath expressions. *The VLDB Journal*, 11(4):354–379, 2002.

[CGGL03] J. Chomicki, P. Godfrey, J. Gryz, and D. Liang. Skyline with Presorting. In *Proc. IEEE Conf. on Data Engineering*, pages 717–719, Bangalore, India, 2003.

[Cho03] J. Chomicki. Preference Formulas in Relational Queries. *ACM Trans. on Database Systems*, 28(4):427–466, 2003.

[CRA04] L. Chen, K. Reddy, and G. Agrawal. GATES: A Grid-Based Middleware for Processing Distributed Data Streams. In *Proc. of the IEEE Intl. Symp. on High-Performance Distributed Computing*, Honolulu, HI, USA, June 2004.

[CÇC+02] D. Carney, U. Çetintemel, M. Cherniack, C. Convey, S. Lee, G. Seidman, M. Stonebraker, N. Tatbul, and S. B. Zdonik. Monitoring Streams - A New Class of Data Management Applications. In *Proc. of the Intl. Conf. on Very Large Data Bases*, pages 215–226, Hong Kong, China, August 2002.

[DC90] S. E. Deering and D. R. Cheriton. Multicast Routing in Datagram Internetworks and Extended LANs. *ACM Transactions on Computer Systems*, 8(2):85–110, May 1990.

[DF03] Y. Diao and M. Franklin. Query Processing for High-Volume XML Message Brokering. In *Proc. of the Intl. Conf. on Very Large Data Bases*, pages 261–272, Berlin, Germany, September 2003.

[DFFT02] Y. Diao, P. M. Fischer, M. J. Franklin, and R. To. YFilter: Efficient and Scalable Filtering of XML Documents. In *Proc. of the IEEE Intl. Conf. on Data Engineering*, pages 341–342, San José, CA, USA, February 2002.

[DGR03] A. Das, J. Gehrke, and M. Riedewald. Approximate Join Processing Over Data Streams. In *Proc. of the ACM SIGMOD Intl. Conf. on Management of Data*, pages 40–51, San Diego, CA, USA, June 2003.

[DRF04] Y. Diao, S. Rizvi, and M. J. Franklin. Towards an Internet-Scale XML Dissemination Service. In *Proc. of the Intl. Conf. on Very Large Data Bases*, pages 612–623, Toronto, Canada, August 2004.

[DT03] A. Deutsch and V. Tannen. Reformulation of XML Queries and Constraints. In *Proc. of the Intl. Conf. on Database Theory*, pages 225–241, Siena, Italy, January 2003.

[FB04] W. Fontijn and P. A. Boncz. AmbientDB: P2P Data Management Middleware for Ambient Intelligence. In *Workshop on Middleware Support for Pervasive Computing (PerWare), In conjunction with the IEEE International Conference on Pervasive Computing and Communications (PerCom)*, Orlando, FL, USA, March 2004.

[FHK+00] D. Florescu, C. Hillery, D. Kossmann, P. Lucas, F. Riccardi, T. Westmann, M. J. Carey, A. Sundararajan, and G. Agrawal. The BEA/XQRL Streaming XQuery Processor. In *Proc. of the ACM SIGMOD Intl. Conf. on Management of Data*, pages 997–1008, Dallas, TX, USA, May 2000.

[FHK+03] D. Florescu, C. Hillery, D. Kossmann, P. Lucas, F. Riccardi, T. Westmann, M. J. Carey, A. Sundararajan, and G. Agrawal. The BEA/XQRL Streaming

XQuery Processor. In *Proc. of the Intl. Conf. on Very Large Data Bases*, pages 997–1008, Berlin, Germany, September 2003.

[FK04] I. Foster and C. Kesselman, editors. *The Grid: Blueprint for a New Computing Infrastructure, 2nd Edition.* Morgan Kaufmann, 2004.

[FKNT02] I. Foster, C. Kesselman, J. Nick, and S. Tuecke. The Physiology of the Grid: An Open Grid Services Architecture for Distributed Systems Integration, June 2002. http://www.globus.org/research/papers/ogsa.pdf.

[FKT01] I. Foster, C. Kesselman, and S. Tuecke. The Anatomy of the Grid: Enabling Scalable Virtual Organizations. *The Intl. Journal of Supercomputer Applications and High Performance Computing*, 15(3):200–222, August 2001.

[FLBC02] L. Fegaras, D. Levine, S. Bose, and V. Chaluvadi. Query Processing of Streamed XML Data. In *Proc. of the ACM Intl. Conf. on Information and Knowledge Management*, pages 126–133, McLean, VA, USA, November 2002.

[FSW01] M. F. Fernández, J. Siméon, and P. Wadler. A Semi-monad for Semi-structured Data. In *Proc. of the Intl. Conf. on Database Theory*, pages 263–300, London, UK, January 2001.

[GAV04] German Astrophysical Virtual Observatory. http://www.g-vo.org, 2004.

[GGF05] The Global Grid Forum (GGF), 2005. http://www.ggf.org/.

[Glo04] The Globus Alliance. http://www.globus.org, 2004.

[GM82] H. Garcia-Molina. Elections in a Distributed Computing System. *IEEE Transactions on Computers*, C-31(1):48–59, January 1982.

[GMOS03] T. J. Green, G. Miklau, M. Onizuka, and D. Suciu. Processing XML Streams with Deterministic Automata. In *Proc. of the Intl. Conf. on Database Theory*, pages 173–189, Siena, Italy, January 2003.

[GS03] A. K. Gupta and D. Suciu. Stream Processing of XPath Queries with Predicates. In *Proc. of the ACM SIGMOD Intl. Conf. on Management of Data*, pages 419–430, San Diego, CA, USA, June 2003.

[GÖ03] L. Golab and M. T. Özsu. Processing Sliding Window Multi-Joins in Continuous Queries over Data Streams. In *Proc. of the Intl. Conf. on Very Large Data Bases*, pages 500–511, Berlin, Germany, September 2003.

[HCH+05] R. Huebsch, B. N. Chun, J. M. Hellerstein, B. T. Loo, P. Maniatis, T. Roscoe, S. Shenker, I. Stoica, and A. R. Yumerefendi. The Architecture of PIER: an Internet-Scale Query Processor. In *Proc. of the Conf. on Innovative Data Systems Research*, pages 28–43, Asilomar, CA, USA, January 2005.

[HFAE03] M. A. Hammad, M. J. Franklin, W. G. Aref, and A. K. Elmagarmid. Scheduling for shared window joins over data streams. In *Proc. of the Intl. Conf. on Very Large Data Bases*, pages 297–308, Berlin, Germany, September 2003.

[HLR03] Q. Huang, C. Lu, and G.-C. Roman. Spatiotemporal Multicast in Sensor Networks. In *Proc. of the Intl. Conf. on Embedded Networked Sensor Systems*, pages 205–217, Los Angeles, CA, USA, November 2003.

[HU79] J. E. Hopcroft and J. D. Ullman. *Introduction to Automata Theory, Languages, and Computation.* Addison-Wesley Publishing Company, 1979.

[IHW02] Z. G. Ives, A. Y. Halevy, and D. S. Weld. An XML Query Engine for Network-Bound Data. *The VLDB Journal*, 11(4):380–402, 2002.

[JLST01] H. Jagadish, L. Lakshmanan, D. Srivastava, and K. Thompson. TAX: A Tree Algebra for XML. In *Proc. of the Intl. Workshop on Database Programming Languages*, pages 149–164, Frascati, Italy, September 2001.

[Kie02] W. Kießling. Foundations of Preferences in Database Systems. In *Proc. of the Intl. Conf. on Very Large Data Bases*, pages 311–322, Hong Kong, China, August 2002.

[KKKK02] M. Keidl, A. Kreutz, A. Kemper, and D. Kossmann. A Publish & Subscribe Architecture for Distributed Metadata Management. In *Proc. of the IEEE Intl. Conf. on Data Engineering*, pages 309–320, San José, CA, USA, February 2002.

[KLP75] H. T. Kung, F. Luccio, and F. P. Preparata. On Finding the Maxima of a Set of Vectors. *Journal of the ACM*, 22(4):469–476, 1975.

[KNV03] J. Kang, J. F. Naughton, and S. D. Viglas. Evaluating Window Joins over Unbounded Streams. In *Proc. IEEE Conf. on Data Engineering*, pages 341–352, Bangalore, India, 2003.

[KRR02] D. Kossmann, F. Ramsak, and S. Rost. Shooting Stars in the Sky: An Online Algorithm for Skyline Queries. In *Proc. of the Intl. Conf. on Very Large Data Bases*, pages 275–286, Hong Kong, China, August 2002.

[KS02] A. Kemper and B. Stegmaier. Evaluating Bestmatch-Joins on Streaming Data. Technical Report MIP-0204, Universität Passau, 2002.

[KS03] C. Koch and S. Scherzinger. Attribute Grammars for Scalable Query Processing on XML Streams. In *Proc. of the Intl. Workshop on Database Programming Languages*, pages 233–256, Potsdam, Germany, September 2003.

[KS04] J. Krämer and B. Seeger. PIPES - A Public Infrastructure for Processing and Exploring Streams. In *Proc. of the ACM SIGMOD Intl. Conf. on Management of Data*, pages 925–926, Paris, France, June 2004.

[KSH+04] R. Kuntschke, B. Stegmaier, F. Häuslschmid, A. Reiser, A. Kemper, H.-M. Adorf, H. Enke, G. Lemson, and W. Voges. Datenstrom-Management für e-Science mit StreamGlobe. *Datenbank Spektrum*, (11):14–22, November 2004.

[KSK05] R. Kuntschke, B. Stegmaier, and A. Kemper. Data Stream Sharing. Technical Report TUM-I0504, TU München, 2005. To Appear.

[KSKR05] R. Kuntschke, B. Stegmaier, A. Kemper, and A. Reiser. StreamGlobe: Processing and Sharing Data Streams in Grid-Based P2P Infrastructures. In *Proc. of the Intl. Conf. on Very Large Data Bases*, Trondheim, Norway, August 2005.

[KSSS04a] C. Koch, S. Scherzinger, N. Schweikardt, and B. Stegmaier. FluXQuery: An Optimizing XQuery Processor for Streaming XML Data. In *Proc. of the Intl. Conf. on Very Large Data Bases*, pages 1309–1312, Toronto, Canada, August 2004.

[KSSS04b] C. Koch, S. Scherzinger, N. Schweikardt, and B. Stegmaier. Schema-based Scheduling of Event Processors and Buffer Minimization for Queries on Structured Data Streams. *CoRR*, cs.DB/0406016, 2004.

[KSSS04c] C. Koch, S. Scherzinger, N. Schweikardt, and B. Stegmaier. Schema-based Scheduling of Event Processors and Buffer Minimization on Structured Data Streams. In *Proc. of the Intl. Conf. on Very Large Data Bases*, pages 228–239, Toronto, Canada, August 2004.

[LA05] X. Li and G. Agrawal. Efficient Evaluation of XQuery over Streaming Data. In *Proc. of the Intl. Conf. on Very Large Data Bases*, Trondheim, Norway, August 2005.

[LF04] D. T. Liu and M. J. Franklin. GridDB: A Data-Centric Overlay for Scientific Grids. In *Proc. of the Intl. Conf. on Very Large Data Bases*, pages 600–611, Toronto, Canada, August 2004.

[LKK+97] P. C. Lockemann, U. Kölsch, A. Koschel, R. Kramer, R. Nikolai, M. Wallrath, and H.-D. Walter. The Network as a Global Database: Challenges of Interoperability, Proactivity, Interactiveness, Legacy. In *Proc. of the Intl. Conf. on Very Large Data Bases*, pages 567–574, Athens, Greece, August 1997.

[LMP02] B. Ludäscher, P. Mukhopadhyay, and Y. Papakonstantinou. A Transducer-Based XML Query Processor. In *Proc. of the Intl. Conf. on Very Large Data Bases*, pages 227–238, Hong Kong, China, August 2002.

[LMSS95] A. Y. Levy, A. O. Mendelzon, Y. Sagiv, and D. Srivastava. Answering Queries Using Views. In *Proc. of the ACM SIGACT-SIGMOD-SIGART Symp. on Principles of Database Systems*, pages 95–104, San José, CA, USA, May 1995.

[LMT+05] J. Li, D. Maier, K. Tufte, V. Papadimos, and P. A. Tucker. Semantics and Evaluation Techniques for Window Aggregates in Data Streams. In *Proc. of the ACM SIGMOD Intl. Conf. on Management of Data*, Baltimore, Maryland, USA, June 2005.

[LS00] H. Liefke and D. Suciu. XMill: An Efficient Compressor for XML Data. In *Proc. of the ACM SIGMOD Intl. Conf. on Management of Data*, pages 153–164, Dallas, TX, USA, May 2000.

[MF02] S. Madden and M. J. Franklin. Fjording the Stream: An Architecture for Queries Over Streaming Sensor Data. In *Proc. of the IEEE Intl. Conf. on Data Engineering*, pages 555–566, San José, CA, USA, February 2002.

[MLM01] M. Murata, D. Lee, and M. Mani. Taxonomy of XML Schema Languages using Formal Language Theory. In *Extreme Markup Languages*, Montreal, Canada, August 2001.

[MRSR01] H. Mistry, P. Roy, S. Sudarshan, and K. Ramamritham. Materialized View Selection and Maintenance Using Multi-Query Optimization. In *Proc. of the ACM SIGMOD Intl. Conf. on Management of Data*, pages 307–318, Santa Barbara, CA, USA, May 2001.

[MS03] A. Marian and J. Siméon. Projecting XML Documents. In *Proc. of the Intl. Conf. on Very Large Data Bases*, pages 213–224, Berlin, Germany, September 2003.

[MSHR02] S. Madden, M. A. Shah, J. M. Hellerstein, and V. Raman. Continuously Adaptive Continuous Queries over Streams. In *Proc. of the ACM SIGMOD Intl. Conf. on Management of Data*, pages 49–60, Madison, WI, USA, June 2002.

[MWA+03] R. Motwani, J. Widom, A. Arasu, B. Babcock, S. Babu, M. Datar, G. S. Manku, C. Olston, J. Rosenstein, and R. Varma. Query Processing, Approximation, and Resource Management in a Data Stream Management System. In *Proc. of the Conf. on Innovative Data Systems Research*, Asilomar, CA, USA, January 2003.

[NSGS+05] M. A. Nieto-Santisteban, J. Gray, A. S. Szalay, J. Annis, A. R. Thakar, and W. O'Mullane. When Database Systems Meet the Grid. In *Proc. of the Conf. on Innovative Data Systems Research*, pages 154–161, Asilomar, CA, USA, January 2005.

[OGS04] OGSA-DAI. http://www.ogsadai.org.uk, 2004.

[OKB03] D. Olteanu, T. Kiesling, and F. Bry. An Evaluation of Regular Path Expressions with Qualifiers against XML Streams. In *Proc. of the IEEE Intl. Conf. on Data Engineering*, pages 702–704, Bangalore, India, March 2003.

[PC03] F. Peng and S. S. Chawathe. XPath Queries on Streaming Data. In *Proc. of the ACM SIGMOD Intl. Conf. on Management of Data*, pages 431–442, San Diego, CA, USA, June 2003.

[PMT03] V. Papadimos, D. Maier, and K. Tufte. Distributed Query Processing and Catalogs for Peer-to-Peer Systems. In *Proc. of the Conf. on Innovative Data Systems Research*, Asilomar, CA, USA, January 2003.

[PS85] F. P. Preparata and M. I. Shamos. *Computational Geometry: An Introduction*. Springer-Verlag, New York, Berlin, etc., 1985.

[PTFS03] D. Papadias, A. Tao, G. Fu, and B. Seeger. An Optimal and Progressive Algorithm for Skyline Queries. In *Proc. of the ACM SIGMOD Intl. Conf. on Management of Data*, pages 467–478, San Diego, CA, USA, June 2003.

[RD01] A. Rowstron and P. Druschel. Pastry: Scalable, Decentralized Object Location, and Routing for Large-Scale Peer-to-Peer Systems. In *Proc. of the IFIP/ACM Intl. Conf. on Distributed Systems Platforms*, pages 329–350, Heidelberg, Germany, November 2001.

[RFH+01] S. Ratnasamy, P. Francis, M. Handley, R. Karp, and S. Schenker. A Scalable Content-Addressable Network. In *Proc. of the Conf. on Applications, Technologies, Architectures, and Protocols for Computer Communications*, pages 161–172, San Diego, CA, USA, 2001.

[RH02] V. Raman and J. M. Hellerstein. Partial Results for Online Query Processing. In *Proc. of the ACM SIGMOD Intl. Conf. on Management of Data*, pages 275–286, Madison, WI, USA, June 2002.

[RHKS02] S. Ratnasamy, M. Handley, R. Karp, and S. Shenker. Topologically-Aware Overlay Construction and Server Selection. In *Proceedings of IEEE INFOCOM*, New York, USA, June 2002.

[RKV95] N. Roussopoulos, S. Kelley, and F. Vincent. Nearest Neighbor Queries. In *Proc. of the ACM SIGMOD Intl. Conf. on Management of Data*, pages 71–79, San Jose, CA, USA, May 1995.

[Sel88] T. K. Sellis. Multiple-Query Optimization. *ACM Trans. on Database Systems*, 13(1):23–52, March 1988.

[SK04] B. Stegmaier and R. Kuntschke. StreamGlobe: Adaptive Anfragebearbeitung und Optimierung auf Datenströmen. In *GI Workshop Dynamische Informationsfusion*, Ulm, Germany, September 2004.

[SK05] S. Scherzinger and A. Kemper. Syntax-directed Transformations of XML Streams. In *Workshop on Programming Language Technologies for XML*, Long Beach, USA, January 2005.

[SKK04] B. Stegmaier, R. Kuntschke, and A. Kemper. StreamGlobe: Adaptive Query Processing and Optimization in Streaming P2P Environments. In *Proc. of the Intl. Workshop on Data Management for Sensor Networks*, pages 88–97, Toronto, Canada, August 2004.

[SMK+01] I. Stoica, R. Morris, D. Karger, M. F. Kaashoek, and H. Balakrishnan. Chord: A Scalable Peer-to-peer Lookup Service for Internet Applications. In *Proc. of the Conf. on Applications, Technologies, Architectures, and Protocols for Computer Communications*, pages 149–160, San Diego, CA, USA, 2001.

[SRM05] H. Su, E. A. Rundensteiner, and M. Mani. Semantic Query Optimization for XQuery over XML Streams. In *Proc. of the Intl. Conf. on Very Large Data Bases*, Trondheim, Norway, August 2005.

[SSDN02] M. T. Schlosser, M. Sintek, S. Decker, and W. Nejdl. HyperCuP – Hypercubes, Ontologies, and Efficient Search on Peer-to-Peer Networks. In *Proc. of the Intl. Workshop on Agents and Peer-to-Peer Computing*, pages 112–124, Bologna, Italy, July 2002.

[SWK+02] A. Schmidt, F. Waas, M. L. Kersten, M. J. Carey, I. Manolescu, and R. Busse. XMark: A Benchmark for XML Data Management. In *Proc. of the Intl. Conf. on Very Large Data Bases*, pages 974–985, Hong Kong, China, August 2002.

[Tar72] R. E. Tarjan. Depth-First Search and Linear Graph Algorithms. *SIAM J. Comput.*, 1(2):146–160, 1972.

[TEO01] K.-L. Tan, P.-K. Eng, and B. C. Ooi. Efficient Progressive Skyline Computation. In *Proc. of the Intl. Conf. on Very Large Data Bases*, pages 301–310, Roma, Italy, September 2001.

[TH04] I. Tatarinov and A. Halevy. Efficient Query Reformulation in Peer Data Management Systems. In *Proc. of the ACM SIGMOD Intl. Conf. on Management of Data*, pages 539–550, Paris, France, June 2004.

[TIM+03] I. Tatarinov, Z. G. Ives, J. Madhavan, A. Y. Halevy, D. Suciu, N. N. Dalvi, X. Dong, Y. Kadiyska, G. Miklau, and P. Mork. The Piazza Peer Data Management Project. *ACM SIGMOD Record*, 32(3):47–52, 2003.

[TMSF03] P. A. Tucker, D. Maier, T. Sheard, and L. Fegaras. Exploiting Punctuation Semantics in Continuous Data Streams. *IEEE Trans. Knowledge and Data Engineering*, 15(3):555–568, 2003.

[UF99] T. Urhan and M. J. Franklin. XJoin: Getting Fast Answers From Slow and Bursty Networks. Technical report CS-TR-3994, University of Maryland, College Park, February 1999.

[VAB+99] W. Voges, B. Aschenbach, Th. Boller, H. Bräuninger, U. Briel, W. Burkert, K. Dennerl, J. Englhauser, R. Gruber, F. Haberl, G. Hartner, G. Hasinger, M. Kürster, E. Pfeffermann, W. Pietsch, P. Predehl, C. Rosso, J. H. M. M. Schmitt, J. Trümper, and H. U. Zimmermann. The ROSAT All-Sky Survey Bright Source Catalogue. *Astronomy and Astrophysics*, 349(2):389–405, July 1999.

[W3C04a] W3C. Extensible Markup Language (XML) 1.0 Third Edition (W3C Recommendation 4 February 2004), 2004. http://www.w3.org/TR/2004/REC-xml-20040204/.

[W3C04b] W3C. XML Schema Part 0: Primer Second Edition (W3C Recommendation 28 October 2004), 2004. http://www.w3.org/TR/xmlschema-0/.

[W3C05a] W3C. XML Query Use Cases (W3C Working Draft 4 April 2005), 2005. http://www.w3.org/TR/xquery-use-cases/.

[W3C05b] W3C. XQuery 1.0: An XML Query Language (W3C Working Draft 04 April 2005), 2005. http://www.w3.org/TR/xquery/.

[W3C05c] W3C. XQuery 1.0 and XPath 2.0 Formal Semantics (W3C Working Draft 4 April 2005), 2005. http://www.w3.org/TR/xquery-semantics/.

[W3C05d] W3C. XQuery 1.0 and XPath 2.0 Functions and Operators (W3C Working Draft 4 April 2005), 2005. http://www.w3.org/TR/xpath-functions/.

[W3C05e] W3C. XSL Transformations (XSLT) Version 2.0 (W3C Working Draft 4 April 2005), 2005. http://www.w3.org/TR/xslt20/.

[WA91] A. N. Wilschut and P. M. G. Apers. Dataflow Query Execution in a Parallel Main-Memory Environment. In *Proc. of the Intl. Conf. on Parallel and Distributed Information Systems, Miami Beach, Florida, USA*, pages 68–77, December 1991.

[YG02] Y. Yao and J. Gehrke. The Cougar Approach to In-Network Query Processing in Sensor Networks. *ACM SIGMOD Record*, 31(3):9–18, September 2002.

[YG03] Y. Yao and J. Gehrke. Query Processing for Sensor Networks. In *Proc. of the Conf. on Innovative Data Systems Research*, Asilomar, CA, USA, January 2003.

[YGM03] B. Yang and H. Garcia-Molina. Designing a Super-Peer Network. In *Proc. of the IEEE Intl. Conf. on Data Engineering*, pages 49–60, Bangalore, India, March 2003.

[YKL97] Y. Yang, K. Karlapalem, and Q. Li. Algorithms for Materialized View Design in Data Warehousing Environment. In *Proc. of the Intl. Conf. on Very Large Data Bases*, pages 136–145, Athens, Greece, August 1997.

[ZHS+04] B. Y. Zhao, L. Huang, J. Stribling, S. C. Rhea, A. D. Joseph, and J. Kubiatowicz. Tapestry: A Resilient Global-Scale Overlay for Service Deployment. *IEEE Journal on Selected Areas in Communications*, 22(1):41–53, January 2004.