<table>
<thead>
<tr>
<th><strong>Title</strong></th>
<th>A Native and Adaptive Approach for Linked Stream Data Processing</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Author(s)</strong></td>
<td>Le Phuoc, Danh</td>
</tr>
<tr>
<td><strong>Publication Date</strong></td>
<td>2013-04-08</td>
</tr>
<tr>
<td><strong>Item record</strong></td>
<td><a href="http://hdl.handle.net/10379/3589">http://hdl.handle.net/10379/3589</a></td>
</tr>
</tbody>
</table>

Some rights reserved. For more information, please see the item record link above.
A Native and Adaptive Approach for Linked Stream Data Processing

Danh Le Phuoc

Submitted in fulfillment of the requirements for the degree of
Doctor of Philosophy

SUPERVISORS:
Prof. Manfred Hauswirth
Dr. Josiane Xavier Parreira

Digital Enterprise Research Institute, National University of Ireland, Galway
February 2013
The studies presented in this thesis were performed at the Digital Enterprise Research Institute at the National University of Ireland, Galway. The research was financially supported by Science Foundation Ireland under Grant No. SFI/08/CE/I1380 (Lion-II).
©2013 Danh Le Phuoc. All rights reserved.
"I have not failed. I’ve just found 10,000 ways that won’t work."

Thomas A. Edison
Abstract

Sensors, mobile devices and social platforms generate an immense amount of stream data in various formats and schemata. For these areas, the idea of Linked Stream Data is to extend RDF data model to cope with the heterogeneity of data sources and to enable the data integration—not only among themselves, but also with other existing sources. This would enable a vast range of new, near real-time applications. Such applications drive the demand for processing engines that support continuous queries over Linked Stream Data and Linked Data. These engines must not only support the necessary functionalities but also meet the typical low-latency response requirement of stream processing applications.

Since unmodified data stream management systems (DSMSs) and triple storages do not provide full functionalities required by Linked Stream Data processing, the rewriting approach could be used to delegate the processing to those systems. However, this suffers from the overhead of data transformation and does not enable full control over the query execution process. The overhead might be prohibitively expensive for the low-latency response requirement and the lack of full control of the execution process restricts optimisations partially and locally in each underlying sub-system. Moreover, the graph-based model of RDF data poses many challenges to designing a physical storage and optimising the processing when mapped to a relation-based data model. Nevertheless, most techniques and algorithms of DSMSs assume stream data being represented in that way. Therefore, algorithms and techniques for DSMSs and triple stores need to be carefully re-engineered to build an efficient and scalable processing engine for Linked Stream Data and Linked Data.

In this work, we present an adaptive and native execution framework for Linked Stream Data and Linked Data, called CQELS (Continuous Query Evaluation over Linked Streams). The framework introduces one of the first continuous query languages over Linked Stream Data and Linked Data which is compatible with SPARQL 1.1. The flexibility of our execution framework enables performance gains of several orders of magnitudes over other related systems. For dealing with large RDF datasets and high update throughput RDF streams, we propose an efficient hybrid physical data organisation using novel data structures that support algorithms for efficient incremental evaluation of continuous query operators over Linked Stream Data. The framework also provides several adaptive optimisation algorithms. To demonstrate the advantages of the framework and of the CQELS processing engine in terms of performance, the thesis provides extensive experimental evaluations. The evaluations cover a comprehensive set of parameters that dictate the performance of a continuous queries over Linked Stream Data and Linked Data.
Acknowledgments

I would like to dedicate this thesis to my father who has been sacrificing all his life to give me the education he did not have. He is always my biggest fan for whatever I do and achieve. He has been by my side for whatever decision I made. My mother receives my deepest gratitude for raising me, giving me the best care I could have as the only son in the family. To my beloved sisters who always love me unconditionally, thank you all for taking care of mom and dad so that I can pursue my career.

I am extremely grateful to my two PhD advisors, Prof. Manfred Hauswirth and Dr. Josiane Xavier Parreira, for their guidance and help. Manfred has been a wonderful teacher who brought me to the amazing world of research. He always shows me how to bring my idea up to a whole new level. He also has been a big supporter that provides me all resources I need to pursue new research interests. Josiane has been a very dedicated advisor who was always patient with the careless mistakes which I make as a regular basis. Working under her advising with care makes me have a very safe feeling that I have someone watching my back. I am really lucky to have you two in my career path.

My thanks to my co-authors who I had the pleasure to work with: Minh Dao-Tran, Giovanni Tummarelo, Axel Polleres, Christian Morbidoni, Thomas Eiter, Michael Fink, Duc Pham Minh, Shu Lei and Vinny Reynolds. I had a very great time writing my first conference paper with Giovanni, Axel and Christian. I learnt a lot from you guys during this useful and fun collaboration which later had a significant impact on my publishing achievements. I am glad that I got to know Minh, he has been one of my favourite co-authors so far.

Last but not least thanks to my colleges who I have the chance to work with: Hoan Nguyen Mau Quoc, Anh Le Tuan, Martin Serrano, Hugo Hromic, Mario Arias, Nigel Flynn. Your help and collaboration played an important role in my PhD and career. Thanks for my dear friend Helena Deus who gave me advices on how to deal with pressure and distraction in writing thesis.
# Contents

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Abstract</td>
<td>v</td>
</tr>
<tr>
<td>Acknowledgments</td>
<td>vi</td>
</tr>
<tr>
<td>Contents</td>
<td>vii</td>
</tr>
<tr>
<td>List of Figures</td>
<td>xi</td>
</tr>
<tr>
<td>List of Tables</td>
<td>xiii</td>
</tr>
<tr>
<td>1 Introduction</td>
<td>1</td>
</tr>
<tr>
<td>1.1 Motivation</td>
<td>1</td>
</tr>
<tr>
<td>1.2 Problem Statement</td>
<td>3</td>
</tr>
<tr>
<td>1.3 Thesis Contributions</td>
<td>4</td>
</tr>
<tr>
<td>1.4 Motivating Scenarios</td>
<td>5</td>
</tr>
<tr>
<td>1.5 Publications</td>
<td>6</td>
</tr>
<tr>
<td>1.6 Thesis Outline</td>
<td>7</td>
</tr>
<tr>
<td>2 Background</td>
<td>9</td>
</tr>
<tr>
<td>2.1 Basic Concepts and Techniques for Stream Processing</td>
<td>9</td>
</tr>
<tr>
<td>2.1.1 Data Stream Models</td>
<td>9</td>
</tr>
<tr>
<td>2.1.2 Continuous Semantics</td>
<td>10</td>
</tr>
<tr>
<td>2.1.3 Time Management</td>
<td>13</td>
</tr>
<tr>
<td>2.1.4 Implementation of Operators over Streams</td>
<td>14</td>
</tr>
<tr>
<td>2.1.5 Handling Memory Overflow</td>
<td>18</td>
</tr>
<tr>
<td>2.1.6 Optimisation</td>
<td>19</td>
</tr>
<tr>
<td>2.1.7 Scheduling</td>
<td>20</td>
</tr>
<tr>
<td>2.2 Linked Data Processing</td>
<td>21</td>
</tr>
<tr>
<td>2.2.1 RDF and SPARQL</td>
<td>21</td>
</tr>
</tbody>
</table>
# CONTENTS

2.2.2 Physical Storage of RDF Data ........................................... 23  
2.2.3 SPARQL Query Processing ............................................. 25

3 State of The Art in Linked Stream Data Processing ........ 29  
3.1 Data Model and Query Semantics ................................. 29  
  3.1.1 Data Model .................................................. 29  
  3.1.2 Query Semantics ............................................ 30  
3.2 Query Languages .................................................. 32  
3.3 Architectural Design .............................................. 35  
3.4 System Design and Implementation ............................ 38  
3.5 Summary ......................................................... 43

4 A Model for Linked Stream Data Processing .............. 45  
4.1 Data Model and Query Semantics ................................. 45  
4.2 CQELS Query Language ........................................... 48  
4.3 Adaptive Query Execution Model ............................... 50

5 CQELS - Continuous Query Evaluation over Linked Streams 53  
5.1 CQELS Execution Framework .................................... 53  
  5.1.1 Input Manager ................................................. 55  
  5.1.2 Data Encoder and Decoder .................................. 55  
  5.1.3 Adaptive Optimiser ......................................... 56  
  5.1.4 Dynamic Executor ........................................... 57  
  5.1.5 Cache Manager and Cache Fetcher ......................... 58  
5.2 Framework Implementation ......................................... 59  
  5.2.1 A CQELS Engine Implementation ............................. 59  
  5.2.2 Performance Evaluation .................................... 61  
5.3 Discussion .......................................................... 64

6 Data Structures and Algorithms for Incremental Evaluation of Triple-based Windowing Operators 65  
6.1 Introduction .......................................................... 65  
6.2 Operator-aware Data Structures .................................. 66  
  6.2.1 Mappings ..................................................... 66  
  6.2.2 Bags of Mappings ............................................ 68  
  6.2.3 Indexes on Bags of Mappings ............................... 69  
  6.2.4 Experimental Results ..................................... 71  
6.3 Incremental Evaluation for Sliding Windows ................ 74  
  6.3.1 Stateless Operators ......................................... 74
<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>6.3.2 Handling the Expiration in Stateful Operators with a Tree-based</td>
<td>75</td>
</tr>
<tr>
<td>Data Structure</td>
<td></td>
</tr>
<tr>
<td>6.3.3 Multiway Join</td>
<td>77</td>
</tr>
<tr>
<td>6.3.4 Aggregate</td>
<td>80</td>
</tr>
<tr>
<td>6.3.5 DISTINCT</td>
<td>83</td>
</tr>
<tr>
<td>6.3.6 Minus</td>
<td>85</td>
</tr>
<tr>
<td>7 Adaptive Query Optimisation</td>
<td>89</td>
</tr>
<tr>
<td>7.1 Adaptive Cost-based Optimisation for Multiway Joins</td>
<td>89</td>
</tr>
<tr>
<td>7.1.1 One-step Adaptation</td>
<td>90</td>
</tr>
<tr>
<td>7.1.2 Two-step(^+) Adaptation</td>
<td>91</td>
</tr>
<tr>
<td>7.1.3 Experimental Results</td>
<td>93</td>
</tr>
<tr>
<td>7.2 Optimisation of Shared Window Joins</td>
<td>94</td>
</tr>
<tr>
<td>7.2.1 Multiple Join Operator</td>
<td>95</td>
</tr>
<tr>
<td>7.2.2 Networks of Multiple Join Operators for Shared Window Joins</td>
<td>96</td>
</tr>
<tr>
<td>7.2.3 Experimental Results</td>
<td>98</td>
</tr>
<tr>
<td>8 Evaluations</td>
<td>101</td>
</tr>
<tr>
<td>8.1 Evaluation Setup</td>
<td>101</td>
</tr>
<tr>
<td>8.1.1 Data Schema and Test Data Generator</td>
<td>101</td>
</tr>
<tr>
<td>8.1.2 Experimental Setup</td>
<td>103</td>
</tr>
<tr>
<td>8.2 Evaluations</td>
<td>105</td>
</tr>
<tr>
<td>8.2.1 Functionality Tests</td>
<td>105</td>
</tr>
<tr>
<td>8.2.2 Basic Throughput Tests</td>
<td>106</td>
</tr>
<tr>
<td>8.2.3 Scalability Tests</td>
<td>106</td>
</tr>
<tr>
<td>8.2.4 Impact of Dictionary Throughput on The Performance</td>
<td>109</td>
</tr>
<tr>
<td>8.3 Summary</td>
<td>110</td>
</tr>
<tr>
<td>9 Conclusion and Future work</td>
<td>111</td>
</tr>
<tr>
<td>9.1 Conclusion</td>
<td>111</td>
</tr>
<tr>
<td>9.2 Future work</td>
<td>112</td>
</tr>
<tr>
<td>A Evaluation Queries for Social Network Scenario</td>
<td>115</td>
</tr>
<tr>
<td>B Dictionary Throughput Evaluations</td>
<td>119</td>
</tr>
<tr>
<td>Bibliography</td>
<td>125</td>
</tr>
</tbody>
</table>
List of Figures

2.1 Recommended architecture for time management. .......................... 14
2.2 Operator implementations ....................................................... 16

3.1 An operator graph of C-SPARQL query. ................................. 31
3.2 The whitebox architecture. .................................................... 36
3.3 The blackbox architecture. .................................................... 37
3.4 Architecture of C-SPARQL. .................................................. 39
3.5 Evaluation process of C-SPARQL. ........................................... 40
3.6 Architecture of SPARQL_stream .............................................. 41

4.1 Possible execution plans for the query Q2. ............................... 51
4.2 Choosing execution plan dynamically at time 0 and 1. .................. 52

5.1 Adaptive Execution Framework of CQELS. ............................... 54
5.2 Left-deep data flows for the query in the localisation scenario. .... 60
5.3 Average query execution time for varying sizes of simulated DBLP dataset. ................................. 63
5.4 Average query execution time when running multiple query instances. ........................................... 64

6.1 Trees of mappings. ............................................................... 67
6.2 Expiration checking. ............................................................. 68
6.3 Indices for bags of mappings. ................................................ 69
6.4 Insert throughput with different number of keys. ......................... 72
6.5 Probing time of an AVL/Red-black Tree/hash index on a 1 million mapping window. ................................. 73
6.6 Insert throughput on windows with hash index. ......................... 73
6.7 Probing throughput on windows with hash index. ....................... 74
6.8 Invalidate expired mappings. ............................................... 75
6.9 Multiway join process. ......................................................... 77
6.10 Join throughput of binary joins. ........................................... 79
6.11 Aggregate throughput on a window with one million mappings. .... 82
6.12 Aggregate throughput with binary join. .......................... 83
6.13 Throughputs of the DISTINCT operator. .......................... 85
6.14 Throughputs of Minus operator. .................................. 87

7.1 Join graphs. .......................................................... 91
7.2 Join throughputs of optimisation algorithms. ...................... 94
7.3 Shared Windows Example. ......................................... 96
7.4 A Network of Multiple Join Operators. ............................ 96
7.5 A probing graph. ..................................................... 97
7.6 Join throughputs of multiple query optimisation. .................. 99

8.1 Logical schema of the stream data in a social network. ............ 102
8.2 Evaluation design. ................................................. 104
8.3 Scalability when varying the static data size. ....................... 107
8.4 Scalability when varying the window size. ......................... 108
8.5 Scalability when varying the number of multiple queries. ......... 109

B.1 Encoding time of simulated post stream. .......................... 119
B.2 Decoding time of simulated post stream. .......................... 120
B.3 Encoding time of simulated Photo stream. ........................ 120
B.4 Decoding time of simulated Photo stream. ........................ 121
B.5 Encoding time of simulated GPS stream. .......................... 121
B.6 Decoding time of simulated GPS stream. .......................... 122
B.7 Encoding time of Billion Triple Challenge 2011 dataset. ......... 122
B.8 Decoding time of Billion Triple Challenge 2011 dataset. ......... 123
# List of Tables

<table>
<thead>
<tr>
<th>Table</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>3.1</td>
<td>Current systems that implement the blackbox architecture.</td>
<td>37</td>
</tr>
<tr>
<td>3.2</td>
<td>System design choices.</td>
<td>37</td>
</tr>
<tr>
<td>3.3</td>
<td>System comparison by features.</td>
<td>42</td>
</tr>
<tr>
<td>3.4</td>
<td>Comparisons by execution mechanism.</td>
<td>43</td>
</tr>
<tr>
<td>4.1</td>
<td>Input and output of Q as time progresses.</td>
<td>51</td>
</tr>
<tr>
<td>5.1</td>
<td>Average query execution time for single queries (in milliseconds).</td>
<td>62</td>
</tr>
<tr>
<td>8.1</td>
<td>Queries classification.</td>
<td>105</td>
</tr>
<tr>
<td>8.2</td>
<td>Maximum execution throughput of CQELS, C-SPARQL and JTALIS for different queries.</td>
<td>106</td>
</tr>
<tr>
<td>8.3</td>
<td>Summary of throughputs (triples/second): CQELS queries, query operators (join, aggregate, distinct, minus), and encoding/decoding operations (1000 triples/sec).</td>
<td>110</td>
</tr>
</tbody>
</table>
Chapter 1

Introduction

1.1 Motivation

We are witnessing a paradigm shift, where real-time, time-dependent data is becoming ubiquitous. Sensor devices were never so popular. For example, mobile phones (accelerometer, compass, GPS, camera, etc.), weather observation stations (temperature, humidity, etc.), patient monitoring systems (heart rate, blood pressure, etc.), location tracking systems (GPS, RFID, etc.), buildings management systems (energy consumption, environmental conditions, etc.), and cars (engine monitoring, driver monitoring, etc.) are continuously producing an enormous amount of information in the form of data streams. Also on the Web, services like Twitter, Facebook, and blogs deliver streams of (typically unstructured) real-time data on various topics.

Integrating these new information sources—not only among themselves, but also with other existing sources—would enable a wide range of new near real-time applications in the areas of smart cities, green IT, e-health, etc. However, due to the heterogeneous nature of such diverse streams, harvesting the data is still a difficult and labor-intensive task, which currently requires a lot of “hand-crafting.” To remedy this, the RDF data model could be applied. This data model enables expressing knowledge in a generic manner, without requiring adherence to a specified schema [Owens, 2011].

There have been efforts to lift stream data to a semantic level, e.g., by the W3C Semantic Sensor Network Incubator Group\(^1\) and Semantic Sensor/Stream [Bouillet et al., 2007, Sheth et al., 2008, Whitehouse et al., 2006]. The goal is to make stream data available according to the Linked Data principles [Bizer et al., 2009a]—a concept that is known as Linked Stream Data [Sequeda and Corcho, 2009]. As Linked Data facilitates the data integration process among heterogenous collections, Linked Stream Data has the same goal with respect to data streams. Moreover, it also bridges the gap between stream and more static data sources.

Besides a unified data representation model, processing engines that support continuous queries on both Linked Stream Data and Linked Data are required. In classical Linked Data processing, data is assumed to change infrequently and stored in a centralised repository before further processing. Updates on a dataset are usually limited to a small fraction of the dataset and occur infrequently, or the whole dataset is replaced by a new version. Query processing in Linked Data databases, as in

\(^1\)http://www.w3.org/2005/Incubator/ssn/
CHAPTER 1. INTRODUCTION

traditional relational databases, is \textit{pull} based and \textit{one-time}, i.e., the data is read from the disk, the query is executed against it once, and the output is a set of results for that point in time. In contrast, in Linked Stream Data, new data items are produced continuously, the data is often valid only during a time window, and it is continually \textit{pushed} to the query processor. Queries are continuous, i.e., they are registered once and then are evaluated continuously over time against the changing dataset. The results of a continuous query are updated as new data appears. Such temporal aspect of Linked Stream Data and the continuity of continuous queries are not considered in Linked Data query processing engines at the moment.

Data stream management systems (DSMSs) seem to be better candidates for processing continuous queries. A DSMS can be used as a sub-component to deal with stream data. However, none of the traditional DSMSs support RDF, therefore a data transformation step is required. This overhead of data transformation might be prohibitively expensive in the context of low-latency processing of stream data. Moreover, delegating processing to a sub-system like a DSMS means losing full control over the query execution. Subsequently, the optimisation only can be done locally in each sub-system. As a sub-system is used as a blackbox, it is only optimised for the data model, query patterns and data distributions that they are designed for.

The hard-to-predict structure of RDF graphs has proved challenging for traditional DBMSs, and they have not been able to scale effectively to large quantities of RDF data [Owens, 2011]. This unpredictivity also applies to RDF-based data streams. As a consequence, it makes it difficult for DSMS query optimisers to handle. On the other hand, optimisation problems of DSMS have been solved in some restricted cases and ad-hoc scenarios, but there are still a lot of open challenges and problems [Deshpande et al., 2007, Golab and Öszu, 2010]. Additionally, most of the optimisation algorithms such as [Chen et al., 2002, Viglas and Naughton, 2002, Ayad and Naughton, 2004, Tao et al., 2005, Krishnamurthy et al., 2006, Wang et al., 2006] are heuristic and only proven to work for certain types of queries and data.

These facts motivated us to develop a native and adaptive approach for processing Linked Data Streams. Our approach aims at building high performance processing engines for Linked Stream Data by combining and re-engineering efficient data structures, algorithms and techniques from both Linked Data processing and traditional DSMS. As shown in several work on the physical representation of RDF data [Broekstra et al., 2002, Wilkinson et al., 2003, Chong et al., 2005, Harth et al., 2007, Abadi et al., 2007, Weiss et al., 2008, Neumann and Weikum, 2010], relational tables are not ideal for storing RDF data elements. Careful design of physical storage and indexing schema is vital for the performance of the triple storages [Abadi et al., 2007, Weiss et al., 2008, Neumann and Weikum, 2010]. Therefore, our approach aims at designing a native data structure that treats RDF and RDF stream data elements as first class citizens. Adaptivity is required in the context of processing continuous queries because the data constantly changes during the query life time. To support the adaptivity in the query execution, we introduce an adaptive execution framework, called CQELS (Continuous Query Evaluation for Linked Stream). The framework is designed for applying adaptive query processing techniques [Deshpande et al., 2007] to meet the performance requirements of stream processing. This framework enables full control of the continuous execution process where the scheduling and optimisation can be done at runtime.

As one of the first works in Linked Stream Data processing, we had to create a new continuous query language for processing Linked Stream Data. To provide an insight into how to build an efficient Linked Stream Data processing, we conducted the first survey and evaluation of Linked Stream Data processing engines that were developed during the time this thesis.
1.2 Problem Statement

The motivation of the thesis leads to the broader research problems that arise when building an efficient query processing engine for Linked Stream Data. The first challenging problem is designing a new declarative query language because neither the state of the art continuous query languages nor SPARQL could be used to query Linked Stream Data. The query language needs a formal data model and sound semantics of continuous query operators. The data model needs to be able to represent not only Linked Stream Data but also Linked Data in a unified view. Therefore, the new data model has to be an extension of the RDF model to transparently integrate conventional RDF datasets. In the context of continuous query processing, the temporal aspect of the data is a required property that has not been covered by any RDF-based data model before. Along with the data model, the graph-based query operators with continuous semantics have to be defined to specify the meanings of the declarative query patterns. It is desirable that the new query language is similar to SPARQL to reduce the learning effort. Consequently, the query operators should be aligned with the semantics of SPARQL [Pérez et al., 2009b] but must be compatible with windowing operations as defined in traditional continuous queries such as CQL [Arasu et al., 2006].

Given the drawbacks of using unmodified DSMSs and triple storages for Linked Stream Data, RDF-based stream data exhibits the new challenges for the physical organisation for both Linked Stream Data and Linked Data. The standard model for storing a bag of RDF triples is that of a triple table storing identifiers representing URIs and literals, combined with mapping tables as dictionaries to translate these identifiers back into their lexical form [Abadi et al., 2007, Owens, 2011]. However, this data layout is designed for heavily read-intensive contexts [Harth et al., 2007, Weiss et al., 2008, Neumann and Weikum, 2010] whilst Linked Stream Data needs high writing throughput. The DSMSs remedy the write-intensive requirement by using in-memory storage however our data model involves Linked Data that might be impossible to host in the main memory. Moreover, RDF-based data elements, e.g., RDF triples and temporal RDF triples, are very small. They thus exhibit unusually large individual data points compared to the amount of information encoded. Consequently, the row-based data structure used in relational DSMSs is inefficient because it requires tuple header sizes that might dominate the total storage size [Abadi et al., 2007]. In particular, the row-based data structure designed for wider and short tables might increase significantly the memory footprint for stream processing. Thus, a new physical organization approach is needed for processing both Linked Stream Data and Linked Data.

RDF-based continuous query operators typically operate on one or a few very large tables, thus, it is vital to have indexes for fast access to random data items. Most modern RDF stores offer a heavy indexing strategy to overcome their large table handicap [Weiss et al., 2008]. Indeed, it is possible to bypass the table because all the access patterns are covered by their indexes. However, a comprehensive indexing scheme is impractical for stream processing due to its expensive maintenance cost. Indexing solutions for stream data such as [Golab et al., 2004, Golab and Özsu, 2005] might be useful but they are designed for relational streams. Therefore, investigating a hybrid solution that can apply indexing strategies of both triple storages and stream data processing is an interesting problem.

Another challenge associated with the physical representation of RDF-based stream data is how to efficiently evaluate windowing operators vs. the unbounded nature of streams. There have been efforts in DSMS for supporting sliding-window queries. The first approach is to re-evaluate over each window independent from all other windows, called re-evaluation computation. This approach is used in Aurora [Abadi et al., 2003] and Borealis [Abadi et al., 2005]. Another approach called incremental evaluation computation only processes changes (inserted and expired tuples) in the windows in the query pipeline. This approach is used in STREAM [Arasu et al., 2003] and Nile [Hammad et al.,
There have been some shortcomings in employing incremental evaluation methods, i.e., direct-timestamp and negative tuples [Hammad et al., 2003a, Ghanem et al., 2007]. While direct-timestamp method needs extra timestamps, the negative tuple method doubles the number of tuples through the query pipelines. Particularly, with the new data structures introduced in this thesis, the associated efficient algorithms for computing windowing operators must address the unusual data characteristics of RDF streams and RDF datasets.

For RDF triple storages, exceptionally long, thin tables are a nonstandard optimisation case, making it challenging for traditionally DBMSs to produce relevant statistics for query optimiser. While this challenge still applies for processing Linked Stream Data and Linked Data, it is even more challenging to maintaining statistics for highly dynamic datasets in stream processing settings. Due to the dynamic nature of stream data and the unpredictivity of RDF data distributions, the adaptivity of a query optimiser for continuous query processing becomes more difficult to achieve. On top of that, the SPARQL-like queries normally have many shared query patterns that pose the requirement of multiple-query optimisation [Golab and Özsu, 2003b]. Even though there are some efforts in multiple-query optimisation such as [Tok and Bressan, 2002, Dobra et al., 2004, Krishnamurthy et al., 2006, Wang et al., 2006, Sharaf et al., 2008, Golab et al., 2008], however, such approaches proposed for relational streams might not work on the RDF-based stream due to its different natures compared to relational ones. Therefore, it is challenging to enable multiple-query-optimisation for Linked Data Streams.

### 1.3 Thesis Contributions

In the context of the problems stated above, the main contributions of the thesis are:

**Adaptive processing model for Linked Data Stream**

We propose an adaptive processing model that includes formal definitions of the data model, query semantics, and execution model. The data model not only covers Linked Stream Data but also the temporal aspect of Linked Data sets that have not been addressed so far. The query semantics is formalised with both mathematical and operational meanings. The mathematical meanings show how a declarative query fragment is mapped to the corresponding mathematical expressions. All the query fragments are accompanied with abstract syntaxes for defining a declarative query language extended from SPARQL 1.1. The operational meanings define how the operators in the expressions are executed in physical execution plans. The operational semantics formally show our execution model for continuously executing equivalent execution plans for a query expressed in the CQELS language. This operational feature facilitates the adaptivity of the execution engines based on our processing model because it enables the execution engine to dynamically switch from the current execution plan to another equivalent one to adapt to the changes in the run-time. The CQELS language is not only one of the first query language for Linked Stream Data but also the only language which is accompanied with sound mathematical and operational semantics so far.

**Adaptive Execution Framework**

We propose an adaptive execution framework to enable adaptivity in CQELS engines. This framework enables full control of the execution process with the flexibility of adding new data structures and new algorithms to any component of the query engine. The framework employs an encoding mechanism to enable less workload, smaller footprint implementations of operators by performing only on small, fixed
1.4. Motivating Scenarios

Similar to Live Social Semantics [Alani et al., 2009, Szomszor et al., 2010], we are inspired by the scenarios that integrate human-centric streaming data from the physical and digital world. The data from the physical world is captured and streamed via sensors and tracking systems such as GPS, RFID, and wireless triangulation, and can be integrated with virtual streams (e.g., Twitters, city traffic data, airport information) to deliver location-based services or up-to-date view of a particular situation.

Conference scenario: Integrating physical stream with online profiles

This scenario focuses on the data integration problem between data streams from a tracking system and static data sets. Similar to several real deployments in Live Social Semantics, the tracking system is used for gathering the relationship between real-world identifiers and physical spaces of conference attendees. This tracking data can then be correlated with non-stream datasets, like online information about the attendees (social network, online profiles, publication record, etc). The benefits of correlating
these two sources of information are manifold. For instance, conference rooms could be automatically assigned to the talks, based on the talk’s topic and the number of people that might be interested in attending it (based on their profile). Conference attendees could be notified about fellow co-authors in the same location. A service that suggests which talks to attend, based on profile, citation record, and distance between talk locations can be designed. These are just a few examples.

For the tracking service in our example, attendees of a conference wear RFID tags that constantly stream the location in a building, i.e., which room/section/area they currently are. Each reading streamed from the RFID tags to RFID readers has two properties, the *tagid* and the *signal strength*.

The static datasets include data about the conference attendees and metadata about the conference building. Each attendee has a profile containing his personal information and the *tagid* given to her/him. The profile has also data links to the attendee’s publication records in DBLP. The data about the conference building includes information such as location name, description, layout, and connection between location/rooms.

**Social network scenario: Data aggregation for heterogeneous social stream data**

Social stream data of interest for a user are spread among different social application platforms such as Facebook, Twitter, Foursquare. Social network aggregation and analysis platforms like Bootlenose \(^2\) need to integrate heterogeneous streams from different social networks and feeds. Such platforms can easily employ Linked Stream Data processing engines for dealing with data integration issues [Balduini et al., 2012].

Along the same lines, this scenario focuses on the aggregation of different social stream sources created by social network users. Social networks provide rich resources of interesting stream data, such as the sequence of social discussions and photo uploading. As they are viewed as highly-connected graphs of users, social networks can be considered as the best test area for RDF engines where RDF can show its advantage in representing graph data. Moreover, data in real life is correlated with skewed data distributions which happens a lot in the social network data. Efficiently handling correlations is recognised as a difficult problem by database engines, but it also opens up many query optimisation opportunities. We use this scenario for building the data simulator to exploit various realistic skewed data distributions and correlations available in a social network. The data simulator is used to generate realistic test cases for evaluating Linked Stream Data processing engines.

**1.5 Publications**

Several parts of this thesis have been published as conference, workshop and journal articles. The first attempt to build a platform for integrating stream data from sensors using RDF was introduced in [Le-Phuoc and Hauswirth, 2009, Le-Phuoc et al., 2010a]. An extended version of this platform was published in [Le-Phuoc et al., 2012b]. This platform is powered by the processing engine presented in [Le-Phuoc et al., 2011]. The surveys on DSMSs and Linked Stream Processing engines were published in [Le-Phuoc et al., 2010b, 2012c]. The benchmark framework and the evaluation of different Linked Stream Data engines can be found in [Le-Phuoc et al., 2012a].

\(^2\)http://bottlenose.com/
1.6 Thesis Outline

The remainder of this thesis is organised as follows: Chapter 2 gives general background on stream processing and Linked Data processing. Chapter 3 presents a survey of Linked Stream Data processing. Chapter 4 presents the processing model for the CQELS query execution framework introduced in Chapter 5. Novel data structures for the CQELS framework are described in Chapter 6. Chapter 7 presents optimisation solutions for the CQELS framework. The evaluation of the CQELS framework is reported in Chapter 8. Finally, Chapter 9 concludes this thesis and points out future work.
Chapter 2

Background

This chapter provides the background concepts and techniques for stream processing and Linked Data processing. In Section 2.1, we will provide the fundamentals of stream processing applied to Linked Stream Data. We discuss the basic models and techniques, the representation of continuous semantics, the operators and optimisation techniques, and how issues like time management and memory overflow are handled. For the Linked Data processing in Section 2.2, we will present the relevant notations and definitions of the RDF data model and the semantics of SPARQL queries. We also give an overview of how to store RDF data and query it using SPARQL.

2.1 Basic Concepts and Techniques for Stream Processing

2.1.1 Data Stream Models

A data stream is an unbounded, continuously arriving sequence of timestamped stream elements. The stream elements may arrive in some orders [Tucker et al., 2003] or out of order with explicit timestamps [Li et al., 2008]. The stream elements are continuously pushed by stream sources, and their arrival might be unpredictable. As a result, the system processing data streams has no control over the order of the stream elements and the streaming rate. The system is only able to access stream elements sequentially in the order in which they arrive.

The most popular data model used for stream data is the relational model [Arasu et al., 2006, Chandrasekaran et al., 2003, Abadi et al., 2003]. In the relational model, stream elements are relational tuples with a fixed schema. Stream elements can be modelled in an object-based model that classifies stream contents according to a type hierarchy. For example, Tribica [Sullivan and Heybey, 1998] proposes hierarchical data types for representing Internet protocol layers for its network monitoring system. Another example of modelling data sources by objects is the COUGAR system for managing sensor data [Bonnet et al., 2001]. In COUGAR, each type of sensor is modelled as an abstract data type, whose interface consists of the supported signal processing methods. This object-oriented model is also used in complex event processing (CEP) engines such as SASE [Wu et al., 2006, Agrawal et al., 2008], ZStream [Mei and Madden, 2009] and ESPER¹. CEP is closely related to stream processing, but its focus is on deriving high-level knowledge and complex events composed from lower-level events [Eckert et al., 2011], rather than modelling and processing time-dependent information.

¹http://esper.codehaus.org/
In parallel, many dynamic applications are built upon distributed network infrastructures, such as social networks, communication networks, biological networks and the Web. Such applications create data that can be naturally modelled as graph streams, in which edges of the underlying graph are stream elements. For instance, an RDF triple is an edge in an RDF graph. These edges are received and updated sequentially in a form of a stream [Bar-Yossef et al., 2002, Ganguly and Saha, 2006, Das Sarma et al., 2008, Zhao et al., 2011, Bifet et al., 2011]. The work on this thesis combines concepts from these areas. Most of the work in Linked Stream Data reuse operators and notations of the relation model. The terms tuple and stream element might be used alternatively in the following chapters.

2.1.2 Continuous Semantics

A continuous query is issued once and runs continuously, incrementally producing new results over time. Its inputs are one or more append-only data streams and zero or more relations. The continuous semantics of a query Q is defined by the result it returns at each time instant \( t \), denoted as \( Q(t) \). Q is monotonic if \( Q(t) \subseteq Q(t') \), \( \forall t \leq t' \). Arasu et al. [2006] formalised the semantics of monotonic queries and proposed how to continuously evaluate them. For non-monotonic queries, their semantics and execution mechanisms are addressed in [Hammad et al., 2003a, Krämer and Seeger, 2005, Law et al., 2004, Golab and Özsu, 2005].

Intuitively, a continuous query provides answers at any point in time, taking into account all the data that has arrived so far. This data is commonly in the form of relations used as inputs of relational algebras. For instance, Arasu et al. [2006] define that a stream S is a bag of tuples associated with timestamps and an instantaneous relation \( R(t) \) is a bag of tuples of a relation R at a given point time \( t \). Note that in this context, a relation R is defined as a mapping from each time instant to a finite but unbound bag of tuples belonging to the schema of R. Therefore, two types of continuous query algebras based on their relational counterparts have been proposed. The first one is the stream-to-stream algebra that was employed in defining the semantics of Streaming SPARQL[Bolles et al., 2008]. In a stream-to-stream algebra, each operator consumes one or more streams (and zero or more relations) and incrementally produces an output stream [Cranor et al., 2003, Krämer and Seeger, 2009].

The second type is the mixed algebra [Arasu et al., 2006, Ghanem et al., 2008]. A mixed algebra includes three sets of operators: stream-to-relation operators which produce a relation from a stream (e.g., sliding windows), relation-to-relation operators which produce a relation from one or more input relations (i.e., the standard relational algebraic operators), and relation-to-stream operators which produce a stream from a relation. Conceptually, at every time tick, an operator converts its input to relations, computes any new results, and converts the results back to a stream that can be consumed by the next operator. Since the converted relations change over time, a natural way of switching back to a stream is to report the difference between the current result and the result computed one time tick ago. This is similar to computing a set of changes (insertions and/or deletions) required to update a materialised view. The mixed algebra approach is used in formalising the semantics of C-SPARQL [Barbieri et al., 2010b], SPARQL\textsubscript{stream} [Calbimonte et al., 2010] and CQELS [Le-Phuoc et al., 2011].

Stream-to-Stream operators

A Stream-to-Stream operator continuously calls one-time queries in native SQL over physical or logical streams to produce results in a derived stream. These operators are specified by common SQL constructions such as SELECT, FROM, WHERE and GROUP BY. In [Krämer and Seeger, 2009], the window specification is defined by extending the FROM clause. Other logical standard operators are
defined similarly to relational algebras.

Stream-to-Relation operators

A stream-to-relation operator takes a stream \( S \) as input and produces a relation \( R \) as output with the same schema as \( S \). For example, CQL [Arasu et al., 2006] introduces three operators: time-based, tuple-based, and partitioned windows. These operators are later extended to define window operators for our query model in Chapter 4.

1. **Time-based sliding windows:** A time-based sliding window on a stream \( S \) takes a time-interval \( T \) as a parameter and is specified by following the reference to \( S \) with \([\text{Range } T]\). Intuitively, a time-based window defines its output relation over time by sliding an interval of size \( T \) time units capturing the latest portion of an ordered stream. More formally, the output relation \( R \) of \( "S [\text{Range } T]" \) is defined as:

\[
R(t) = \{ s | \langle s, t' \rangle \in S \land (t' \leq t) \land (t' \geq \max\{(t - T + 1), 0\}) \}
\]  
(2.1)

When \( T = 0 \), \( R(t) \) consists of tuples obtained from elements with timestamp \( t \), denoted with the syntax \( "S [\text{NOW}]" \). And when \( T = \infty \), \( R(t) \) consists of all tuples obtained from elements with timestamps up to \( t \), given with the SQL-99 syntax [Melton and Simon, 2001] \( "S [\text{Range Unbounded}]" \).

**Example 1:** “RFIDstream [Range 60 seconds]” represents a time-based sliding window of 60 seconds over a stream of RFID readings. At any time instant \( t \), \( R(t) \) will contain a bag of RFID readings from the previous 60 seconds.

2. **Tuple-based windows:** A tuple-based sliding window on a stream \( S \) takes a positive integer \( N \) as a parameter and is specified by following the reference to \( S \) in the query with \([\text{Rows } N]\). At any given point in time, the window contains the last \( N \) tuples of \( S \). More formally, let \( s_1, s_2, ..., s_n(t) \) denote the tuples of \( S \) in increasing order of their timestamps, breaking ties arbitrarily. The output relation \( R \) of \( "S [\text{Rows } N]" \) is defined as:

\[
R(t) = \{ s_i | \max\{1, n(t) - N + 1\} \leq i \leq n(t) \}
\]  
(2.2)

where \( n(t) \) denotes the size of \( S \) at time \( t \), i.e., the number of elements of \( S \) with timestamps \( \leq t \).

**Example 2:** Similar to Example 1, “RFIDstream [ROWS 1]” returns the last RFID reading from the stream at any time instant.

3. **Partitioned windows:** A partitioned sliding window is applied to a stream \( S \) with two parameters: a positive number \( N \) for number of rows and a subset of attributes of \( S \), \( \{A_1, ..., A_k\} \). The CQL syntax for partitioned windows is \([\text{Partition } S \text{ By } A_1, ..., A_k \text{ Rows } N]\). Similar to SQL’s Group-By, this window operator logically partitions the stream \( S \) into sub-streams based on equality of attributes \( A_1, ..., A_k \). The parameter \( N \) is used to compute the tuple-based windows from those sub-streams.

**Example 3:** “RFIDstream [Partition By tagid ROWS 1]” partitions the RFIDstream into a collection of sub-streams based on \( \text{tagid} \) and gets the latest readings from each sub-stream. This query can be used to find the last locations of all the RFID tags.
The windows might have a slide parameter for specifying the granularity at which the window slides. The slide parameter is a time-interval for time-based windows and a possible integer for row-based and partitioned windows. For example, “RFIDstream [Range 60 minutes Slide 2 minutes]” denotes a 60-minute window over RFIDstream that slides at 2-minute granularity. The formal definition can be found in [Arasu et al., 2006]. Additionally, fixed windows and value-based windows were proposed in [Sullivan and Heybey, 1998] and [Seshadri et al., 1995], respectively.

Relation-to-relation operators

The relation-to-relation operators are introduced to employ relational operators. Therefore, they have the same semantics as their counterparts. However, CQL introduces the concept of instantaneous relations that are relations computable at a specific time instant \( t \), e.g., outputs from stream-to-relation operators.

**Example 4:** The projection (SELECT) and duplicate elimination (Distinct) operators can be applied to the output of the sliding window in Example 1 to create the query below:

```
SELECT DISTINCT tagid
FROM RFIDstream [RANGE 60 seconds]
```

Relation-to-stream operator

A relation-to-stream operator produces a stream from a relation. It takes a relation \( R \) as input and produces a stream \( S \) as output with the same schema as \( R \). For instance, CQL introduces three relation-to-stream operators: \( Istream \), \( Dstream \), and \( Rstream \). We also reuse their definitions to define our streaming operators in Chapter 4.

1. \( Istream \) (for “insert stream”) applied to a relation \( R \) produces a stream element \( \langle s, t \rangle \) whenever the tuple \( s \) is in \( R(t) - R(t - 1) \). Assuming \( R(-1) = \emptyset \) for notational simplicity, it is defined as follow:

\[
Istream(R) = \bigcup_{t \geq 0} (R(t) - R(t - 1)) \times \{t\} \tag{2.3}
\]

**Example 5:** Consider the following CQL query for creating a new stream by filtering another stream:

```
SELECT Istream(*)
FROM RFIDstream [RANGE Unbounded]
WHERE signalstrength >= 85
```

This query continuously applies the unbounded window to the RFIDstream, then filter all the RFID readings that have signal strength values equal or greater than 85.

2. \( Dstream \) (for “delete stream”) applied to a relation \( R \) produces a stream element \( \langle s, t \rangle \) whenever the tuple \( s \) is in \( R(t - 1) - R(t) \). Formally:

\[
Dstream(R) = \bigcup_{t > 0} (R(t - 1) - R(t)) \times \{t\} \tag{2.4}
\]
2.1. BASIC CONCEPTS AND TECHNIQUES FOR STREAM PROCESSING

Example 6: Below is a query to detect when a person leaves the building by tracking the RFID tag of that person. The sliding window keeps all the readings in the last 60 seconds, and the Dstream operator will report the tagid that was not detected in the last 60 seconds but had been detected before.

SELECT Dstream(tagid)
FROM RFIDstream [60 seconds]

3. Rstream (for “relation stream”) applied to a relation R produces a stream element \( \langle s, t \rangle \) whenever the tuple s is in R at time t. Formally:

\[
Rstream(R) = \bigcup_{t \geq 0} (R(t) \times \{t\})
\] (2.5)

Example 7: The query in Example 5 can be written with Rstream as follows:

SELECT Rstream(*)
FROM RFIDstream [NOW]
WHERE signalStrength > 85

2.1.3 Time Management

The described semantics for continuous queries in a data stream system typically assume timestamps on data stream elements. Thus, a consistent semantics for multiple streams and updatable relations relies on timestamps. To achieve semantic correctness, the DSMS query processor usually needs to process tuples in increasing order of timestamps. That is, the query processor should never receive a stream element with a lower timestamp than any previously received ones. According to [Srivastava and Widom, 2004], there are two common types of timestamps: system timestamp and application timestamp.

A system timestamp is assigned to a stream element when entering the DSMS using the DSMS system time. An application timestamp is given by the data sources before sending the stream elements to the DSMS. As an example of application timestamps, consider monitoring sensor readings to correlate changes in temperature and pressure. Each tuple consists of a sensor reading and an application timestamp affixed by the sensor, denoting the time at which that reading was taken. In general, there may not be any relationship between the time at which the reading is taken (the application timestamp) and the time at which the corresponding stream tuple reaches the DSMS (the system timestamp).

The recommended architecture for time management is shown in Figure 2.1 [Srivastava and Widom, 2004]. As stream tuples may not arrive at the DSMS in increasing timestamp order, there is an input manager that buffers tuples until they can be moved to the query processor in proper order. Continuous queries (CQ₁, CQ₂, . . . , CQₘ) registered to the query processor consume ordered stream tuples. The decision when a tuple can be moved to the query processor is based on heartbeats. A heartbeat for a set of streams S₁, S₂, . . . , Sₙ at wall-clock time \( c \) is defined as the maximum application timestamp \( \tau \) such that all tuples arriving on S₁, S₂, . . . , Sₙ after time \( c \) must have timestamp > \( \tau \). Streams are generated by possibly distributed sources \( \phi₁, \phi₂, \ldots, \phiₙ \). Streams are transmitted from their respective
sources to the DSMS over a network which may have some transmission latencies, upper-bounded by $L_1$, $L_2$, \ldots, $L_n$ respectively for each stream.

![Figure 2.1: Recommended architecture for time management.](image)

Along with the proposal for generating heartbeats by the same authors [Srivastava and Widom, 2004], there are also other proposals for time management from other data stream management projects, such as Aurora [Abadi et al., 2003], Niagara [Chen et al., 2000], TelegraphCQ [Chandrasekaran et al., 2003], and Gigascope [Cranor et al., 2003]. The operators of Aurora have a slack parameter to deal with out-of-order streams. Essentially, the slack parameter instructs its operator to wait a certain period of time before closing each window. In Niagara, the proposed solution is based on punctuations [Tucker et al., 2003]. Punctuations define arbitrary predicates over streams. Thus, heartbeats can be thought of special types of punctuations. A more detailed comparison of heartbeat solutions can be found in [Srivastava and Widom, 2004].

### 2.1.4 Implementation of Operators over Streams

For continuously executing operators over streams, there are two main execution strategies: **eager execution** and **periodic execution** [Golab and Özsu, 2003a]. The eager execution strategy generates new results every time a new stream element arrives. However, this might be infeasible in situations where streams have high arrival rates. The periodic execution executes the query periodically [Arasu and Widom, 2004a, Chandrasekaran and Franklin, 2003]. In this case, sliding windows may be advanced and queries are evaluated periodically with a specified frequency [Abadi et al., 2003, Chen et al., 2000, Krishnamurthy et al., 2006, Li et al., 2005, Liu et al., 1999, Shivakumar and García-Molina, 1997, Chandrasekaran et al., 2003, Zhang et al., 2004]. A disadvantage of periodic query evaluation is that results may be stale if the frequency of re-executions is lower than the frequency of the updates.
One way to stream new results after each new item arrives is to bound the error caused by delayed expiration of tuples in the oldest sub-window. However, long delays might be unacceptable in streaming applications that must react quickly to unusual patterns in data.

A continuous query contains sliding windows to deal with unbounded nature of data streams. When the window slides, the query is continuously computed to reflect both new tuples entering the windows and old tuples expiring from the window. Two methods for continuous computation are query re-evaluation and incremental evaluation. In the query re-evaluation method, two consecutive evaluations of a query are carried out in two independent query pipelines. For example, Aurora [Abadi et al., 2003] and Borealis [Abadi et al., 2005] use the re-evaluation method. On the other hand, the query pipeline of the query incremental evaluation method only processes changes in the window to produce new answer when the window slides. The incremental operators are used in the pipeline to process both new and expired tuples to produce incremental changes to the output stream of a query [Ghanem et al., 2007]. The query incremental evaluation method is used in STREAM [Arasu et al., 2003] and Nile [Hammad et al., 2004]. The query re-evaluation method is easier to implement compared to the query incremental evaluation because it can reuse operator implementations of relational database. The query re-evaluation method will be used and evaluated in Chapter 5. However, the query incremental evaluation method is widely adopted in stream processing systems because it can avoid redundant computation to deliver better performance. Therefore, we will propose new data structures and algorithms to support the query incremental evaluation method in Linked Stream Data processing in Chapter 6.

The incremental query evaluation needs to handle two types of events: arrivals of new stream elements and expirations of old stream elements [Golab, 2006]. The actions taken upon arrival and expiration vary across operators [Hammad et al., 2003a, Vossough and Getta, 2002]. A new stream element may generate new results (e.g., join) or remove previously generated results (e.g., negation). Furthermore, an expired stream element may cause a removal of one or more items from the result (e.g., aggregation) or an addition of new items to the result (e.g., duplicate elimination and negation). Moreover, operators that must explicitly react to expired elements (by producing new results or invalidating existing results) have to perform state purging eagerly (e.g., duplicate elimination, aggregation, and negation), whereas others may do so eagerly or lazily (e.g., join).

The new stream element arrivals are triggered by stream sources. However, there should be mechanisms to signal expiration events. There are two main techniques to signal expirations: direct timestamp [Arasu et al., 2006, Golab, 2006], and negative tuple [Arasu et al., 2006, Golab, 2006, Golab and Özsu, 2010]. In the negative tuple technique, every window in the query is equipped with an operator to explicitly generate a negative tuple for every expiration on this window. The negative tuples are used to signal expirations by propagating them through the query pipeline to trigger the operators to invalidate expired tuples in their state. The basic idea of this technique is attractive but may not be practical due to the fact that generating a negative tuple for every expired input doubles the number of tuples through the query pipeline. Therefore, the overhead of processing tuples through various operators such as join is doubled [Ghanem et al., 2007]. For queries without negation operations, base tuples or intermediate results are associated with extra timestamps to explicitly specify when the tuples are expired, called expiration timestamps. Therefore, the expiration of each tuple can be checked directly based on the expiration timestamp. This technique does not incur the overhead of negative tuples and does not have to store the base windows referenced in the query. However, as shown in [Hammad et al., 2003a, Ghanem et al., 2007], this technique might cause some incorrectnesses in the output results and it might be slower than the negative tuple technique. In the context of RDF-based stream processing, shortcomings are discussed in Chapter 6.

The complexity of an incremental evaluation depends also on the type of the operators. For stateless
operators, re-execution can be done on-the-fly without having to maintaining any processing state. For instance, Figure 2.2(a) shows how the selection operation over a stream S1 works [Golab and Özsu, 2010]. The duplicate-preserving projection and union operators are also examples of stateless operators. In contrast to stateless operators, a stateful operator needs to probe the previous processing states in every incremental computation step. Maintaining processing state is done differently for each type of operator. Next, we will discuss how to deal with stateful operators such as window join, aggregation, duplication elimination and non-motonic operators.

![Operator implementations](image)

**Figure 2.2:** Operator implementations: selection (a), window join (b), duplication elimination (c), aggregation (d), and negation (e).

**Window join operators**

In a sliding window join, newly arrived tuples on one of the inputs probe\(^2\) the state of the other inputs. Additionally, expired tuples are removed from the state [Golab and Özsu, 2003a, Hammad et al., 2003b, 2005, Kang et al., 2003, Wang et al., 2004]. Expiration can be done periodically, provided that old tuples can be identified and skipped during the processing. Figure 2.2(b) is an example of a non-blocking pipeline join [Wilschut and Apers, 1991, Haas and Hellerstein, 1999, Dittrich et al., 2002, Luo et al., 2002, Viglas et al., 2003, Hammad et al., 2003b, Mokbel et al., 2004, Bertino et al., 2004, Tao et al., 2005]. It stores the input streams (S1 and S2), possibly in the form of hash tables, and for each arrival on one of the inputs, the state of the other input is probed to generate new results. Joins of more than two streams and joins of streams with static relations are straightforward extensions. In the former, for each arrival on one input, the states of the other inputs are probed [Viglas et al., 2003]. In the latter, new arrivals on the streams trigger the probing of the relation.

\(^2\)term *probe* is used for operations of searching a processing state for needed data and the processing state can be an original data storage or auxiliary data structures like a hash table, a B-tree
2.1. BASIC CONCEPTS AND TECHNIQUES FOR STREAM PROCESSING

Duplicate elimination operators

Duplicate elimination, as illustrated in Figure 2.2(c), maintains a list of distinct values already seen and filters out duplicates from the output stream. When a new tuple with value \( b \) arrives, the operator probes its output list, and drops the new tuple if a tuple with value \( b \) has already been seen and appended to the output stream.

Duplicate elimination over a sliding window may also produce new output when an input tuple expires. This occurs if a tuple with value \( v \) was produced on the output stream and later expires from its window, yet there are other tuples with value \( v \) still present in the window [Hammad et al., 2003a]. Alternatively, duplicate elimination may produce a single result tuple with a particular value \( v \) and retain it on the output stream as long as there is at least one tuple with value \( v \) present in the window [Arasu et al., 2006, Golab and Özsu, 2005]. In both cases, expirations must be handled eagerly so that the correct result is maintained at all times.

Aggregation operators

Aggregation over a sliding window updates its result when new tuples arrive and when old tuples expire. In many cases, the entire window needs to be stored in order to account for expired tuples, though selected tuples may sometimes be removed early, if their expiration is guaranteed not to influence the result. For example, when computing MAX, tuples with a value \( v \) do not need to be stored if there is another tuple in the window with a value greater than \( v \) and with a younger timestamp (see, e.g., [Lin et al., 2005, Tao and Papadias, 2006] for additional examples of reducing memory usage in the context of skyline queries and [Mouratidis et al., 2006] in the context of top-k queries). Additionally, in order to enable incremental computation, the aggregation operator stores the current answer or frequency counters of the distinct values present in the window. For instance, computing COUNT entails storing the current count, incrementing it when a new tuple arrives, and decrementing it when a tuple expires. Note that, in contrast to the join operator, expirations must be dealt with immediately so that an up-to-date aggregate value can be returned right away. A non-blocking aggregation [Hellerstein et al., 1997, Wang et al., 2003, Law et al., 2004] is shown in Figure 2.2(d). When a new tuple arrives, a new result is appended to the output stream if the aggregate value has changed. The new result is understood to replace previously reported results. GROUP BY may be thought of as a general case of aggregation, where a newly arrived tuple may produce new output if the aggregate value for its group has changed.

The time and space requirements of the aggregation operator depend on the type of function being computed [Gray et al., 1997]. An aggregate \( f \) is distributive if, for two disjoint multi-sets \( X \) and \( Y \), \( f(X \cup Y) = f(X) \cup f(Y) \). Distributive aggregates, such as COUNT, SUM, MAX and MIN, may be computed incrementally using constant space and time (per tuple). For instance, SUM is evaluated by storing the current sum and continually adding to it the values of new tuples as they arrive. Moreover, \( f \) is algebraic if it can be computed using the values of two or more distributive aggregates using constant space and time (e.g., AVG is algebraic because \( AVG = \frac{SUM}{COUNT} \)). Algebraic aggregates are also incrementally computable using constant space and time. On the other hand, \( f \) is holistic if, for two multi-sets \( X \) and \( Y \), computing \( f(X \cup Y) \) requires space proportional to the size of \( X \cup Y \). Examples of holistic aggregates include TOP-k, QUANTILE, and COUNT DISTINCT. For instance, multiplicities of each distinct value seen so far may have to be maintained in order to identify the \( k \) most frequent item types at any point in time. This requires \( \Omega(n) \) space, where \( n \) is the number of stream tuples seen so far, considering a stream with \( n - 1 \) unique values and one of the values occurring twice.
Non-monotonic operators

As non-monotonic query patterns like negation are parts of SPARQL 1.1 which is used as the basic for language for our system in Chapter 4, non-monotonic operators over streams are desirable. Indeed, these operators are possible if previously reported results can be removed when they no longer satisfy the query. This can be done by appending corresponding negative tuples to the output stream [Hammad et al., 2003a, Arasu et al., 2006]. Negation of two sliding windows, \( S_1 - S_2 \), may produce negative tuples (e.g., arrival of an \( S_2 \)-tuple with value \( v \) causes the deletion of a previously reported result with value \( v \)), but may also produce new results upon expiration of tuples from \( S_2 \) (e.g., if a tuple with value \( v \) expires from \( S_2 \), then an \( S_1 \)-tuple with value \( v \) may need to be appended to the output stream [Hammad et al., 2003a]). An example is shown in Figure 2.2(e), where a tuple with value \( d \) was appended to the output because it is generated on the output stream upon subsequent arrival of an \( S_2 \)-tuple with value \( d \).

2.1.5 Handling Memory Overflow

To handle memory overflow, the secondary storage must be used in the query operators. The XJoin operator [Urhan and Franklin, 2000] was introduced to address memory overflow in binary window joins by spilling some partitions of inputs to disk. XJoin extends the Symmetric Hash Join (SHJ) [Hong and Stonebraker, 1993, Wilschut and Apers, 1993] to use less memory by allowing parts of hash tables to be moved to a secondary storage. The MJoin operator [Viglas et al., 2003] generalises the XJoin operator to deal with multiway stream joins. MJoin maximises the output rate of the multi-join operator by efficiently coordinating the spilling processes instead of spilling the inputs to disk randomly without considering the values in their join attributes.

If the secondary storage is used for storing the sliding window, then an index might be used to improve the performance. However, the index introduces maintenance costs especially in the context of frequent updates. In order to reduce the index maintenance costs, it is desirable to avoid bringing the entire window into memory during every update. This can be done by partitioning the data to localise updates (i.e., insertions of newly arrived data and deletion of tuples that have expired from the window) to a small number of disk pages. For example, if an index over a sliding window is partitioned chronologically [Folkert et al., 2005, Shivakumar and García-Molina, 1997], then only the youngest partition incurs insertions, while only the oldest partition needs to be checked for expirations (the remaining partitions in the “middle” are not accessed). A similar idea of grouping objects by expiration time appears in [Douglis et al., 2004] in the context of clustering large file systems, where every file has an associated lifetime. However, the disadvantage of chronological clustering is that records with the same search key may be scattered across a very large number of disk pages, causing index probes to incur prohibitively many disk I/Os. One way to reduce index access costs is to store a reduced (summarised) version of the data that fit in fewer disk pages [Chandrasekaran and Franklin, 2004], but this does not necessarily improve index update times. In order to balance the access and update times, a wave index has been proposed that chronologically divides a sliding window into \( n \) equal partitions, each of which is separately indexed and clustered by a search key for efficient data retrieval [Shivakumar and García-Molina, 1997]. However, the access time of this approach is slower because multiple sub-indices are probed to obtained the answer. To accelerate the access time, [Golab et al., 2006] proposed doubly partitioned indices to simultaneously partition the index on insertion and expiration times.
2.1.6 Optimisation

A continuous query is usually expressed in a declarative language such as CQL. Then it is translated to a logical query plan. In some DSMSs like Aurora, the logical query plan can be composed by the user. Query optimisation might be applied at a logical level by rewriting the plan to improve efficiency, called algebraic optimisation. The common rewriting rules such as reordering selection before joins and evaluating inexpensive predicates before complex ones were used in [Babu et al., 2004b, Golab et al., 2008]. Particularly for continuous queries, Arasu et al. [2006] proposed rules on window-based operators such as commutative rules on time-based and count-based windows.

The logical query plan needs to be scheduled to be executed in the execution engine with a physical plan composed of concrete physical operators and costs. As the data arrives to the engine continuously, the DSMS scheduler can use different equivalent physical plans to execute a logical query plan during the life-time of a query. Traditional DBMSs use selectivity information and available indices to choose efficient physical plans (e.g., those which require fewer disk accesses). However, this cost metric does not apply to (possibly approximate) continuous queries, where processing cost per-unit-time is more appropriate [Kang et al., 2003]. Alternatively, if the stream arrival rates and output rates of query operators are known, then it may be possible to optimise for the highest output rate or to find a plan that takes the least time to output a given number of tuples [Urhan and Franklin, 2001, Viglas and Naughton, 2002, Tao et al., 2005]. Finally, quality-of-service metrics such as response time may also be used in DSMS query optimisation [Abadi et al., 2003, Berthold et al., 2005, Schmidt et al., 2004, 2005].

Optimisation by rescheduling physical query plans are similar to those used in relational databases, e.g., re-ordering a sequence of binary joins in order to minimise a particular cost metric. There has been work in join ordering for data streams in the context of the rate-based model [Viglas and Naughton, 2002, Viglas et al., 2003]. Furthermore, adaptive re-ordering of pipelined stream filters is studied in [Babu et al., 2004a], and adaptive materialisation of intermediate join results are considered in [Babu et al., 2005a]. Note the prevalence of the notion of adaptivity in query rescheduling: operators may need to be re-ordered on-the-fly in response to changes in system conditions. In particular, the cost of a query plan may change for three reasons: change in the processing time of an operator, change in the selectivity of a predicate, and change in the arrival rate of a stream [Avnur and Hellerstein, 2000b].

Initial efforts on adaptive query plans include mid-query re-optimisation [Kabra and DeWitt, 1998] and query scrambling, where the objective was to pre-empt operators that become blocked and to schedule other operators instead [Amsaleg et al., 1996, Urban et al., 1998]. To further increase adaptivity, instead of maintaining a rigid tree-structured query plan, the Eddies approach [Avnur and Hellerstein, 2000b] performs scheduling of each tuple separately by routing it through the operators that make up the query plan. Thereby, the operators of the query plan are dynamically re-ordered to adapt to the current system conditions. This is driven by tuple routing policies that attempt to find which operators are fast and selective, such that those operators are executed first. This approach was applied to continuous queries in [Chandrasekaran and Franklin, 2003, Madden et al., 2002] and was evaluated in [Deshpande, 2004]. The extended version for multi-way joins can be found in [Tok and Bressan, 2002, Raman et al., 2003]. On top of that, it was also extended to consider semantic information such as attribute correlations during routing [Bizarro et al., 2005]). For distributed settings, the queue length is considered as a third factor for tuple routing strategies [Tian and DeWitt, 2003].

To achieve adaptivity, the processing engine has to deal with some overheads. The first overhead is having to re-route each tuple separately. The next overhead is migrating internal states stored in some operators from the current query plan to the new query plan that has a new arrangement of operators.
The issue of state migration across query plans was studied in [Deshpande and Hellerstein, 2004, Zhu et al., 2004]. More details on adaptive query processing can be found in [Gounaris et al., 2002, Babu, 2005a, Deshpande et al., 2007].

When there are multiple continuous queries registered, memory and computing resources can be shared to optimise the overall processing. For selection queries, a possible multi-query optimisation is to index the query predicates and store auxiliary information in each tuple that identify which queries it satisfies [Carnes et al., 1999, Wu et al., 2004, Chandrasekaran and Franklin, 2003, Krishnamurthy et al., 2006, Lim et al., 2006]. When a new tuple arrives for processing, its attribute values are extracted and matched against the query index to see which queries are satisfied by this tuple. Data and queries may be thought of as duals, in some cases reducing query processing to a multi-way join of the query predicate index and the data tables [Chandrasekaran and Franklin, 2003, Lim et al., 2006]. Indexing range predicates is discussed in [Wu et al., 2004, Lim et al., 2006], whereas a predicate index on multiple attributes is presented in [Lee et al., 2004, Lim et al., 2006].

In addition, memory usage may be reduced by sharing internal data structures that store operators’ states [Denny and Franklin, 2005, Dobra et al., 2004, Zhang et al., 2005]. Moreover, in the context of complex queries containing stateful operators such as joins, computation may be shared by building a common query plan [Chen et al., 2000]. For example, queries belonging to the same group may share a plan, which produces the union of the results needed by the individual queries. A final selection is then applied to the shared result set and new answers are routed to the appropriate queries. An interesting tradeoff appears between doing similar work multiple times and doing too much unnecessary work. Techniques that balance this tradeoff are presented in [Chen et al., 2002, Krishnamurthy et al., 2004, Wang et al., 2006]. For example, suppose that the workload includes several queries referencing a join of the same windows, but having a different selection predicate. If a shared query plan performs the join first and then routes the output to the appropriate queries, it is possible that unnecessary tuples are generated, because, some of the joined tuples may not satisfy any selection predicate. On the other hand, if each query performs its selection first and then joins the surviving tuples, then the join operator cannot be shared and the same tuples will be probed many times. Finally, sharing a single join operator among queries referencing different window sizes is discussed in [Hammad et al., 2005].

2.1.7 Scheduling

After the query optimiser chooses a physical query plan, the query engine starts to execute it. In contrast to pull-based operators of DBMSs, DSMS operators consume data pushed into the plan by the sources. At any point during an execution, there may be many tuples in the input and inter-operator queues. Queues allow sources to push data into the query plan and operators to retrieve data as needed [Abadi et al., 2003, Avnur and Hellerstein, 2000b, Arasu et al., 2006, Madden et al., 2002, Madden and Franklin, 2002]. See [Jiang and Chakravarthy, 2003] for a discussion of calculating queue sizes of streaming relational operators using classical queueing theory.

Each operator consumes data from its input queue(s) to return outputs to upper queues. The DSMS scheduler must determine which data item in which queue to process next. A round-robin strategy can be used to execute each operator in a round-robin fashion until it has processed all the data items in its queue(s). Another simple technique, first-in-first-out, is to process one data item at a time in order of arrival, such that each item is processed to completion by all the operators in the plan. This execution strategy ensures good response times, however, scheduling one tuple at a time may incur too much overhead.

Another scheduling strategy is to allocate a time slice to each operator, during which the operator
extracts tuples from its input queue(s), processes them in timestamp order, and deposits output tuples into the next operator’s input queue. The time slice may be fixed or dynamically calculated based upon the size of an operator’s input queue and/or processing speed. A possible improvement could be to schedule one or more tuples to be processed by multiple operators at once. In general, there are several possible conflicting criteria involved in choosing a scheduling strategy, among them queue sizes in the presence of bursty stream arrival patterns [Babcock et al., 2004], average or maximum latency of output tuples [Carney et al., 2003, Jiang et al., 2004, Ou et al., 2005], and average or maximum delay in reporting the answer relative to the arrival of new data [Sharaf et al., 2005]. Additionally, [Viglas and Naughton, 2002, Carney et al., 2003, Sharaf et al., 2008] proposed strategies for scheduling operators to achieve low latency by producing high output rates.

2.2 Linked Data Processing

This section introduces the relevant notations and definitions for the RDF data model [Gutierrez et al., 2004] and the SPARQL query [Pérez et al., 2009a]. It also provides an overview of how to store RDF data and query it using SPARQL.

2.2.1 RDF and SPARQL

From the RDF semantics in [Gutierrez et al., 2004], we introduce definitions related to the RDF model in the following. Let $I$, $B$, and $L$ be RDF nodes which are pair-wise disjoint infinite sets of Information Resource Identifiers (IRIs), blank nodes and literals respectively, and $IL = I \cup L$, $IB = I \cup B$ and $IBL = I \cup B \cup L$ be the respective unions. A triple $(s, p, o) \in IB \times I \times IBL$ is an RDF triple where $s$ is the subject, $p$ the predicate, and $o$ the object. An RDF graph is a set of RDF triples. An RDF graph is referred as an RDF dataset, or simply dataset.

SPARQL is essentially a graph-matching query language. A SPARQL query is of the form $H \leftarrow B$, where $B$, the body of the query, is a complex RDF graph pattern expression that may include RDF triples with variables, conjunctions, disjunctions, optional parts, and constraints over the values of the variables, and $H$, the head of the query, is an expression that indicates how to construct the answer to the query [Pérez et al., 2009a]. The evaluation of a query $Q$ against a dataset $D$ is done in two steps: The body of $Q$ is matched against $D$ to obtain a set of binding values (or bindings) for the variables in the body, and then using the information on the head of $Q$, these bindings are processed applying classical relational operators (projection, distinct, etc.) to produce the answer to the query, which can have different forms, such as a yes/no answer, a table of values, or a new RDF dataset.

The official syntax of SPARQL [Prud’hommeaux and Seaborne, 2008] considers operators OPTIONAL, UNION, and FILTER, and concatenation via a point symbol (.), to construct graph pattern expressions. The syntax also includes {} to group patterns, and some implicit rules of precedence and association. For example, the point symbol (.) has precedence over OPTIONAL, and OPTIONAL is left associative. In order to avoid ambiguities in the parsing, we present the syntax of SPARQL graph patterns in a more traditional algebraic formalism, using binary operators AND (.), UNION (UNION), OPT (OPTIONAL), and FILTER (FILTER).

1. A tuple from $(IB \cup V) \times (I \cup V) \times (IBL \cup V)$ is a graph pattern (a triple pattern) where $V$ is an infinite set of variables disjoint from $IBL$. 
2. If $P_1$ and $P_2$ are graph patterns, then the expressions $(P_1 \text{ AND } P_2)$, $(P_1 \text{ OPT } P_2)$, and $(P_1 \text{ UNION } P_2)$ are graph patterns (conjunction graph pattern, optional graph pattern, and union graph pattern, respectively).

3. If $P$ is a graph pattern and $R$ is a SPARQL built-in condition, then the expression $(P \text{ FILTER } R)$ is a graph pattern (a filter graph pattern).

As defined in [Prud’hommeaux and Seaborne, 2008], a SPARQL built-in condition is constructed using elements of the set $(I \cup L \cup V$ and constants), logical connectives ($\neg, \vee, \wedge$), inequality symbols ($<, \leq, \geq, >$), the equality symbol ($=$), and unary predicates like $\text{bound}$, $\text{isBlank}$, and $\text{isIRI}$, etc. A built-in condition is a Boolean combination of terms constructed by using $=$ and $\text{bound}$ as follows:

1. If $?X , ?Y \in V$ and $c \in I \cup L$, then $\text{bound(?X )}, ?X = c$ and $?X = ?Y$ are built-in conditions.
2. If $R_1$ and $R_2$ are built-in conditions, then $(\neg R_1), (R_1 \wedge R_2)$ and $(R_1 \vee R_2)$ are built-in conditions.

The semantics of SPARQL is defined via mappings. A mapping $\mu$ is defined as a partial function from $V$ to $\text{IBL}$ defined as

$$\mu : V \mapsto \text{IBL.} \quad (2.6)$$

The domain of $\mu$, $\text{dom}(\mu)$, is the subset of $V$ where $\mu$ is defined. Two mappings $\mu_1$ and $\mu_2$ are compatible, denoted as $\mu_1 \equiv \mu_2$, if:

$$\mu_1 \equiv \mu_2 \iff \forall x \in \text{dom}(\mu_1) \cap \text{dom}(\mu_2) \Rightarrow \mu_1(x) = \mu_2(x) \quad (2.7)$$

Let $P$ be a SPARQL graph pattern, we denote the set of variables occurring in $P$ as $\text{var}(P)$. In particular, if $t$ is a triple pattern, then $\text{var}(t)$ denotes the set of variables occurring in the components of $t$. Similarly, for a built-in condition $R$, we use $\text{var}(R)$ to denote the set of variables occurring in $R$. The triple obtained by replacing elements in $\text{var}(t)$ according to $\mu$ is denoted as $\mu(t)$.

For composing a query, a series of relational operators on sets of mappings are provided such as $\text{join}(\bowtie)$, $\text{union}(\cup)$, $\text{minus}(\setminus)$ and $\text{left outer join}(\bowtie \left)$.

$$\Omega_1 \bowtie \Omega_2 = \{ \mu_1 \cup \mu_2 \mid \mu_1 \in \Omega_1 \land \mu_2 \in \Omega_2 \land \mu_1 \equiv \mu_2 \} \quad (2.8)$$

$$\Omega_1 \cup \Omega_2 = \{ \mu \mid \mu_1 \in \Omega_1 \lor \mu_2 \in \Omega_2 \} \quad (2.9)$$

$$\Omega_1 \setminus \Omega_2 = \{ \mu \in \Omega_1 \mid \neg \exists \mu' \in \Omega_2, \mu' \equiv \mu \} \quad (2.10)$$

$$\Omega_1 \bowtie \left \Omega_2 = (\Omega_1 \bowtie \Omega_2) \cup (\Omega_1 \setminus \Omega_2) \quad (2.11)$$

The semantics of graph pattern expressions is a function $[[.]]_D$ which takes a pattern expression and returns a set of mappings. The evaluation of a graph pattern $P$ over an RDF dataset $D$, denoted by $[[P]]_D$, is defined recursively as follows:

1. If $P$ is a triple pattern $t$, then $[[P]]_D = \{ \mu \mid \text{dom}(\mu) = \text{var}(t) \text{ and } \mu(t) \in D \}$. 

2. If $P$ is $(P_1 \text{ AND } P_2)$, then $[[P]]_D = [[P_1]]_D \bowtie [[P_2]]_D$. 

3. If $P$ is $(P_1 \text{ OPT } P_2)$, then $[[P]]_D = [[P_1]]_D \bowtie \left [[P_2]]_D$. 


2.2. LINKED DATA PROCESSING

4. If $P$ is $(P_1 \text{ UNION } P_2)$, then $[[P]]_D = [[P_1]]_D \cup [[P_2]]_D$.

The semantics of filter expressions is defined as follows. Given a mapping $\mu$ and a built-in condition $R$, we say that $\mu$ satisfies $R$, denoted by $\mu \models R$, if:

1. $R$ is $\text{bound}(X)$ and $X \in \text{dom}(\mu)$.
2. $R$ is $?X = c$, $?X \in \text{dom}(\mu)$ and $\mu(?X) = c$.
3. $R$ is $?X = ?Y$, $?X \in \text{dom}(\mu)$, $?Y \in \text{dom}(\mu)$ and $\mu(?X) = \mu(?Y)$.
4. $R$ is $(\neg R_1)$, $R_1$ is a built-in condition, and it is not the case that $\mu \models R_1$.
5. $R$ is $(R_1 \lor R_2)$, $R_1$ and $R_2$ are built-in conditions, and $\mu \models R_1$ and $\mu \models R_2$.
6. $R$ is $(R_1 \lor R_2)$, $R_1$ and $R_2$ are built-in conditions, $\mu \models R_1$ or $\mu \models R_2$.

Hence, the evaluation of the filter expression $(P \text{ FILTER } R)$ over a dataset $D$ is defined as:

$$[[P \text{ FILTER } R]]_D = \{ \mu \in [[P]]_D \mid \mu \models R \}.$$ 

2.2.2 Physical Storage of RDF Data

Constructing a basic schema for RDF storage is rather simple. For example, Harris and Shadbolt [2005] represent RDF using the relational model and translate SPARQL queries into SQL. Also, many RDF stores are built into or on top of existing relational DBMS engines, and even non-relational RDF stores usually use the concepts of select, project, and join to answer queries. Conceptually, RDF can be simply modelled as a long list of triples, and this can be represented using a single relation. However, in practice, an RDF store consists of a triple table and associated mapping tables. The triple table stores identifiers representing URIs and literals and the mapping tables are used to translate these identifiers back into their lexical form [Owens, 2011].

This approach is exemplified by 3Store [Harris and Shadbolt, 2005], a system of moderate performance that runs on top of the MySQL relational engine. 3Store uses a single table to store the graph shape. As a consequence, additional SQL is required in 3Store to determine the lexical representation of the hash values that would be returned by a query. In the case of additional constrains in the SPARQL query, 3Store simply performs joins back onto the triple table. 3Store relies on the MySQL query optimiser to optimise the SQL it produces. This schema offers a significant degree of flexibility, by virtue of the fact that any representation of triples is stored in a generic fashion, without the requirement for schema or index customisation. There is no limitation upon the structure of the graph, except for the amount of data that MySQL can efficiently process.

The approach of a long triple table stored in a relational database is common in the world of RDF stores. Popular systems such as Jena [Wilkinson et al., 2003], Sesame [Broekstra et al., 2002], and Redland [Beckett, 2001] have relational backends that utilise this kind of structure. An alternative structure for RDF data, called Property Tables, was described in [Abadi et al., 2007]. This approach assigns a separate table to every property, storing each unique Subject-Object (SO) combination associated with that property in the table. While initial results showed that this ordering has substantial advantages, subsequent investigations showed that using a different sort order for the triple table approach substantially reduces the performance improvement [Schmidt et al., 2008].
While implementing an RDF storage is relatively simple, the nature of the simple RDF schema intractable for real RDBMSs because the triple tables are exceptionally long, with very little information per row. Since RDF stores typically operate on one or a few very large tables, it is impractical to simply iterate through them to find a particular data item. Indexes make it possible to specify fixed values for one or several attributes, and jump straight to results with these values. Most modern RDF stores employ a heavy indexing strategy to overcome their large table handicap. Indeed, many stores have no need for a table, because all possible access patterns are covered by their indexes [Weiss et al., 2008].

Assuming the use of an index type that is ordered (that is, given an SPO index, one cannot restrict by P alone: S must be specified), N! indexes, where N is the number of attributes, are required if one wishes to maintain a truly comprehensive set. While it is possible to use just three indexes: Subject-Predicate-Object (SPO), Predicate-Object-Subject (POS), and Object-Subject-Predicate (OSP), some newer RDF stores do implement all six of the possible index orderings over the RDF data. Particularly, Hexastore [Weiss et al., 2008] and RDF-3X [Neumann and Weikum, 2010] create all six possible index permutations, allowing them to make the greatest possible use of merge joins. It can be seen that for any given combination of subject, predicate, or object, a corresponding index can be found in this set that is suitable to retrieve related data. A second point of interest in this design is that the table becomes unnecessary since the indexes contain all the data, so all the work can be done within them.

A commonly used index for RAM-based storage is the hash table (or hash map) [Date, 1991]. Using a hash map, one might take the hash of a piece of data, and then store in the memory position corresponding to that hash, a pointer to the location of that piece of data in the database. This is an O(1) operation, and since hash indexes usually require only one or two comparisons to be performed, the problem of unpredictable branches is effectively eliminated. It is, of course, necessary to utilise a suitable hashing algorithm to ensure that there are not too many hash collisions, and that the process as a whole offers good performance. Unfortunately, hashes do exhibit a variety of less desirable characteristics. Hash indexes do not, of course, guarantee that there is any proximity on disk of logically ordered data (for example, sorted order). This means that if one were to perform a query that acts on a range of values, a disk seek would likely be required for each different value, creating massive efficiency issues. For this reason, hash indexes are usually used only in situations where queries are operating on discrete specified values, not over a range. The most popular in-memory RDF stores such as Jena, Sesame, and SwiftOwlim use hash maps to store data.

From the point of view of RDF/SPARQL, which rarely utilise limited range searches, hash maps can be an appropriate solution for both disk and memory storage. Indexing is of critical importance to RDF stores as indexes offer vast benefits when attempting to retrieve a few values from a very long table, which is a common situation in RDF storages. As mentioned above, some RDF stores exhibit such comprehensive indexing that there is no longer a need for the original data table. Maintenance of such a strategy is sustainable for triple stores, but becomes more challenging as more attributes, such as provenance or temporal information, are required. Traditional B-tree indexes perform well for disk based storage. They are simple to implement, and require a small number of disk seeks to find a given item compared to other tree-based methods. They waste a certain amount of space through their partially filled nature and the repetition of prefixes, but this latter can be mitigated through the use of compression. As a result of their flexibility and reliably good performance, B-trees, in particular the B+tree variant, remain perhaps the most common data structure for implementing disk-based indexes [Comer, 1979]. Most triple stores backed by existing RDBMSs will make exclusive use of this

---

3http://jena.apache.org/
4http://www.openrdf.org/
5http://www.ontotext.com/owlim
index, and other dedicated systems such as Jena TDB [Owens et al., 2008] and RDF-3X [Neumann and Weikum, 2010] implement their own versions.

For memory-based RDF systems, trees in general are a capable but limited solution. While they provide strong guarantees regarding sorted order, this is more than required for RDF stores that do not rely on merge sort, and thus have no use for sorted order. Trees generally waste a substantial amount of space in pointers and/or empty space, and offer poor characteristics with regards to contiguity of access during find operations. Block based tree indexes like B-Trees, however, offer very good contiguity of access when iterating over leaf nodes. Hash indexes are generally appropriate for in-memory RDF storage, as they offer amortised O(1) retrieval and update with a low constant factor. Hash indexes bring with them their own issues. Care must be taken to ensure efficient use of space when creating hash indexes, and it should be noted that hash indexes do not inherently support indexing over more than one attribute. To scale to large RDF datasets, a space efficient solution to provide multi-attribute indexing is required.

Current RDF stores, particularly those that scale to very large numbers of triples, tend towards read optimisation. While the initial bulk insert can be extremely fast, subsequent insertions can exhibit much poorer performance. Deletions pose their own difficulties. Particularly, in a normalised ID-based system, it is relatively time or space consuming to keep track of when IDs are no longer in use, and there needs to be a mechanism for ID recovery and reuse—whether it is an ongoing process or a bulk operation (which requires a sufficiently large ID and storage space). This is a relatively small problem in stores that do not experience significant deletions, but is important for systems that experience loads with regular updates. Current stores tend to be optimised for read operations, and do not perform ID deletion.

2.2.3 SPARQL Query Processing

There are four steps in processing a SPARQL query:

1. compile a SPARQL query to an internal form;
2. convert to a canonical form;
3. choose the candidate physical operators;
4. generate the query plans and choose the cheapest.

The first two steps essentially transform the query from a textual representation of SPARQL into an internal form that is easier to process, performing trivial optimisations such as eliminating irrelevant statement ordering on the way. Step 3 is more complex, and involves working out low-level operations that can satisfy parts of the query. This attempts to produce worthwhile operations by considering information such as physical data structure on disk, availability of indexes to speed up the operation. Each potential operation will have an associated cost calculated for it, at the minimum specifying number of disk accesses required, but it will possibly also include information such as memory and CPU usage. This data may be estimated where hard figures are not available or can not be easily calculated. Depending on whether the operation has prerequisites for other operations to be performed first, it may well be possible to perform them simultaneously across multiple processor cores, processors, and disks to enhance the performance. Finally, Step 4 involves the creation of a set of potential plans from the procedures generated in Step 3. Clearly, there could be too many plans produced if there were a significant set of candidate procedures generated, so a heuristic to create only plausible plans is of
great use in this situation. The order in which operations are performed has a huge impact on the query performance if the correct operations are performed early in the query, the working set can be cut down to the point that later, more challenging operations only have to work on a small amount of data.

Answering a SPARQL query over a traditional triple table schema implies joining the table onto itself repeatedly, once for each triple in the query. Join is the most-used operation on RDF data. Join operation can quickly become very expensive if the working set of information is allowed to grow too large. Thus, there are two types of join algorithms to reduce time spent in joins: high-performance join algorithms and algorithms for minimizing the set of data to be joined in the first place. There are a variety of algorithms of each kind which are differentiated by how the data is sorted and indexed for the join operations. This ranges from the very basic brute force algorithm, with a scaling factor of $O(n^2)$ with the size of the data being examined, to more useful techniques, such as indexed nested loops, merge, sort/merge, and hash joins [Date, 1991].

The index nested loops (INL) join relies on an index being available on the inner join input. This index is consulted for matches against each row of the outer join input. This is substantially faster than the basic brute force algorithm because, for each row of the outer join input, only a single operation has to be performed if there is no match, rather than iterating through the entire inner input. Using index on the inner join is especially effective in terms of reducing computation in some situations: the left join input is small and is being matched against a very large right join input and the selectivity of the index is considerably low. In such cases, INL joins require very little memory. They are particularly relevant to RDF storage, since RDF stores often have comprehensive indexing strategies [Owens, 2011]. INLs are faster than or comparably effective to hash joins in the general case, as long as an appropriate index is available. In some other cases, hash joins are substantially more effective [DeWitt and Gerber, 1985], and are thus used more often in ad-hoc queries. Nested loops queries also are highly effectively parallelised [Sheu and Thai, 1991]. However, in disk-based stores, the looped accesses to the index will have a cost in terms of repeated disk seeks, compared to hash or sort/merge joins, which are less inherently selective, but allow more linear disk accesses. Seek time is a smaller issue in memory-backed stores, and INLs joins can be highly effective in this environment, benefitting especially from their low memory consumption.

A hash join performs a single scan over each input. It creates a hash table on the first input, with a pointer to the corresponding tuple. When scanning the second input, it compares against that hash table to produce the joined output. This technique scales in a linear manner with the amount of data scanned, and does not require inputs to be sorted to work efficiently (although, of course, sorting will ensure better contiguity of access and cache utilisation). It is, however, likely to be slower than a merge join, since a computational expense comes with operations such as hashing. Further, it is less tractable to hold all the intermediate data on disk if no memory is available.

Merge joins assume that both inputs are sorted on the columns that are being joined on. With this being the case, a simple scan of both inputs can perform a join in linear time with the amount of data being joined, if the join is one to many, or near linear if it is many to many. A merge join is always faster than sort/merge or hash joins if data is sorted correctly. For this reason, query optimisers in an RDBMS will usually keep track of the sort order of the current working set of data, and will order joins to allow as much use of merge joining as possible. Sort/Merge joins simply sort the inputs as required, and then perform a merge join on the resulting data. This approach is largely constrained by the performance of the sort.

RDF exhibits an unusual problem with regards to operator implementation. Its large triple tables (or, indeed, property tables) and the high likelihood of appropriate indexes being available means that the choice between INL and merge join is not completely clear for disk-backed systems. For
2.2. LINKED DATA PROCESSING

memory-backed systems, it is likely that INL will have a significant advantage due to the lower cost of seeking. Thanks to the sheer quantity of data points in a typical RDF store, RDF does require a special emphasis on minimising the time spent in joins. This can be achieved by methods such as intelligent query optimisation, and join pre-calculation. The former is important for ordering queries appropriately, such that the working set stays as small as possible. This is a challenging problem which requires high quality statistics to estimate the size of each data retrieval.

Some systems, like RDF-3X [Neumann and Weikum, 2010], generate exhaustive statistics over their datasets, while others, such as Virtuoso, rely on estimations to save space [Erling and Mikhailov, 2009]. Since it is expensive to generate high quality statistics about RDF data due to the large quantity of data points, there is room for research in this area. The standard physical storage schema in RDF stores is a three index layout using SPO, POS, and OSP indexes, effectively covering all access paths into the system. Some dedicated stores, such as Jena TDB, use an exclusively index nested loop approach to joining data, while systems based on top of RDBMSs will use whatever join mechanism the system chooses at query time. Some newer stores, specifically Hexastore [Weiss et al., 2008] and RDF-3X [Neumann and Weikum, 2010], have chosen a different approach. They create all six possible index permutations, allowing them to make the greatest possible use of merge joins, and where merging is not possible employ sort-merge or hash join strategies.

Join pre-calculation is clearly attractive for the large corpus, read-mostly use case. There is a clear need to be able to determine what precalculation is necessary, which again offers an opening for new work in the area. As previously noted, reducing time spent in joins is an excellent method for improving overall RDF store performance. One method for achieving this is to perform the work in advance. Abadi et al. [2007] describe the concept of “materialised path expressions”, in essence the process of pre-calculating joins such that they do not have to be performed at run time. The authors note that this can provide an orders of magnitude level improvement in performance on suitable queries.

Join pre-calculation is generally very attractive for read optimised disk-based systems. If a given join is performed regularly, a great deal of time can be saved by storing the completed join on disk. There are, however, a variety of complications to this approach. The pre-calculated data needs to be updated every time a related piece of information is added or removed, which can be expensive. In addition, it is necessary to determine what pre-calculated information would actually offer a significant benefit, which can be a complex process. Doing this work manually would be difficult, so it becomes necessary to maintain accurate usage statistics (or batch-processable logs) to allow the determination of what joins should be pre-calculated. Finally, pre-calculated joins are clearly not suitable for systems where storage space is limited.
Chapter 3

State of The Art in Linked Stream Data Processing

Linked Stream Data [Sequeda and Corcho, 2009] was introduced in order to bridge the gap between stream and Linked Data, and to facilitate data integration among stream sources and also between streams and other static sources. It follows the standards of Linked Data [Bizer et al., 2009b], and it is usually represented as an extension of RDF. Assigning URIs to RDF streams not only allows to access the RDF streams as materialised data, but also enables the query processor to treat the RDF stream elements as RDF nodes, such that other SPARQL query patterns can be directly applied. This chapter presents how the concepts and techniques of stream processing and Linked Data introduced in the previous chapter are used to define the Linked Stream Data model, and how they are employed to build Linked Stream Data processing engines. A survey of state-of-the-art systems is given to highlight the issues to be addressed in our system which will be described in Chapter 5. These systems, namely Streaming SPARQL [Bolles et al., 2008], C-SPARQL [Barbieri et al., 2010b], SPARQL\textsubscript{stream} [Calbimonte et al., 2010], and EP-SPARQL [Anicic et al., 2011] have been developed in parallel with our CQELS engine [Le-Phuoc et al., 2011] that will be described in the following chapters. Therefore, we include the CQELS engine in the survey to clarify our contribution in the field and defer the description of CQELS to later chapters.

3.1 Data Model and Query Semantics

This section reviews how to formalise the data model for RDF streams and RDF datasets in a continuous context as well as the semantics of the continuous queries.

3.1.1 Data Model

Linked Stream Data is modelled by extending the definitions of RDF nodes and RDF triples presented in Section 2.2.1. Stream elements of Linked Stream Data are represented as RDF triples with temporal annotations. A temporal annotation of an RDF triple can be an interval-based [Lopes et al., 2010] or point-based [Gutierrez et al., 2007] label. An RDF stream $S$ is defined as a bag of elements $\langle (s, p, o) : [t] \rangle$, where $(s, p, o)$ is an RDF triple and $t$ is a temporal annotation, i.e., a timestamp or a time interval.
An interval-based label is a pair of timestamps, commonly natural numbers representing logical time. A pair of timestamps, \([\text{start}, \text{end}]\), is used to specify the interval in which the RDF triple is valid. For instance, \(\langle \text{:John :at :office},[7,9]\rangle\) means that John was at the office from 7 to 9. A point-based label is a single natural number representing the point in time that the triple was recorded or received. In the previous example, the triple \(\langle \text{:John :at :office}\rangle\) might be continuously recorded by a tracking system, so three temporal triples could be generated \(\langle \text{:John :at :office},7\rangle,\langle \text{:John :at :office},8\rangle,\langle \text{:John :at :office},9\rangle\).

Point-based labels might look redundant and less efficient in comparison to interval-based labels. Furthermore, interval-based labels are more expressive than point-based because the later is a special case of the former, i.e., when \(\text{start} = \text{end}\). Streaming SPARQL [Bolles et al., 2008] uses interval-based labels for representing its physical data stream items and EP-SPARQL [Anicic et al., 2011] uses them for representing triple-based events. However, point-based labels are more practical for streaming data sources because it allows triples to be generated unexpectedly and instantaneously. For example, a tracking system detecting people at an office can generate a triple with a timestamp whenever it receives a reading from a sensor. Otherwise, the system has to buffer the readings and do further processing in order to generate the interval in which a triple is valid. Moreover, the instantaneity of point-based labels is vital for some applications that need to process the data as soon as it arrives in the system. For instance, an application that notifies where John is should be triggered at time point 7 and not wait until time 9 to report that he was in the office from 7 to 9. Point-based labels are supported in C-SPARQL [Barbieri et al., 2010b], SPARQL\textit{stream} [Calbimonte et al., 2010] and CQELS [Le-Phuoc et al., 2011].

To enable the integration of stream data with non-stream data, the concept of an RDF dataset has to be included in the data model. The current state-of-the-art considers RDF datasets as static data sources. However, applications on stream data can be run for days, months or years, and thus the changes in the RDF dataset during the query lifetime need to be reflected in the outputs of the continuous queries. In the next chapter, we propose a new data model that can cope with such changes in RDF datasets.

### 3.1.2 Query Semantics

For defining the semantics of continuous query languages, current state-of-the-art approaches extend the query operators of SPARQL such as join, union and filter shown in Definitions 2.8-2.11. As such operators consume and output mappings, these approaches also introduce operators on RDF streams to output mappings. For instance, in C-SPARQL, the \textit{stream} operator is defined to access an RDF stream identified by its IRI and the \textit{window} operator is defined to extract an RDF graph from an RDF stream based on a certain window. The definitions of window operators on RDF streams are adopted from window operators on relational streams of CQL [Arasu and Widom, 2004a].

The semantics of a continuous query on RDF streams are defined as a composition of the query operators. In Streaming SPARQL and C-SPARQL, a query is composed as an operator graph. The definition of the operator graph is based on SPARQL query graph model [Hartig and Heese, 2007]. Figure 3.1 shows an operator graph of a C-SPARQL query. The operator graph shows how two new operators stream (<http://../market.trdf>) and window \(( \omega_{\text{logical}(24\,\text{hours})})\) on the bottom right are connected to the rest of the graph that is actually the traditional operator graph of a SPARQL query.
3.1. DATA MODEL AND QUERY SEMANTICS

This visual representation only shows the “one-shot” semantics of the query, meaning, the output of the query at a certain point in time. It does not explicitly show the continuous semantics of a continuous query. The continuous semantics defines how the output stream is generated when the time progresses. For example, in CQL [Arasu and Widom, 2004a], the continuous semantics are defined by three streaming operators introduced in Section 2.1.2, namely ISTREAM, DSTREAM and RSTREAM. The ISTREAM and DSTREAM operators are used for generating the stream of new results or expired results, respectively. The RSTREAM operator generates the stream of the latest results of the query. Therefore, ISTREAM and DSTREAM only stream new outputs or expired outputs instead of streaming all the outputs of each query shot regardless of the previous query shot like RSTREAM.

These streaming operators are accompanied with definitions that show the relationship between their two consecutive outputs. For instance, Definitions 2.3–2.4 of ISTREAM, DSTREAM in Section 2.1.2 show how to compute the outputs at the time \( t \) based on the outputs at time \( t - 1 \). Such definitions provide the foundation for building the incremental computation algorithms of continuous query processing which will be presented in Chapter 6. The continuous semantics of C-SPARQL [Barbieri et al., 2010b] are not formally defined, but C-SPARQL engine uses the continuous semantics of

---

**Figure 3.1:** An operator graph of C-SPARQL query.
CHAPTER 3. STATE OF THE ART IN LINKED STREAM DATA PROCESSING

RSTREAM for its output in the current implementation [Le-Phuoc et al., 2012a]. EP-SPARQL uses the incremental execution triggered by new inputs, so it has the same continuous semantics as that of ISTREAM. In general, none of the current works on introducing continuous query languages for Linked Stream Data provide a formal continuous semantics for the queries. Our own formalisation is given in detail in Chapter 4.

3.2 Query Languages

To define a declarative query language for Linked Stream Data, first the basic query patterns need to be introduced to express the primitive operators. These basic patterns are triple matching, window matching, and sequential operators. Then, the composition of basic query patterns can be expressed by the AND, OPT, UNION and FILTER patterns of SPARQL. These patterns correspond to the operators in Definitions 2.8-2.11 of Section 2.2.1. To support aggregation operators, Barbieri et al. [2010b] defines their semantics with the query pattern AGG. The query pattern AGG is defined to be compatible with other query patterns of SPARQL. Thereby, the evaluation of query pattern AGG is defined as 
\[ P_{\text{AGG}} A = P \bowtie [[A]] \]
where \( A \) is an aggregate function that consumes output of a SPARQL query pattern \( P \) to return a set of mappings. Let \( P_1, P_2 \) and \( P \) be basic or composite query patterns, a declarative query can be composed recursively using the following rules:

1. \([[[P_1 \text{ AND } P_2]]] = [[P_1]] \bowtie [[P_2]].\]
2. \([[[P_1 \text{ OPT } P_2]]] = [[P_1]] \nabla [[P_2]].\]
3. \([[[P_1 \text{ UNION } P_2]]] = [[P_1]] \cup [[P_2]].\]
4. \([[[P \text{ FILTER } R]]] = \{ \mu \in [[P]] | \mu \models R \} \), where \( \mu \models R \) if \( \mu \) satisfies condition \( R \).
5. \([[[P \text{ AGG } A]]] = [[P]] \bowtie [[A]].\]

The above query pattern construction rules enable extending the SPARQL grammar for continuous queries. Streaming SPARQL extends the SPARQL 1.0 grammar\(^1\) by adding the DatastreamClause and a clause for defining windows as shown in the following EBNF grammar:

\[
\begin{align*}
\text{SelectQuery} &::= \text{’SELECT’} (\text{’DISTINCT’} | \text{’REDUCED’})?((\text{Var} | \text{’*’})(\text{DatasetClause+} | \\
\text{DatastreamClause})\text{WhereClause} \text{SolutionModifier}) \\
\text{DatastreamClause} &::= \text{’FROM’} (\text{DefaultStreamClause} | \text{NamedStreamClause}) \\
\text{DefaultStreamClause} &::= \text{’STREAM’}\ SourceSelector\ Window \\
\text{NamedStreamClause} &::= \text{’NAMED’}\ SourceSelector\ Window \\
\text{GroupGraphPattern} &::= \{\text{TriplesBlock}?((\text{GraphPatternNotTriples} \text{Filter} )?')? \\
&\quad\quad\text{TriplesBlock}?('((\text{Window})?)')\} \\
\text{Window} &::= \text{’WINDOW’} \text{’RANGE’}\ ValSpec\ ”SLIDE” \ ValSpec? \\
\text{SlidingDeltaWindow} &::= \text{’WINDOW’} \text{’RANGE’}\ ValSpec\ ”SLIDE” \ ValSpec? \\
\text{FixedWindow} &::= \text{’WINDOW’} \text{’RANGE’}\ ValSpec\ ”FIXED” \\
\text{SlidingTupleWindow} &::= \text{’WINDOW’} \text{’ELEMS’}\ \text{INTEGER} \\
\text{ValSpec} &::= \text{INTEGER} \ | \ \text{Timeunit}? \\
\text{Timeunit} &::= (\text{‘MS’} \ | \ \text{’S’} \ | \ \text{’MINUTE’} \ | \ \text{’HOUR’} \ | \ \text{’DAY’} \ | \ \text{’WEEK’})
\end{align*}
\]

Similarly, the C-SPARQL language extends the SPARQL 1.1 grammar\(^2\) by adding the FromStrClause

\[^1\text{http://www.w3.org/TR/rdf-sparql-query/#grammar}\]
\[^2\text{http://www.w3.org/TR/sparql11-query/#grammar}\]
and a clause for windows as follows:

\[
\begin{align*}
&\text{FromStrClause} \rightarrow \text{FROM} [\text{NAMED}] \text{STREAM} \text{IRI} \{\text{RANGE} \text{ Window}\} \\
&\text{Window} \rightarrow \text{LogicalWindow} | \text{PhysicalWindow} \\
&\text{LogicalWindow} \rightarrow \text{Number} \text{ TimeUnit} \text{WindowOverlap} \\
&\text{TimeUnit} \rightarrow \text{d} | \text{h} | \text{m} | \text{s} | \text{ms} \\
&\text{WindowOverlap} \rightarrow \text{STEP} \text{Number} \text{ TimeUnit} | \text{TUMBLING}
\end{align*}
\]

For generating an RDF stream from the query output, the CONSTRUCT language construct is introduced in C-SPARQL. The triple patterns of the CONTRUCT grammar define the template of the output RDF stream. To show the main differences in the proposed languages, we discuss 5 continuous queries over Linked Stream Data and how they are expressed in these languages in the following.

**Example 8:** To enhance the user experience in our conference scenario introduced in Section 1.4, each participant would have access to the following services, which can all be modelled as continuous queries:

(Q1) inform a participant about the name and description of the location he/she just entered;

(Q2) notify two people when they can reach each other from two different and directly connected locations (in the following called nearby);

(Q3) notify an author of his co-authors who have been in his current location during the last 5 seconds;

(Q4) notify an author of the editors of a paper of his/her and that have been in a nearby location in the last 15 seconds;

(Q5) count the number of co-authors appearing in nearby locations in the last 30 seconds, grouped by location.

The grammars of Streaming SPARQL and C-SPARQL are similar: The URI of the stream is defined after the keywords FROM STREAM and the triple patterns are placed in the WHERE clause. Let us consider a query Q1, and let us assume that the URI of the RFID-based location stream is \(<http://deri.org/streams/rfid>\). An RDF dataset has to be specified to integrate the metadata of the building. For example, the named graph \(<http://deri.org/floorplan/>\) can be used as the RDF dataset to be correlated with the RFID stream. In Streaming SPARQL [Bolles et al., 2008], using an RDF dataset is not clearly described, therefore none of the above queries can be expressed in its language. In C-SPARQL, query Q1 is expressed as follows.

```sparql
PREFIX lv: <http://deri.org/floorplan/>
PREFIX dc: <http://purl.org/dc/elements/1.1/>
PREFIX foaf: <http://xmlns.com/foaf/0.1/>
SELECT ?locName ?locDesc
FROM STREAM <http://deri.org/streams/rfid> [NOW]
FROM <http://deri.org/floorplan/>
WHERE {
?person foaf:name "$Name$".
}
```

Query Q1 in C-SPARQL
Note that, among the four query languages introduced, only Streaming SPARQL and C-SPARQL explicitly define the grammars for their query specifications. The others only illustrate their query language via examples. The FROM clause for representing a window on a RDF stream creates a limitation when two windows are defined on one single RDF stream in a query. The queries Q2, Q3, Q4, and Q5 below need to declare two different windows on the RFID stream, then join these windows with other data. However, using a stream window clause after the FROM keyword only allows one window in each stream URI. Therefore, these four queries can not be expressed directly as single queries in the Streaming SPARQL and C-SPARQL languages. In C-SPARQL, it is possible to get around this issue by creating two separate logical streams from the same stream. These new streams will then be used to apply the two windows needed. For example, we can clone a new stream from the original stream <http://deri.org/streams/rfid> with a new URI <http://deri.org/streams/rfidcopy>. Then we can represent Q2, Q3, Q4, and Q5 as follows:

```
PREFIX lv: <http://deri.org/floorplan/>
  GRAPH <http://deri.org/streams/rfid> {?person1 lv:detectedat ?loc1}
  GRAPH <http://deri.org/streams/rfidcopy> {?person2 lv:detectedat ?loc2}
  GRAPH <http://deri.org/floorplan/> {?loc1 lv:connected ?loc2}
}
```

Query Q2 in C-SPARQL

```
PREFIX lv: <http://deri.org/floorplan/>
PREFIX dc: <http://purl.org/dc/elements/1.1/>
PREFIX foaf: <http://xmlns.com/foaf/0.1/>
  GRAPH <http://deri.org/streams/rfid> {?auth lv:detectedat ?loc}
  GRAPH <http://deri.org/streams/rfidcopy> {?coAuth lv:detectedat ?loc}
  FILTER (?auth != ?coAuth) }
```

Query Q3 in C-SPARQL

```
PREFIX lv: <http://deri.org/floorplan/>
PREFIX dc: <http://purl.org/dc/elements/1.1/>
PREFIX foaf: <http://xmlns.com/foaf/0.1/>
PREFIX dcterms: <http://purl.org/dc/terms/>
PREFIX swrc: <http://swrc.ontoware.org/ontology#>
  GRAPH <http://deri.org/streams/rfid> {?auth lv:detectedat ?loc1}
  GRAPH <http://deri.org/streams/rfidcopy> {?editor lv:detectedat ?loc2}
  GRAPH <http://deri.org/floorplan/> {?loc1 lv:connected ?loc2}
  ?auth foaf:name "$Name$" }
```

Query Q4 in C-SPARQL
However, cloning a stream to get the URI is impractical not only for representing a query but also in terms of processing efficiency. In a recent C-SPARQL implementation (version 0.7.4\(^3\)), specifying multiple windows on one stream URI is introduced. However, this new option leads to an ambiguity: The windows are not accompanied by the triple patterns, the compiler would not be able to identify which triple patterns are specified for which windows. For example, the query Q2 can be rewritten as:

```
PREFIX lv: <http://deri.org/floorplan/>
CONSTRUCT {?person1 lv:reachable ?person2}
FROM STREAM <http://deri.org/streams/rfid> [NOW]
FROM STREAM <http://deri.org/streams/rfidcopy> [RANGE 3s]
FROM NAMED <http://deri.org/floorplan/>
WHERE {
  {?person1 lv:detectedat ?loc1}
  {?person2 lv:detectedat ?loc2}
  GRAPH <http://deri.org/floorplan/> {?loc1 lv:connected ?loc2}
}
```

Here, the compiler does not have information to associate the triple patterns \(?\text{person1} \text{lv : detectedat} \?\text{loc1}\) and \(?\text{person2} \text{lv : detectedat} \?\text{loc2}\) in the WHERE clause to the correct windows specified in the FROM clauses. In our proposed Linked Stream Data processing engine, we overcome this issue by associating the window syntaxes with the triple patterns. This will be described in detail in Chapter 4.

### 3.3 Architectural Design

The architectural design of the current systems that support continuous query processing over Linked Streams data can be classified into two categories. The first category follows a “whitebox” architecture as illustrated in Figure 3.2.

---

\(^3\)http://streamreasoning.org/larkc/csparql/CSPARQL-ReadyToGoPack-0.7.4.zip released on 9.5.2012
From the semantics formalised in Section 3.1, this architecture needs to implement the physical operators such as sliding windows, join, and triple pattern matching. Such operators can be implemented using techniques and algorithms described in Section 2.1.4. To consume data, these operators use access methods which may employ data structures such as B+Trees++, hash tables, or triple-based indexes for fast random data access to RDF datasets or RDF streams. To execute a declarative query in an SPARQL-like language, the optimiser has to translate it to a logical query plan, and then find an optimal execution plan based on the corresponding operator implementations. This execution plan is then executed by the Executor. As the continuous query is executed continuously, the Optimiser must re-optimise it to find a new execution plan to adapt to the changes in the data and the computing environment, if a processing cost threshold is exceeded. In the whitebox architecture, the query engine needs to implement all the components of a DSMS. Streaming SPARQL and CQELS follow a whitebox approach. To avoid implementing most of these components, an alternative is the “blackbox” architecture shown in Figure 3.3.

The blackbox architecture uses existing systems as sub-components for the processing needed. Usually, the chosen sub-components are accessed with different query languages and different input data formats. Hence, the blackbox approach needs a Query Rewriter, an Orchestrator and a Data Transformer. The Query Rewriter rewrites a SPARQL-like query to sub-queries that the underlying systems can understand. The Orchestrator is used to orchestrate the execution process by externalising the processing to sub-systems with the rewritten sub-queries. In some cases, the Orchestrator also includes some components for correlating and aggregating partial results returned from the blackbox systems if they support this. The Data Transformer is responsible for converting input data to the compatible formats of the Access methods used in the sub-components. The Data Transformer also has to transform
3.3. ARCHITECTURAL DESIGN

Figure 3.3: The blackbox architecture.

<table>
<thead>
<tr>
<th>Data transformation</th>
<th>Query rewriter</th>
<th>Underlying systems</th>
</tr>
</thead>
<tbody>
<tr>
<td>C-SPARQL RDF→Relation</td>
<td>C-SPARQL→SPARQL&amp;CQL</td>
<td>SPARQL engine&amp;STREAM or ESPER</td>
</tr>
<tr>
<td>SPARQLstream Relation→RDF</td>
<td>SPARQLstream→SNEEql</td>
<td>SNEE engine</td>
</tr>
</tbody>
</table>

Table 3.1: Current systems that implement the blackbox architecture.

By delegating the processing to available systems, building a system following the blackbox architecture takes less effort than using the whitebox approach. However, such systems do not have full control over the sub-components, e.g., for optimisation purposes, and they suffer from performance costs and lower throughputs. This will be discussed in details in Chapter 5. Table 3.2 gives the design choices of all available systems including our own system, the CQELS engine.

<table>
<thead>
<tr>
<th>Architecture</th>
<th>Access methods</th>
<th>Execution</th>
</tr>
</thead>
<tbody>
<tr>
<td>Streaming SPARQL</td>
<td>whitebox</td>
<td>Physical RDF streams</td>
</tr>
<tr>
<td>C-SPARQL</td>
<td>blackbox</td>
<td>Relations&amp;Views</td>
</tr>
<tr>
<td>EP-SPARQL</td>
<td>blackbox</td>
<td>Logic predicates</td>
</tr>
<tr>
<td>SPARQLstream</td>
<td>blackbox</td>
<td>Web services</td>
</tr>
<tr>
<td>CQELS</td>
<td>whitebox</td>
<td>Native data structures</td>
</tr>
</tbody>
</table>

Table 3.2: System design choices.
3.4 System Design and Implementation

In this section, we provide an extensive and comparative analysis of the state-of-the-art Linked Stream Data processing engines, their different design choices and their solutions to address the different issues. Our CQELS system which will be introduced in Chapter 5 is added to the comparison for later reference.

**Streaming SPARQL**

Streaming SPARQL [Bolles et al., 2008] extends the ARQ query compiler to transform the continuous query to a logical query plan. This logical query plan is then compiled to a physical execution plan composed of physical query operators. Streaming SPARQL extends SPARQL 1.0 for representing continuous queries on RDF Streams. For implementing the physical operators, Streaming SPARQL uses the approaches proposed by [Krämer and Seeger, 2009]. However, to the best of our knowledge, the implementation of the system is not publicly available. Therefore, we could not include it into our evaluations. Streaming SPARQL suggests to apply algebraic optimisation after translating a declarative query to a logical query plan.

**C-SPARQL**

C-SPARQL implements a blackbox architecture as shown in Figure 3.4. The C-SPARQL engine uses a SPARQL plugin to connect to a SPARQL query engine to evaluate the static part of the query, i.e., the sub-queries involving the RDF datasets. For evaluating the parts of the query relevant to streams and aggregates, the engine delegates the processing to an existing relational data stream management system. One limitation of this architecture is that aggregations can only be performed by the DSMS using CQL. A C-SPARQL query is first parsed and sent to the orchestrator. The orchestrator is the central component of C-SPARQL engine and it translates the query into a static and a dynamic part. The static sub-queries are used to extract the static results from the SPARQL engine, while the dynamic sub-queries are registered in the DSMS. The query initialisation is executed only once when a C-SPARQL query is registered as the continuous evaluation is handled subsequently by the DSMS. Therefore, C-SPARQL does not support updates in the non-stream data. The evaluation process of the C-SPARQL engine is illustrated in Figure 3.5.
The results returned from the SPARQL engine for the static part of the query are loaded into materialised relations as inputs for the DSMS. These relations together with RDF streams are computed via cascading views created as CQL queries [Arasu et al., 2006], driven by the C-SPARQL query. The first views in the query pipeline are sliding window views over RDF Streams. They are then correlated with the static relations via join views. As C-SPARQL employs algebraic optimisation, it tries to filter the data as early as possible. Hence, it pushes the filters to the bottom of the query pipeline by rewriting rules [Smith and Chang, 1975]. The order of the evaluation process illustrates how views are created on top each other. At the time of this writing, a version of C-SPARQL based on Jena\(^5\) and ESPER\(^6\) can be downloaded at [http://streamreasoning.org/download](http://streamreasoning.org/download).

---

\(^5\)http://incubator.apache.org/jena/
\(^6\)http://esper.codehaus.org/
EP-SPARQL

EP-SPARQL uses a blackbox approach backed by a logic engine. It translates queries into logic programs which are then executed by a Prolog engine. The execution mechanism of EP-SPARQL is based on event-driven backward chaining (EDBC) rules, introduced in [Anicic et al., 2010]. EP-SPARQL queries are compiled into EDBC rules, which enable timely, event-driven, and incremental detection of complex events (i.e., answers to EP-SPARQL queries). EDBC rules are logic rules, and
Fig. 1. Ontology-based streaming data access service

As such, it enables the detection of real-time situations that are identified based on explicit data (e.g., events) as well as on implicit knowledge (derived from the background knowledge). The background knowledge may be specified as a Prolog knowledge base or as an RDFS ontology. This enables EP-SPARQL’s execution model to have all relevant parts expressible in a unified (logic rule) formalism, and ultimately to reason over a unified space. At the time of writing, an implementation of EP-SPARQL is available at http://code.google.com/p/etalis/.

SPARQLstream

SPARQLstream is designed for supporting the integration of heterogenous relational stream data sources. The SPARQLstream engine rewrites the SPARQLstream query language to a relational continuous query language, e.g., SNEEql [Galpin et al., 2008]. Its architecture is shown in Figure 3.6.

Figure 3.6: Architecture of SPARQLstream.

In order to transform the SPARQL\textsubscript{stream} query, expressed in terms of the ontology, into queries in terms of the data sources, a set of mappings must be specified. These mappings are expressed in the S\textsubscript{2}O mapping language, an extension of the R\textsubscript{2}O mapping language [Calbimonte et al., 2010]. It supports streaming queries and data, most notably window and stream operators.

After the continuous query has been generated, the query processing phase starts. The evaluator uses distributed query processing techniques [Kossmann, 2000] to extract the relevant data from the sources and to perform the required query processing, i.e., selection, projection, and joins. Query execution in sources such as sensor networks may include in-network query processing, pull or push based delivery of data between sources, and other data source specific settings. The result of the query processing is a set of tuples that the data translation process transforms into ontology instances. At the time of writing, a version of SPARQL\textsubscript{stream} is available at http://code.google.com/p/semanticstreams/wiki/SPARQLStream.

**System Comparison**

Table 3.3 gives a comparison of the systems described according to the features supported. Unlike the other systems, SPARQL\textsubscript{stream} takes relational streams as input and not RDF streams. C-SPARQL, EP-SPARQL and CQELS support the correlation of RDF streams and RDF datasets. All systems extend SPARQL for stream processing, but each of the languages supports some special patterns as shown in the column “Special support for”. TF stands for supporting built-in time functions in the query patterns. EVENT denotes the support for event-based patterns. NEST means that the engine support nested queries. VoS indicates that the query language allows using variable for the Stream’s URI. The “Extras” column shows the extra features supported by the corresponding engines.

<table>
<thead>
<tr>
<th>System</th>
<th>Input</th>
<th>Special support for</th>
<th>Extras</th>
</tr>
</thead>
<tbody>
<tr>
<td>Streaming SPARQL</td>
<td>RDF stream</td>
<td></td>
<td></td>
</tr>
<tr>
<td>C-SPARQL</td>
<td>RDF Stream &amp; RDF</td>
<td>TF</td>
<td></td>
</tr>
<tr>
<td>EP-SPARQL</td>
<td>RDF Stream &amp; RDF</td>
<td>EVENT, TF</td>
<td>Event operators</td>
</tr>
<tr>
<td>SPARQL\textsubscript{stream}</td>
<td>Relational stream</td>
<td>NEST</td>
<td>Ontology-based mapping</td>
</tr>
<tr>
<td>CQELS</td>
<td>RDF Stream &amp; RDF</td>
<td>VoS, NEST</td>
<td>Disk spilling</td>
</tr>
</tbody>
</table>

*Table 3.3: System comparison by features.*

To compare the execution mechanisms, Table 3.4 categorises the systems by architecture, re-execution strategy, how the engine schedules the execution, and what type of optimisation is supported. Since Streaming SPARQL and C-SPARQL schedule the execution at a logical level, the optimisation can only be done at an algebraic level and statically. In contrast, CQELS is able to choose alternative execution plans composed from available physical implementations of operators, thus, the optimiser can adaptively optimise the execution at physical level. EP-SPARQL and SPARQL\textsubscript{stream} schedule the execution via a declarative query or a logic program, so they completely delegate the optimisation to other systems.

In Chapter 5, we will present a thorough performance evaluation of all these systems. The evaluation consists of several experiments to compare the performance in terms of query execution time, scalability, etc.
3.5 Summary

This chapter overviewed Linked Stream Data processing in terms of the concepts applied in the state-of-the-art systems. It highlighted some issues in formalising the data model and semantics of the query language. Temporal aspects of RDF data have to be taken into account and the continuous semantics of queries need to be defined along with the semantic of the query operators. There are also some language ambiguity issues in Streaming SPARQL, C-SPARQL, SPARQL_stream, and EP-SPARQL. These issues are discussed in the following chapters. Our conjecture is that the systems that use “blackbox” approach might suffer from performance problems. To show that this conjecture holds, we will present the first extensive performance comparisons in Chapter 5. This will show the advantages of the “whitebox” approach in building Linked Data Stream processing engines.

<table>
<thead>
<tr>
<th></th>
<th>Architecture</th>
<th>Re-execution</th>
<th>Scheduling</th>
<th>Optimisation</th>
</tr>
</thead>
<tbody>
<tr>
<td>Streaming SPARQL</td>
<td>whitebox</td>
<td>periodical</td>
<td>Logical plan</td>
<td>Algebraic &amp; Static</td>
</tr>
<tr>
<td>C-SPARQL</td>
<td>blackbox</td>
<td>periodical</td>
<td>Logical plan</td>
<td>Algebraic &amp; Static</td>
</tr>
<tr>
<td>EP-SPARQL</td>
<td>blackbox</td>
<td>eager</td>
<td>Logic program</td>
<td>Externalised</td>
</tr>
<tr>
<td>SPARQL_stream</td>
<td>blackbox</td>
<td>periodical</td>
<td>External call</td>
<td>Externalised</td>
</tr>
<tr>
<td>CQELS</td>
<td>whitebox</td>
<td>eager</td>
<td>Adaptive physical plans</td>
<td>Physical &amp; Adaptive</td>
</tr>
</tbody>
</table>

Table 3.4: Comparisons by execution mechanism.
Chapter 4

A Model for Linked Stream Data Processing

This chapter describes a novel processing model for Linked Stream Data processing. We introduce a data model to represent both RDF-based streams and RDF datasets. Three classes of operators for processing data in this model are formally defined using notations from denotational semantics [Stoy, 1977]. The operational semantics of the processing model is described using structural operational semantics notations [Plotkin, 2004].

4.1 Data Model and Query Semantics

To define our new data model and query semantics, we reuse notations and definitions from Section 2.2.1, namely, RDF triple, triple pattern, mapping, and operations on mappings.

**RDF Stream and Instantaneous RDF dataset.** In continuous query processing over dynamic data, the temporal nature of the data is crucial and needs to be captured in the data representation. This applies to both types of data sources, since updates in Linked Data collections are also possible. We define RDF streams to represent Linked Stream Data, and we model Linked Data by generalising the standard definition of RDF datasets to include the temporal aspect. Thereby:

1. An RDF dataset at timestamp $t$, denoted by $G(t)$, is a set of RDF triples valid at time $t$ and called instantaneous RDF dataset. An RDF dataset is a sequence $G = [G(t)], t \in \mathbb{N}$, ordered by $t$. When it holds that $G(t) = G(t + 1)$ for all $t \geq 0$, we call $G$ a static RDF dataset and denote it as $G^s = G(t)$.

2. An RDF stream $S$ is a bag of elements $\langle (s, p, o) : [t] \rangle$, where $(s, p, o)$ is an RDF triple and $t$ is a timestamp. $S_{\leq t}$ denotes the bag of elements in $S$ with timestamps $\leq t$, i.e., $\{(s, p, o) : [t'] \} \in S | t' \leq t$.

The main primitive operation on RDF stream and instantaneous RDF dataset is pattern matching, which is extended from the triple pattern of SPARQL semantics [Pérez et al., 2009a]. Each output of a pattern matching operator consists of a set of mappings. Following [Arasu and Widom, 2004b], we use notations of denotational semantics [Stoy, 1977] to formally define query patterns of our processing model. The query patterns are represented in BNF-style as abstract syntax. In Section 4.2, this abstract
syntax is realised as a SPARQL-extension based on SPARQL 1.1 grammar. The semantics of the compositions of abstract syntax are formalised by the meaning functions, called denotations. Those compositions consist of three classes of operators: namely, pattern matching, relational, and streaming operators.

Recursively, a denotation of a query is constructed from denotations of its parts, i.e. operators. For example, consider a construct “$P_1 \text{ AND } P_2$”. Its meaning is defined by the meanings of the three fragments $P_1$, $P_2$ and AND. Similar to CQL [Arasu and Widom, 2004a], we use the lambda calculus [Turbak and Gifford, 2008] to represent the meaning functions of these operators. The lambda calculus expression $\lambda.x_1...x_n.F$ is used to represented a function that takes $n$ arguments $v_1, ..., v_n$ and return the result of evaluating the function $F$ with all free occurrences of $x_i$ in $F$ replaced by $v_i$, $1 \leq i \leq n$. The argument $v_i$ and the returned result could themselves be functions, i.e., lambda calculus expressions.

**Pattern Matching Operators.** Listing 4.1 shows the abstract syntax of pattern matching operators. These operators extract triples from an RDF stream or dataset that match a given triple pattern and are valid at a certain time $t$.

| $P_G$ | ::= $<$ IRI $>$ $<$ Triple Pattern $>$ |
| $P^o_G$ | ::= $<$ IRI $>$ $<$ Window $\omega$ $>$ $<$ Triple Pattern $>$ |
| $\text{PM}_{\text{op}}$ | ::= $P_G$ | $P^o_G$ |

**Listing 4.1:** Abstract syntax for Pattern Matching Operators.

Similar to SPARQL, we define the meaning of the triple matching pattern operator $P_G$ on an RDF dataset at timestamp $t$ as

$$[P_G](G,t) \overset{\text{def}}{=} \lambda G.\lambda t.\{\mu \mid \text{dom}(\mu) = \text{var}(P) \land \mu(P) \in G(t)\}.$$ (4.1)

The meaning of the window-based triple matching operator $P^o_G$ on an RDF stream is then defined by extending the operator above as follows.

$$[P^o_G](S,\omega,t) \overset{\text{def}}{=} \lambda S.\lambda \omega.\lambda t.\{\mu \mid \text{dom}(\mu) = \text{var}(P) \land \langle \mu(P) : [t'] \rangle \in S \land t' \in \omega(t)\}.$$ (4.2)

where $\omega(t) : \mathbb{N} \rightarrow 2^{\mathbb{N}}$ is a function mapping a timestamp to a (possibly infinite) set of timestamps. This gives us the flexibility to choose between the different window modes introduced in Section 2.1.2. For example, a time-based sliding window of size $T$ can be expressed as $\omega_{\text{RANGE}}(t) = \{t' \mid t' \leq t \land t' \geq \text{max}(0, t - T)\}$, and a window that extracts only events happening at the current time corresponds to $\omega_{\text{NOW}}(t) = \{t\}$. Moreover, we can similarly define triple-based windows that returns the latest $N$ triples ordered by the timestamps.

**Relational Operators.** Due to the composability of denotational semantics we define the abstract syntax for compound query patterns constructed from the above pattern matching operators and relational operators.

Before defining the syntax and its semantics, we define the aggregation operator that is not defined in [Pérez et al., 2009a]. If a set of mappings contains only mappings having the same domain, we call it a uniform mapping set. We also denote dom($\Omega$) of a uniform mapping set $\Omega$ as the domain of its mapping. Similar to the aggregate operator in SQL, we define the aggregate operator on a uniform
mapping set $\Omega$ as

$$G_1, G_2, \ldots, G_m, \text{AGG} f_1(A_1), f_2(A_2), \ldots, f_k(A_k)(\Omega) : G_i \in \text{dom}(\Omega) \land A_i \in \text{dom}(\Omega)$$

where $f_i(A_i)$ is an aggregate function such as MIN, MAX, COUNT, AVERAGE on variable $A_i$ of every mapping contained in $\Omega$, and $G_i$ is the variable used to group values for computing aggregates like the "GROUP BY" clause of SQL.

The abstract syntax for the for relational operators $R_{op}$ are defined recursively as shown in Listing 4.2. Note that $F$ is a SPARQL built-in filter condition introduced in Section 2.2.1.

Let $R^1_{op}$, $R^2_{op}$ and $R_{op}$ be relational operators. The meanings of the operators $\text{AND}_{op}$, $\text{UNION}_{op}$, $\text{OPT}_{op}$, $\text{FILTER}_{op}$ and $\text{AGG}_{op}$ are then defined as follows.

$$[\text{AND}_{op}](R^1_{op}, R^2_{op}, t) \equiv \lambda R^1_{op}.\lambda R^2_{op}.\lambda t.\{[R^1_{op}](t) \cup [R^2_{op}](t)\}$$  \hspace{1cm} (4.3)

$$[\text{UNION}_{op}](R^1_{op}, R^2_{op}, t) \equiv \lambda R^1_{op}.\lambda R^2_{op}.\lambda t.\{[R^1_{op}](t) \cup [R^2_{op}](t)\}$$  \hspace{1cm} (4.4)

$$[\text{OPT}_{op}](R^1_{op}, R^2_{op}, t) \equiv \lambda R^1_{op}.\lambda R^2_{op}.\lambda t.\{[R^1_{op}](t) \supset [R^2_{op}](t)\}$$  \hspace{1cm} (4.5)

$$[\text{FILTER}_{op}](R_{op}, t) \equiv \lambda R_{op}.\lambda t.\{\mu \in [R_{op}](t) \mid \mu \models F\}$$  \hspace{1cm} (4.6)

$$[\text{AGG}_{op}](R_{op}, t) \equiv \lambda R_{op}.\lambda t.\{G_1, G_2, \ldots, G_m, \text{AGG} f_1(A_1), f_2(A_2), \ldots, f_k(A_k)(R_{op}(t))\}$$  \hspace{1cm} (4.7)

**Streaming Operators.** Similar to the relation-to-stream operator of CQL [Arasu et al., 2006], we define streaming operators. The output of a streaming operator is either a relational stream or an RDF stream from the above relational operators. Listing 4.3 shows the abstract syntax of the streaming operators.

For operators generating relational streams of SPARQL results, $S^{IS}$ and $S^{DS}$ are used for streaming only the new outputs and expired outputs, respectively, and $S^{RS}$ is used for repeatedly streaming the current outputs at time point $t$. The semantics of $S^{IS}$, $S^{DS}$ and $S^{RS}$ are defined as follows.

$$[S^I](R_{op}, t) \equiv \lambda R_{op}.\lambda t.\{\mu, f(t) : \mu \in [R_{op}](t) \land \mu \notin [R_{op}](t-1)\}$$  \hspace{1cm} (4.8)
To construct RDF streams from relational streams generated by these operators, we introduce the RDF streaming $S^RDF_{\text{op}}$. From a graph template $T$ that provides a set of triple patterns, called a graph pattern, the semantics of the RDF streaming operator $S^RDF_{\text{op}}$ is defined as

$$[S^RDF_{\text{op}}](T, S_{\text{op}}) \overset{\text{def}}{=} \lambda T. \{ \bigcup_{t \geq 0} \{(\mu, t) | \mu \in S_{\text{op}}(t) \wedge P \in T\} \}$$

(4.11)

where $f : \mathbb{N} \rightarrow \mathbb{N}$ is a function that maps $t$ to a new timestamp to indicate when the results should be streamed out. In the simplest case, $f$ is the identical function, indicating that triples are streamed out immediately.

### 4.2 CQELS Query Language

From the abstract syntaxes of the operators introduced in Section 4.1, we now define a declarative query language (CQELS query language, or CQELS-QL) for our CQELS execution framework which will be described in Chapter 5. The grammar of the CQELS-QL is defined from the SPARQL 1.1 grammar in the EBNF notation. First, we add a query pattern to represent window operators on RDF Stream, called \textit{StreamGraphPattern}, into the \textit{GraphPatternNotTriples} pattern of SPARQL 1.1.

$$\text{GraphPatternNotTriples} ::= \text{GroupOrUnionGraphPattern} | \text{OptionalGraphPattern} | \text{MinusGraphPattern} | \text{GraphGraphPattern} | \text{StreamGraphPattern} | \text{ServiceGraphPattern} | \text{Filter} | \text{Bind}$$

Assuming that each stream has an IRI as identifier, the \textit{StreamGraphPattern} pattern is defined as follows.

$$\text{StreamGraphPattern} ::= \text{‘STREAM’} ‘[‘ Window ‘]’ \text{VarOrIRIref} ‘{‘TriplesTemplate’}’$$

- \text{Window} ::= \text{Range} | \text{Triplet} | \text{‘NOW’} | \text{‘ALL’}
- \text{Range} ::= \text{‘RANGE’} \text{Duration} (\text{‘SLIDE’} \text{Duration})?
- \text{Triplet} ::= \text{‘TRIPLES’} \text{INTEGER}
- \text{Duration} ::= (\text{INTEGER} ‘d’ | ‘h’ | ‘m’ | ‘s’ | ‘ms’ | ‘ns’)?

where \text{VarOrIRIRef} and \text{TripletTemplate} are patterns for the variable/IRI and triple template from SPARQL 1.1. \text{Range} corresponds to a time-based window, while \text{Triplet} corresponds to a triple-based window. The keyword \text{SLIDE} is used for specifying the sliding parameter of a time-based window, whose time interval is given by \text{Duration} (c.f. Section 2.1.2).

As the relational operators are defined based on their corresponding operators in SPARQL, the grammar of SPARQL1.1 is reused. For relational streaming operators, we modify the SELECT grammar with the new modifiers STREAM, DSTREAM and RSTREAM corresponding to the abstract syntax in Listing 4.3. If the modifier of the SELECT keyword is omitted, it implicitly means SELECT ISTREAM. Similarly, we extend the CONSTRUCT grammar for being able to represent the RDF streaming operator. Given the five queries from the conference scenario described in Example 8 from Section 3.2, we can represent them in the CQELS-QL as shown below. In these examples, $\$\text{Name}\$ is replaced by a constant representing an author’s name when instantiating the query.

1\http://www.w3.org/TR/sparql11-query/#grammar
4.2. CQELS QUERY LANGUAGE

Query Q1

```
PREFIX lv: <http://deri.org/floorplan/>
PREFIX dc: <http://purl.org/dc/elements/1.1/>
PREFIX foaf: <http://xmlns.com/foaf/0.1/>
CONSTRUCT {?person1 lv:reachable ?person2}
FROM NAMED <http://deri.org/floorplan/>
WHERE {
  STREAM <http://deri.org/streams/rfid> [NOW] {?person1 lv:detectedat ?loc1}
  GRAPH <http://deri.org/floorplan/> {?loc1 lv:connected ?loc2}
}
```

Query Q2

```
PREFIX lv: <http://deri.org/floorplan/>
PREFIX dc: <http://purl.org/dc/elements/1.1/>
PREFIX foaf: <http://xmlns.com/foaf/0.1/>
SELECT ?coAuthorName
FROM NAMED <http://deri.org/floorplan/>
WHERE {
  FILTER (?auth != ?coAuth)
}
```

Query Q3

```
PREFIX lv: <http://deri.org/floorplan/>
PREFIX dc: <http://purl.org/dc/elements/1.1/>
PREFIX foaf: <http://xmlns.com/foaf/0.1/>
PREFIX dcterms: <http://purl.org/dc/terms/>
PREFIX swrc: <http://swrc.ontoware.org/ontology#>
SELECT ?editorName
WHERE {
  GRAPH <http://deri.org/floorplan/> {?loc1 lv:connected ?loc2
  ?auth foaf:name "$Name$"}
}
```

Query Q4

```
PREFIX lv: <http://deri.org/floorplan/>
PREFIX dc: <http://purl.org/dc/elements/1.1/>
PREFIX foaf: <http://xmlns.com/foaf/0.1/>
SELECT ?loc2 ?locName ?noCoAuths
FROM NAMED <http://deri.org/floorplan/>
WHERE {
  GRAPH <http://deri.org/floorplan/> {?loc1 lv:connected ?loc2
  FILTER (?auth != ?coAuth)
  GROUP BY ?loc2 ?locName
}
```
4.3 Adaptive Query Execution Model

The denotational representations of the query operators define the semantics of a query, but they do not define the operational aspect of a query. Therefore, this section provides the operational semantics of our adaptive execution model. This execution model shows how a query represented in CQELS-QL is executed in CQELS execution framework from Chapter 5. To highlight the importance of the adaptivity for a query execution model for continuous queries, we start with investigating the following example.

Example 9: For the conference scenario in Section 1.4, assume that we have the information about direct connectivity between the rooms, in a static RDF dataset \(G\) with triples of the form \(P = (?loc_1, \text{conn}, ?loc_2)\), where \(G^S = \{(r_1, \text{conn}, r_2), (r_1, \text{conn}, r_3), (r_2, \text{conn}, r_1), (r_3, \text{conn}, r_1)\}\). Also assume that people’s locations are provided in a single stream \(S\) with triples of the form \( (?person, \text{detectedAt}, ?loc)\). Consider the query \(Q_2\) from Example 8 which notifies two people when they can reach each other from two different and directly connected rooms.

According to Definitions 4.1- 4.11, to execute this query, we apply the matching operator on the static dataset \(G\) with the triple pattern \(P\) to find pairs of rooms which can be reached from each other. Then we extract two windows from stream \(S\) using the windowing functions \(\omega_1 = \omega_{\text{NOW}}\) and \(\omega_2 = \omega_{\text{RANGE}}\). The former looks at the latest detected person, and the latter monitors people during the last \(T\) logical clock ticks by which we can assume that they are still in the same room. For the example, let us assume \(T = 2\). Let \(W_1(t)\) and \(W_2(t)\) be the outputs of the window operators corresponding to \(\omega_1\) and \(\omega_2\) at time \(t\). We use the triple patterns \(P_i = (?person_i, \text{detectedAt}, ?loc_i)\) for \(i = 1, 2\) in the window operators. Hence, the mappings in \(W_i\) are of the form \(\{(?person_i \mapsto \text{id}, ?loc_i \mapsto \text{lid}\}\). The lambda expressions for these three operators are given in Equations 4.12, 4.13 and 4.14. We use \(\text{AND}_{op}\) to correlate the outputs of these three operators before generating an “ISTREAM” with the \(S^{IS}\) operator. To generate the RDF stream with the template \(T = (?person_1, \text{reaches}, ?person_2)\), we apply the RDF stream operator as shown in Equation 4.15. By replacing the expression of \(\text{AND}_{op}\) (Definition 4.6), we can represent the query \(Q\) as shown in Equation 4.16.

\[
P(t) = \lambda t. (?loc_1, \text{conn}, ?loc_2)_G(G^S(t)) = (?loc_1, \text{conn}, ?loc_2)_G(G^S) \quad (4.12)
\]
\[
W_1(t) = \lambda t. (?person_1, \text{detectedAt}, ?loc_1)_S(LOC_{\text{stream}}, \omega_{\text{NOW}}) \quad (4.13)
\]
\[
W_2(t) = \lambda t. (?person_2, \text{detectedAt}, ?loc_2)_S(LOC_{\text{stream}}, \omega_{\text{RANGE}}) \quad (4.14)
\]
\[
Q(t) = \lambda t. S^{RDF}(T, S^{IS}(\text{AND}_{op}(\text{AND}_{op}(W_1(t), W_2(t), t), P(t), t), t)) \quad (4.15)
\]
\[
Q(t) = \lambda t. S^{RDF}(T, S^{IS}((W_1(t) \bowtie W_2(t)) \bowtie P(t))) \quad (4.16)
\]

To illustrate this query, Table 4.1 shows the input/output of \(Q\) as time progresses (we use the following abbreviations: \(dA\) for \(\text{detectedAt}\), \(p\) for \(?person\), and \(?ℓ\) for \(?loc\)).

To compute \(Q(t)\) at time \(t\), there are two possible execution plans as illustrated in Figure 4.1. The first plan joins the outputs of the two windows before joining with those of \(P(t)\). The second plan joins the outputs of \(P(t)\) with \(W_1(t)\) before joining with outputs of \(W_2(t)\).

Figure 4.2 shows the intermediate results generated by both execution plans after the first join. The plan with the smallest number of results is highlighted. Usually, these are preferred since less intermediate results are often more efficient. Two simplified versions of execution plans \(P_1\) and \(P_2\) are on the left.
### 4.3. Adaptive Query Execution Model

<table>
<thead>
<tr>
<th>t</th>
<th>S</th>
<th>W₁</th>
<th>W₂</th>
<th>Sout</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>((m₀, dₐ, r₁) : [0])</td>
<td>{?p₁ → mₐ, ?l₁ → r₁}</td>
<td>{?p₂ → m₀, ?l₂ → r₁}</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>((m₁, dₐ, r₁) : [1])</td>
<td>{?p₁ → m₁, ?l₁ → r₀}</td>
<td>{?p₂ → m₁, ?l₂ → r₂}</td>
<td>((m₀, reaches, m₁) : [1])</td>
</tr>
<tr>
<td>2</td>
<td>((m₁, dₐ, r₁) : [1])</td>
<td>{?p₁ → m₁, ?l₁ → r₁}</td>
<td>{?p₂ → m₁, ?l₂ → r₀}</td>
<td>((m₁, reaches, m₂) : [1])</td>
</tr>
<tr>
<td>3</td>
<td>((m₃, dₐ, r₂) : [3])</td>
<td>{?p₁ → m₃, ?l₁ → r₀}</td>
<td>{?p₂ → m₃, ?l₂ → r₂}</td>
<td>((m₂, reaches, m₃) : [3])</td>
</tr>
</tbody>
</table>

**Table 4.1:** Input and output of Q as time progresses.

![Diagram of Execution Plan P₁ and P₂](image)

**Figure 4.1:** Possible execution plans for the query Q2.

On the right-hand side, we show the input/output of the join operators ⊙₁₂ and ⊙₄₁ which generate the intermediate results. At timestamp 0, |W₁(0)| = |W₂(0)| = 1 as the first triple is streamed into the system. It is preferable at this point to use P₁, i.e., to join W₁(0) with W₂(0) using ⊙₁₂ because the intermediate result ⊙₁₂(0) has size 1. If we follow P₂, then joining W₁(0) with P₂(0) using ⊙₄₁ yields ⊙₄₁(0) with size 2. However, at t = 1, D₂ is preferable because |⊙₁₂(1)| = 2 and |⊙₄₁(1)| = 1.

This example indicates that to enable efficient continuous computation, the query execution plan may need to be changed dynamically to adapt to changes in the input data. Therefore, we propose an abstract query execution model that supports such adaptability. This model is represented by the notations of structural operational semantics (SOS) [Plotkin, 2004]. SOS defines operational semantics of a program, i.e., the program or execution plan which executes a continuous query, by the behaviors of its parts. Such behaviors are defined as a set of transition relations, which play the role of inference rules defining the valid transitions of the composite program via transitions of its components. The behavior of our adaptive query execution model is given by the following transition rule:
\[ \langle \text{NextExecPlan}(Q), s^t \rangle \Rightarrow P_Q : \langle P_Q, s^t \rangle \rightarrow s'^t \]

(4.17)

Intuitively, this rule says that if the execution engine chooses \( P_Q \) as the next execution plan to execute at state \( s^t \), and given that the execution plan \( P_Q \) in state \( s^t \) finishes in state \( s'^t \), then the continuous programme of \( Q \) at state \( s'^t \) can be transformed to the programme that executes \( P_Q \) then executes \( Q \) at state \( s'^t \). The states \( s^t \) and \( s'^t \) are used to model system states of the execution engine at certain stages of the computation. A system state includes the input stream data as time progresses. It also contains data structures used in the execution plans, such as indexes and caches. Monitoring data such as statistics of input data, executions, and resource usage, can be counted as system states. By taking into account the system state, the engine is able to choose the next execution plan for the continuous query by the function \( \text{NextExecPlan}(Q) \). For example, by using statistics from input streams in the states at time 0 and 1, the engine can determine the execution plan that generates the lowest amount intermediate results at time 0 and 1. The function \( \text{NextExecPlan}(Q) \) ranges over all the equivalent execution plans supported by the execution engine. This function is used to implement adaptive algorithms for each instance of our adaptive query execution model. Building such adaptive algorithms is a difficult task that will be discussed in detail in Chapter 6 and 7.
This chapter presents CQELS (Continuous Query Execution over Linked Stream), an adaptive and native execution framework for Linked Stream Data and Linked Data. CQELS provides a flexible architecture for implementing efficient continuous query processing engines over Linked Data Stream and Linked Data by treating triples as first-class citizens. Next, we describe the CQELS framework and its components. An evaluation detailed at the end of this chapter highlights the advantages of CQELS over existing continuous query processing engines for Linked Stream Data but also identifies what needs to be done to achieve further better performance of CQELS over other engines.

5.1 CQELS Execution Framework

The CQELS execution framework supports native and adaptive query execution over RDF streams and RDF datasets. The whitebox architecture of the framework is illustrated in Figure 5.1. This architecture accepts RDF streams and RDF datasets as inputs and returns outputs as RDF streams or relational streams in the SPARQL Result format. The output RDF streams can be fed into any CQELS engine again, and the relational streams can be used by other relational stream processing systems. The processing works as follows: The stream data is pushed to the Input Manager and is then encoded by the Encoder into a normalised input stream representation which can be consumed by the Dynamic Executor. The outputs of the Dynamic Executor have to be decoded by the Decoder before being streamed to the receiver. The Encoder and Decoder share a Dictionary for the encoding and decoding operations. The static RDF datasets are accessed by the Dynamic Executor via the Cache Fetcher. They can be hosted in a local RDF store or remote RDF stores with SPARQL endpoints. The Cache Fetcher retrieves the necessary data and encodes it for the Cache Manager using the Encoder. This encoded data is represented in a normalised representation of intermediates results which share the same Dictionary with input stream. The Window Buffer Manager and Cache Manager are responsible for managing the input data for the RDF streams and RDF datasets respectively. The Adaptive Optimiser continuously selects the most efficient execution plans based on data statistics and operator costs.

The native property of the CQELS framework comes from the fact that the RDF nodes and triples are

---

1 [http://www.w3.org/TR/rdf-sparql-XMLres/](http://www.w3.org/TR/rdf-sparql-XMLres/)
treated as first-class citizens. The Encoder and Decoder employ the typical encoding techniques of triples storages [Abadi et al., 2007, Broekstra et al., 2002, Wilkinson et al., 2003, Harris, 2005, Owens et al., 2008] to store the data in efficient data structures for subsequent processing. The adaptivity of the framework is enabled by two components: The Dynamic Executor and the Adaptive Optimiser. The Dynamic Executor enables a dynamic query execution strategy where the query plan can be changed during the life time of a continuous query. The best query plan to be executed by the Dynamic Executor is defined by the Adaptive Optimiser. The Adaptive Optimiser is able to adapt to changes in the input data and execution environment by selecting the best query plan at each stage of the processing. In the next sections, we describe the components of the CQELS framework in detail.

Figure 5.1: Adaptive Execution Framework of CQELS.
5.1.1 Input Manager

As defined in Section 4.1, RDF streams consist of RDF triples with timestamps from a discrete, ordered time domain. The continuous semantics is based on a temporal order. We use logical timestamps to enable ordered logical clocks for local and distributed data sources as done in classic time-synchronisation approaches [Lamport, 1978]. The issues of distributed time synchronization and flexible time management are beyond the scope of this thesis. The techniques and solutions for such issues can be found in [Fidge, 1991, Mattern, 1989, Srivastava and Widom, 2004].

The Input Manager is responsible for receiving the input data and delivering it as triples with timestamps in an increasing time order. The input data arrives at the Input Manager as line-based, plain-text format extended from N-Triple. They can be of two types: With or without timestamps as shown below.

\[
\begin{align*}
< & \text{subject} > & < \text{predicate} > & < \text{object} > . \\
< & \text{subject} > & < \text{predicate} > & < \text{object} > < \text{timestamp} > .
\end{align*}
\]

The timestamp field should be in well-formed DateTime format as in the N-Triple standard. The Input Manager parses the lines of stream inputs and stores them in an internal buffer before sending them in the proper order to the Encoder. If a triple arrives to the Input Manager without a timestamp, it will be assigned a system timestamp. For the timestamped triples, we use the heartbeat approach [Srivastava and Widom, 2004] to dispatch triples to the Window Buffer Manager. By using this approach, the timestamped triples do not necessarily have to arrive at the Input Manager in increasing order of timestamps. To dispatch a group of unordered triples in the buffer at a certain wall-clock time, the Input Manager generates a heartbeat \( t \), called maximum tolerant application timestamp. All triples with timestamps smaller than the heartbeat are sorted and sent to the Encoder.

5.1.2 Data Encoder and Decoder

As commonly used in triple stores [Abadi et al., 2007, Broekstra et al., 2002, Wilkinson et al., 2003], data encoding (RDF nodes into integers) is used to substantially improve the performance compared to storing RDF nodes in a lexical form. By encoding RDF nodes, most of query operators are executed on small, fixed size integers rather than large variable length strings, meaning less complex workload, smaller memory footprint, and a large improvement in cache efficiency, as well as reduced I/O time in many cases [Owens, 2011].

RDF nodes are encoded as 64 bit node identifiers (IDs). RDF triples and mappings are encoded as a series of these identifiers. Since it is desirable to eliminate expensive node ID to RDF node conversions whenever possible, node IDs represent literals of certain datatypes (called inline). Node IDs are comprised of 8 bits of type information, and 56 bits of node reference, which allows literals that can be encoded in 56 bits or less to be inlined. The XML Schema Datatypes integer (and derived types), decimal, datetime, date and boolean are represented directly in the 56 bit section if possible. For example, an XSD dateTime including the timezone with millisecond resolution can be encoded over a range of 8000 years. Along with URIs, RDF literals whose values are outside the encoded range are stored in a dictionary, as are RDF literals with illegal lexical forms for the datatype [Owens et al., 2008]. The 54 bit node reference is used as the key to look up the lexical version of the RDF node.

The dictionary provides unique node references that are continuously created during the lifetime of the queries. Consequently, the size of the dictionary is unpredictable and it may not fit to main memory.

\(^{2}\)http://www.w3.org/2001/sw/RDFCore/ntriples/
Therefore, the dictionary might have to be stored in disk-based node stores, e.g., mapping tables or key-value stores. The node reference itself can be, under normal circumstances, a disk address for retrieving a node serialisation. The placement of the disk address directly in the node ID has the following advantage: It allows the node store to use simple, fast, appending writes, and removes the requirement for an index over the node store. This eliminates an index lookup on the critical path of node ID to RDF node conversion. In order to allow the conversion of queries into their lexical form, it is necessary to allow nodes to be converted into node IDs. Hence, an index from node to node ID (henceforth called the node/ID index) is maintained to map the node reference of each node to its node ID [Harris, 2005].

Using a dictionary introduces an overhead that might be a bottleneck for the process of streaming encoded data to query operators. Therefore, throughputs of encoding and decoding data are critical for the performance of our execution engine. There are several techniques that can be employed to avoid a possible bottleneck. In RDF-based stream data, the frequencies of RDF nodes vary. For instance, URIs of predicates appear considerably more often than RDF literals. Hence, the dictionary can be partially cached in memory by assessing frequencies of RDF nodes. Thereby, the throughputs can be tuned by deciding how much memory is allocated to the dictionary cache.

Another way to adjust the throughput of the encoding is to choose the mechanism of ID generation. There are two possible mechanisms: Hash generation and incremental ID generation. Hash generation of IDs is faster but it provides no guarantees that prevent the generation of duplicate IDs. A collision can not be easily detected, while in event of such a collision, incorrect results might be retrieved. On the other hand, generating IDs in a incremental way is safer but slower. It requires a smaller ID space which can save space, but also requires an index to allow conversion from lexical form to ID. This index needs to be consulted for every encoding operation, thus it has a significant impact on the throughput of encoding [Owens, 2011].

Dictionaries of current RDF stores are designed for read optimisation. However, in the stream processing context, a large amount of data expires during the query life time. Furthermore, the expiration rate might be dramatically high. Therefore, the dictionaries used for the continuous query processing engine need to be able to purge IDs and associated data when they are no longer in use. However, keeping track of IDs when all the associated RDF triples and mappings have expired is time and space consuming. In some cases, the increase rate of the dictionary converges to zero and ID deletions are not needed. For instance, the GPS stream in the social network data integration scenario of Chapter 1 only generates new coordinates of users after the data from all users have been streamed. As new coordinates are encoded by inline IDs, thus, the dictionary does not grow. In contrast, if the dictionary grows too fast without deleting unnecessary IDs, the encoding and decoding throughput degrades rapidly.

5.1.3 Adaptive Optimiser

The Adaptive Optimiser has a compiler to translate declarative queries in CQELS-QL to a logical query plan. From the logical query plan the optimiser enumerates candidate physical query plans supported by the Dynamic Executor. In the CQELS framework, the Adaptive Optimiser provides abstract interfaces for implementing adaptive optimisation algorithms. According to stream processing requirements, similar to the Profiler in the StreaMon system [Babu and Widom, 2004], the Adaptive Optimiser continuously collects and maintains the statistics used by such algorithms. Based on these statistics, the adaptive optimisation algorithm deployed in the CQELS framework ensures that current plans and resource usage are the most efficient for the current running conditions. Running conditions
include input data, processing states and the system conditions. The system conditions, in turn, are all aspects of the executing environment that might affect the processing performance such as memory and CPU usage.

An optimisation algorithm uses statistics to compute cost metrics which determine the best physical query plan. The Dynamic Executor provides an interface for the Adaptive Optimiser to continuously monitor the incoming data and physical query operators in order to estimate the relevant cost metrics. By maintaining these statistics over time, the Adaptive Optimiser can detect when the currently used query plan is no longer the most efficient one by using a dynamic cost model on such changing statistics to define what the most efficient query plan is. The new most efficient query plan is identified by repeating the plan enumeration and cost computation based on the latest statistics. When a new query plan is elected, the Adaptive Optimiser will notify the Dynamic Executor to change the current execution plan by to newly selected one.

A common cost metric used in stream processing is unit-time cost [Kang et al., 2003]. Cost metrics are used to estimate the execution cost of query plans based on a cost model. The cost model provides functions to estimate the cost of each query operator and the cost of the query plan composed by such operators. For instance, Kang et al. [2003] propose formulas for computing unit-time costs of window joins, e.g., hash joins, nested loop joins, and nested loop hash joins. A cost model is chosen based on what parameters are being optimised. For instance, for optimising output rate, Viglas and Naughton [2002] introduce the rate-based cost metric and cost model. Along with several other cost models for stream processing [Ayad and Naughton, 2004, Cammert et al., 2008, Yang et al., 2007], some cost models for generic adaptive query processing can also be used to implement our Adaptive Optimiser. The cost model using join cardinality as a cost metric for adaptively reordering joins [Li et al., 2007] could be used to dynamically choose the best join order of triple patterns.

An adaptive optimisation algorithm introduces run-time overhead [Babu, 2005b]. The run-time overhead comes from maintaining statistics, computing costs of candidate query plans and migrating query plans. A good cost model should take into account such overhead to guarantee that the overall performance will not be harmed. The complexity of a query could make the optimisation impossible even for a traditional one-time query. Therefore, adaptive optimisation can only be applied in some subsets of a query such as multiway joins and filters. The cardinality-based optimisation technique will be introduced in Section 5.2.1 and the adaptive cost-based optimisation algorithms for multiway joins will be presented in Chapter 7.

### 5.1.4 Dynamic Executor

The Dynamic Executor provides interfaces for switching the current execution plan of a continuous query to a new one at a clock tick. It also enables the new query plan to re-use the processing state from the old execution plan [Zhu et al., 2004, Deshpande and Hellerstein, 2004, Raman et al., 2003]. For instance, in the MJoin implementation (c.f. Chapter 6), the new execution plan with a new order of probing can still re-use the indexes on inputs of the previous one. Sophisticated state migration mechanisms such as [Zhu et al., 2004, Deshpande and Hellerstein, 2004] are provided if the previous processing state can be partially reused. The processing state at a clock tick includes the input and intermediate data up to that clock tick. Input data is stored in the Window Buffer Manager and the Cache Manager, and is consumed by the leaf operators of query execution plans. The intermediate data is the data that is generated during the course of the execution of each plan. To provide a unified interface for accessing the intermediate and input data, we propose an abstract data structure similar to SweepArea of [Krämer and Seeger, 2009], called Abstract Window Buffer (AWB). An AWB is used
to store a bag of mappings which can be accessed by the interface with functions/procedures listed in
Algorithm 1. Each implementation of AWB is associated with a data structure for storing its bag of
mappings and some indexes for indexing binding values. Each operator implementation needs different
implementations of the AWB. Designing efficient data structures and indexing schemes for the AWB is
discussed in detail in Chapter 6.

Algorithm 1: AWB Access Methods

1. Procedure insert(µ)
2. Iterator probe(condition)
3. Iterator purge(condition)
4. Iterator iterator()
5. int size()

insert(µ) is used to insert a new mapping µ into the buffer and probe(condition) is used to retrieve
the list of mappings that meet a certain criteria. A simple criteria might be a single search key or a
composite search key accompanied with a time-constraint filter. In some special implementations, the
probing criteria could be a complex search condition combined with other conditions on auxiliary
data of each mapping put into the buffer. Examples of such auxiliary data are punctuations and
processing flags [Avnur and Hellerstein, 2000a, Tian and DeWitt, 2003, Tucker et al., 2003, Ding and
Rundensteiner, 2004]. The output list is returned as an iterator—an abstraction which is commonly used
for accessing tuples in relation tables [Graefe, 1993]. The AWB provides the function purge(condition)
to remove the mappings that match a certain criteria similar to the probe function. For instance, this
function is used to evict the expired mappings from the buffer of a sliding window. The purge method
might return the list of mappings removed that might be needed for other processing. The iterator() and
size() methods are used to iterate over the bag of mappings and to get the size of the bag.

5.1.5 Cache Manager and Cache Fetcher

The CQELS query execution framework supports querying RDF datasets that are too large to fit entirely
into main memory. Since RDF datasets are expected to have a much slower update rate compared
to RDF streams, the output of a sub-query over an RDF dataset rarely changes during a series of
updates of RDF streams. The Cache Manager and Cache Fetcher modules exploit this property to avoid
re-computing the sub-queries and to accelerate the repeated operations on the intermediate results of
sub-queries. The Cache Fetcher is responsible for pre-computing and materialising the sub-queries on
the RDF datasets hosted in local triple storages or remote storages associated with SPARQL endpoints.
The pre-computed intermediate results are sent to the Encoder and can then be integrated with other
encoded data in window buffers. The Cache Manager is responsible for maintaining the freshness of
such pre-computed results.

To keep the cache updated, the CQELS framework uses triggers to notify changes in the RDF datasets.
The CQELS engine has a triple storage that allows the engine to load and update RDF datasets as
named graphs. This triple storage provides triggers that will notify the engine to update the respective
cached data. For RDF datasets that are not loaded in main memory, we manually set a timer to trigger
an update. There are two updating policies: Fresh update and incremental update. The fresh update
policy is carried out by re-computing the full sub-query as a background process and replacing the old
cached data by the new results as soon as they are ready. This policy is easy to implement but it might
cause very high latency of updating the caches. The incremental update policy is more complicated
5.2. FRAMEWORK IMPLEMENTATION

but far more efficient. This incremental update policy employs the techniques of materialized view
maintenance [Gupta and Mumick, 1999] and streaming data warehouse [Golab and Özesu, 2010].
Some subsets of the intermediate results might be small enough to fit into main memory. If there is still
free space in main memory, the Cache Manager might store some copies of the intermediate results in
memory with an adaptive caching policy, for example, using the adaptive caching policy with a cost
model for storing and accessing cached data introduced in [Babu et al., 2005a] for caching intermediate
results for join operators.

5.2 Framework Implementation

In this section, an implementation of the CQELS framework is discussed and systematically evaluated.

5.2.1 A CQELS Engine Implementation

In the current version of the CQELS implementation, we use relational tables for realising the abstract
window buffers. B+-tree indexes are used on the binding values if needed. For the window pattern
matching operators, we extend the window sliding operators of CQL [Arasu et al., 2006] for triple
streams and AWB-based window buffers. Following the re-evaluation method (cf. Section 2.1.4), the
relational operators in Definitions 4.6-4.7 are implemented by extending the relational operators of Jena
ARQ. The AWB implementations associated with the relational operators store data in main memory.
In addition, the dictionary is stored on disk using a modified version of the disk-based dictionary
implementation of the Jena TDB code base. We do not purge the unused RDF nodes but we cache the
frequently used nodes in the main memory. We also reuse the physical storage and indexing structures
of Jena TDB for storing cached data in the Cache Manager. The Cache Fetcher is also implemented
by using the Jena query processor backed with Jena TDB as the local triple storage. An LRU caching
policy is applied for AWBs that can be put into main memory.

We use a similar approach to Eddies [Avnur and Hellerstein, 2000a] to implement the Dynamic
Executor. This approach enables the Dynamic Executor to route the incoming data as routing entries
to query operators. The routing policy at each clock tick dictates the routing paths. Intuitively, a routing
policy contains the data flows of a query execution plan. The abstract routing algorithm is given in
Algorithm 2. To avoid concurrent operations on window buffers, we use serial execution for scheduling
the routing entries.

Algorithm 2: route(routingEntry, O, t)

| Input: routingEntry : timestamped triple/mapping, O : operator, t : timestamp |
| Ω := compute(routingEntry, O, t) |
| if O is not root then |
| nextOp := findNextOp(O, t) |
| for μ ∈ Ω do route(μ, nextOp, t) |
| else deliver Ω |

The function route(routingEntry, O, t) recursively applies the operator O on a mapping or time-
stamped triple routingEntry and routes the output mappings to the next operator. It uses the following
primitives:

3http://jena.apache.org/
• **compute**(routingEntry, O, t): applies O, a window, relational, or streaming operators, to routingEntry, a timestamped triple or a mapping, at timestamp t, and returns a discrete result set.

• **findNextOp**(O, t): find the next operator to route the output mapping to, at timestamp t, based on a given routing policy.

The routing policy decides the order in which the operators are executed to achieve a certain optimisation purpose, for instance, minimising the processing time. There are many ways to implement a routing policy. One example, common to DBMSs, is a cost-based strategy: The routing policy computes an estimated “cost” to each possible data flow, and chooses the one with the smallest cost. While the definition of cost is not fixed, it is usually measured by estimating the number of output mappings an operator will produce.

To allow an adaptive query execution, we implement a “cardinality-based” routing policy, based on a set of heuristics. For a given query, the engine keeps all possible left-deep data flows that start with a window operator. For instance, Figure 5.2 shows the four data flows that are maintained for the query in the localisation scenario from Chapter 4.

![Figure 5.2: Left-deep data flows for the query in the localisation scenario.](image)

Algorithm 3 shows the findNextOp function used in the current routing policy called in Algorithm 2. Algorithm 3 applies two simple heuristics: The first one, also common in DBMS, pushes operators like filters closer to the data sources. The rationale here is that the earlier we prune the triples that will not make to the final output the better, since operators will then process less triples. The second looks at the cardinality of the operator’s output and sorts them in increasing order of this value, which also helps reducing the number of intermediate mappings to process.

The function **nextUnaryOp**(O) returns the set of possible next unary operators that O can route data to, while **nextBinaryOpOnLeftDeepTree**(O) returns the binary ones. Examples of unary operators are filters and projections, and they can be directly executed on the output produced by O. Binary operators, such as joins and unions, have two inputs, called left and right child, due to the tree shape of the data flows. O will be the left child, since the data flows are all left-deep. The right child is given by the **rightChildOp** attribute. For each binary operator we obtain the cardinality of the right child at time t from **card(binaryOp.rightChildOp, t)**. We then route the output of O to the one whose cardinality function returns the smaller amount.
5.2. FRAMEWORK IMPLEMENTATION

Algorithm 3: findNextOp(Ω, t)

Input: Ω : operator, t : timestamp
1 nextOp := null
2 for unaryOp ∈ nextUnaryOp(Ω) do
3     if unaryOp is a filter operator then return unaryOp else nextOp := unaryOp
4     mincard := +∞
5 for binaryOp ∈ nextBinaryOpOnLeftDeepTree(Ω) do
6     if mincard > card(binaryOp.rightChildOp, t) then
7         mincard := card(binaryOp.rightChildOp, t)
8         nextOp := binaryOp
9 return nextOp

5.2.2 Performance Evaluation

To analyse the performance of our CQELS framework, we compare the CQELS implementation described in the previous section with two other systems that also offer integrated processing of Linked Stream Data and Linked Data: C-SPARQL [Barbieri et al., 2010a], and ETALIS [Anicic et al., 2011]. Note that EP-SPARQL [Anicic et al., 2011] is implemented on top of ETALIS. At first, we considered expressing the queries we want to run in EP-SPARQL, which would be then translated into the language used in ETALIS. However, the translation from EP-SPARQL to ETALIS is currently not mature enough to handle all queries in our setup, so we decided to represent the queries directly into the ETALIS language. We have also considered comparing our system against SPARQL_stream [Calbimonte et al., 2010]. However, SPARQL_stream currently does not support querying on both stream and RDF dataset. Additionally, Streaming SPARQL [Bolles et al., 2008] is not publicly available, therefore it is not included in this experimental comparison and evaluation. Next, we describe our experimental setup, and then report and discuss the results obtained.

Experimental Setup

We use the 5 queries introduced in Example 8. The conference scenario of this example integrates RFID streams generated from RFID tags worn by conference participants and their DBLP profiles. For the stream data, we use the RFID-based tracking data streams provided by the Open Beacon community\(^4\). The data is generated from active RFID tags deployed in real conferences\(^5\), the same hardware was used in the Live Social Semantics deployment [Alani et al., 2009]. The data generator from SP\(^2\)Bench [Schmidt et al., 2009] is used to create simulated DBLP datasets. We have also created a small RDF dataset, 172 triples, to represent the connectivity between the rooms of the conference based on the Open Beacon dataset.

The experiments were executed on a standard workstation with 1 x Quad Core Intel Xeon E5410 2.33 GHz, 8GB memory, 2 x 500GB Enterprise SATA disks, running Ubuntu 11.04/x86_64, Java version 1.6, Java HotSpot(TM) 64-Bit Server VM, and SWI-Prolog 5.10.4. The maximum heap size on JVM instances when running CQELS and C-SPARQL was set to 4GB. For ETALIS the global stack size is also 4GB.

\(^4\)http://www.openbeacon.org/

As shown in [Le-Phuoc et al., 2012a], comparing input/output throughputs of CQELS, C-SPARQL and ETALIS is invalid because they do not output the same output streams. On top of that, Sharaf et al. [2008] show that average response time might not be the appropriate metric to compare continuous query performance because it misses the processings that do not produce output. Therefore, we use the average query execution time as a metric to compare the efficiencies of such engines. At each run, after registering the query, we stream a number of triples into the system and every time the query is re-executed we measure its processing time. We then average these values over the number of executions that have been triggered.

The queries tested follow the templates specified in Section 4.2. They were selected in a way that we cover many operators with different levels of complexity, for instance joins, filters and aggregations. One query instance is formed by replacing $Name$ in the template with a particular author’s name from the DBLP dataset. We have performed the following three types of experiments:

Exp.(1) **Single query:** For each of the Q1, Q3, Q4 and Q5 templates we generate 10 different query instances. For query template Q2, since it has no constants, we create one instance only. Then we run each instance at a time and compute the average query execution time.

Exp.(2) **Varying size of the DBLP dataset:** We perform the same experiment as in Exp. (1) but varying the numbers of triples of the DBLP dataset, between $10^4$ and $10^7$ triples. We do not include Q2 in this experiment, since it does not involve the DBLP dataset.

Exp.(3) **Multiple queries:** For query templates Q1, Q3 and Q4, we register $2^M$ query instances at the same time, with $0 \leq M \leq 10$, and execute them in parallel.

In the experiments Exp.(1) and Exp.(3), the number of triples from the DBLP dataset is fixed to $10^5$.

### Results

Table 5.1 shows the results for Exp.(1). We can see that, for most cases, CQELS outperforms the other approaches by orders of magnitude. The only exception is query Q1, where ETALIS is considerably faster. The reason is that ETALIS supports three consumption policies, namely recent, chronological, and unrestricted, where recent is very efficient for queries containing only simple filters on the stream data. For more complex queries, the performance of ETALIS drops significantly. C-SPARQL is currently not designed to handle large dataset, which explains its poor performance in our setup. CQELS, on the other hand, is able to constantly deliver good performance, due to its combination of pre-processing and adaptive routing policy. For query Q4, CQELS is over 800 times faster than ETALIS and 700 times faster than C-SPARQL.

The results from Exp.(2) are shown in Figure 5.3, for queries template Q1, Q3 and Q5. We can see how the performance is affected when the size of the RDF dataset increases. For both ETALIS and C-SPARQL, not only the average execution time increases with the size of the RDF dataset, but they are only able to run up to a certain number of triples. They could execute queries with a RDF dataset of 1 million triples, but at 2 million ETALIS crashes and C-SPARQL does not respond. CQELS’s

<table>
<thead>
<tr>
<th></th>
<th>$Q_1$</th>
<th>$Q_2$</th>
<th>$Q_3$</th>
<th>$Q_4$</th>
<th>$Q_5$</th>
</tr>
</thead>
<tbody>
<tr>
<td>CQELS</td>
<td>0.47</td>
<td>3.90</td>
<td>0.51</td>
<td>0.53</td>
<td>21.83</td>
</tr>
<tr>
<td>C-SPARQL</td>
<td>332.46</td>
<td>99.84</td>
<td>331.68</td>
<td>395.18</td>
<td>322.64</td>
</tr>
<tr>
<td>ETALIS</td>
<td>0.06</td>
<td>27.47</td>
<td>79.95</td>
<td>469.23</td>
<td>160.83</td>
</tr>
</tbody>
</table>

**Table 5.1:** Average query execution time for single queries (in milliseconds).
performance is only marginally affected by the RDF dataset’s size, even for values as high as 10 million triples, and the performance gains sometimes were of three orders of magnitude. This is mainly due to the cache and indexes used for storing and accessing pre-computed intermediate results. We have observed that the size of the cache, which stores the co-authors and editors of a certain author, does not increase linearly with the size of the dataset. Moreover, by using indexes on this cache, the access time of a mapping increases only logarithmically with the cache size. This behaviour shows the importance of having proper cache and index structures for efficient query processing.

As a scalability test, we analysed how the systems perform under a number of queries running in parallel. Figure 5.4 presents the results for Exp.(3). Again, ETALIS delivers the best performance when there is no join operator on the stream data (Q1). But for the other cases the number of queries it can handle in parallel is very limited (less than 10). CQELS seems to be faster than other systems, however, it scales only slightly better than the other systems do. If we look closely the average execution time increases linearly with respect to number of queries. Therefore, CQELS is faster because its single queries are faster than those of other systems. As all the parallel queries have similar patterns, they share most of the query patterns and windows. We expect the system to scale better with multiple query optimisation techniques such as shared windows [Krishnamurthy et al., 2006], and multiple filters [Babu et al., 2004a].
5.3 Discussion

The performance study on the preliminary CQELS engine implementation shows that the encoding and caching components, as well as adaptive optimisation can achieve better performance than other black box systems by order of magnitude. However, it also shows the scalability issues of the system with respect to multiple concurrent queries.

In addition, this implementation uses unmodified relation tables for physical storages and a re-evaluation approach which we believe there are better alternatives to improve the performance of CQELS. Therefore, in Chapter 6, we propose data structures and incremental execution strategies for operator implementations. For performance tuning and query optimisation, we will propose solutions for the above scalability issues and optimisation algorithms to achieve much better performance for advanced implementations of CQELS in the Chapter 7.
Chapter 6

Data Structures and Algorithms for Incremental Evaluation of Triple-based Windowing Operators

As the CQELS framework uses a whitebox architecture, it is possible to improve the CQELS engine performance by improving the performance of its components. This chapter aims at modifying the access methods and physical operators in the Dynamic Executor to achieve better performance than the one implemented in Chapter 5. First, we analyse which parts of the Dynamic Executor should be modified in Section 6.1, i.e., data structures and incremental evaluation algorithms. Next, we propose novel efficient data structures as alternatives to the relation-based ones used in traditional relational DSMSs. Based on these data structures, we also address the shortcomings of the incremental evaluation approaches discussed in Section 2.1.4. and introduce efficient algorithms for incremental evaluation of triple-based windowing operators.

6.1 Introduction

CQELS continuous query operators are defined as operators on bags of mappings in Section 4.1, i.e., join, union, minus, filter, aggregate. Each of these operators consumes a set of bags of mappings and returns a bag of mappings which then can be used as intermediate mappings to be consumed in another operator of an execution pipeline. In each pipeline, the inputs at the bottom are bags of mappings stored in the window buffers of sliding window operators and the top returns a bag of mapping to the output stream. Execution pipelines are continually applied to the input streams. Thus, processing state such as window buffers and operator inputs constantly changes through inserting and evicting operations. Therefore, data structures and physical storages for mappings and bags of mappings have a significant impact on the performance of continuous query operators.

The CQELS implementation from Section 5.2.1 however uses relations to store mappings. In the CQELS framework, the mappings are stored in the encoded form under fixed-size integers, thus, its relation-based storage requires tuple header sizes that dominate the total storage size [Abadi et al., 2007]. Moreover, relation-based data structure designed for wider and short tables might waste significant memory amounts of for maintaining processing state in the Dynamic Executor. This motivated us to
design novel data structures in Section 6.2 to deal with unusually large number of individual RDF-based data points that need to be processed in the Dynamic Executor of the CQELS framework. Besides this, as discussed in Section 2.2.2, the B+Tree-based indexing strategy used in the implementation in Section 5.2.1 is not the best choice for in-memory and write-intensive storage. Therefore, a new indexing strategy is also introduced in Section 6.2 to improve the performance of the CQELS engine. Additionally, the re-evaluation method used in the CQELS implementation from Section 5.2.1 can be improved by adopting the incremental evaluation method (c.f. Section 2.1.4). In principle, the incremental evaluation method enables the CQELS engine to avoid unnecessary re-evaluation by incremental computation. However, in practice, the techniques for handling the expirations of incremental evaluation such as direct timestamp and negative tuple have their own technical issues (c.f. Section 2.1.4). Therefore, to build efficient algorithms for incrementally computing triple-based windowing operators, we address such issues in Section 6.3.2 by using new data structures presented in Section 6.2.

### 6.2 Operator-aware Data Structures

This section presents *query operator-aware data structures* for mappings and bags of mappings that provide efficient access methods for the incremental evaluations of windowing operators. These data structures are called “query operator-aware” because their design is based on how the query operators manipulate the data items, i.e., binding values and mappings. Next, we present the data structures for mappings, bags of mappings, and evaluate their performance in great detail.

#### 6.2.1 Mappings

In contrast to relation-based data structures that store mappings in rows of a table, we introduce a tree-based data structure which contains *leaf-mappings* and *intermediate-mappings*. Leaf-mappings are similar to the row-based data structure of a relational table. The leaf-mappings are generated by pattern matching operators. The intermediate-mappings are mappings generated by relational operators during the execution steps of a query plan. Figure 6.1 illustrates how a mapping is represented as a tree of the constituent mappings. In this figure, we have three leaf-mappings in window \( W_1 \), two leaf-mappings in window \( W_2 \) and one leaf-mapping for \( W_3 \). The query execution of \( (W_1 \Join W_2) \Join W_3 \) generates the intermediate mappings \( \Join_{11}, \Join_{12}, \Join_{13}, \Join_{21} \) and \( \Join_{22} \).

With the tree-based data structure, each intermediate mapping does not need to store binding values. It only needs one or two pointers to reference to the mappings that generated it. Therefore, they require considerably less memory to store intermediate mappings in comparison to row-based storage. For instance, if we use a row-based storage, mappings \( \Join_{11}, \Join_{12}, \Join_{13} \) and \( \Join_{21} \) would need more than 3x64 bits to store each one and also 4x64 bits for mappings \( \Join_{21} \) and \( \Join_{22} \). On top that, each pointer only needs less than 32 bits for a 32-bit CPU and 64 bits for a 64-CPU since the operating systems or JVMs support pointer compression.

Moreover, with this tree structure, we only need to store timestamps on the leaf-nodes to be able to generate timestamps for the final results and to detect the expiration of intermediate results. Previous work usually uses extra data such as timestamps [Ghanem et al., 2007] or punctuations [Ding and Rundensteiner, 2004] to enable such functionalities. These works assume that storage space for timestamps is much smaller that the space of each tuple. However, in our processing context, a leaf-mapping only needs space for one or two integer binding values, i.e., 64 bits for each binding value,
which means that each timestamp would need the same storage size. Hence, avoid using timestamps for intermediate mappings significantly reduces the memory consumption for the query execution process. Algorithm 4 shows how to check if a mapping has expired. The algorithm traverses to its constituent leaf-mappings to check if at least one of them has expired.

**Algorithm 4: isExpired(μ, t)**

- **Input**: μ : mapping, t : timestamp
- 1. if μ is a leaf-mapping then
- 2. if leaf-mapping μ is expired at clock tick t then
- 3. return True
- 4. else
- 5. return False
- 6. else
- 7. for μ* ∈ childrenOf(μ) do
- 8. if isExpired(μ*, t) then
- 9. return True
- 10. return False

Checking expiration for leaf-mappings at a clock tick is done by the window implementation. For instance, for sliding windows, the expiration of a leaf-mapping can be detected by its timestamp. Figure 6.2 shows an example of how the expiration of some mappings from Figure 6.1 is detected. In this example, if the mapping ⟨a₁, b₁⟩ of window W₁ has expired, the expirations of the mappings ⊿²₁ and ⊿₁₁ are detected by traversing the doubled-tip arrows in Figure 6.2.

While saving memory space, this data structure needs more steps to reach the leafs to retrieve the values of the bindings or to compute timestamps for intermediate or output mappings. However, this does not have a critical impact in the processing due to several reasons. The first reason is that retrieving
binding values takes much less time in comparison to probing operations that dominate the processing time of the query processor. Another reason is due to the characteristics of modern hardware in which the time spent for traversing to the leafs is much smaller than the time spent for retrieving/copying binding values in the main memory. In particular, a single core of modern CPU can execute up to four instructions per cycle [Owens, 2011] whilst the latency of accessing data in RAM can be over 200 processor cycles [Boncz et al., 2005]. On top of that, a query usually has 5-20 patterns on average. Therefore, the tree of a mapping has only a few nodes to traverse for reaching the binding values.

6.2.2 Bags of Mappings

An operator consumes or outputs mappings in a bag stored in a buffer. In this section, we describe the design of physical storages for storing bags of mappings. Similar to the concept of synopsis from STREAM [Arasu et al., 2006], these physical storages are used in window buffers and in intermediate processing states. They implement the AWB Access Methods defined in Section 5.1.4. To support fast probing, the storages need indexing structures that have low maintenance cost for inserting and evicting the data items that are stored in the tree-based data structure. Operations such as inserting, evicting and probing dictate the choice of the data structures for storing bags of mappings and the indexing structures on their subsets of binding values.

A bag of mappings is stored as a list of pointers referencing to mappings stored in the tree-based data structure. The mappings in the list might be linked in chronological order, for example, the newest leaf-mapping in the head and the oldest in the tail for window buffers. The list could be implemented as a linked list or an expandable array. The array-based implementation of the list has more chances of cache hits and it does not need an extra data structure or pointers for linking data elements in the list [Owens, 2011].

When the maximum size of the list is known, a circular array implementation of the list can be used, for instance, a count-based window with a fixed number of items. For the lists whose maximum size is unknown, if the size of list goes over the size of the array, another array has to be allocated. This
introduces an overhead of copying data elements to the new array. Therefore, the linked list is an alternative if the maximum size of the list is unpredictable. In practice, if the size of the list does not change dramatically, the circular array-based implementation of the list is faster than the linked list. In particular, for lists used in window buffers, it only needs to remove items from the tail and to insert into the head. Therefore, the circular array can be used to save pointers to the next items. In other cases, where data items might be removed randomly, the next pointer has its advantage over the circular array.

6.2.3 Indexes on Bags of Mappings

Inspired by the domain index [Bobineau et al., 2000, Golab et al., 2004, Ding and Rundensteiner, 2004], we propose indices on keys constructed from a subset of value bindings of mappings stored in a list data structure. The index provides an interface to look up a key for a lookup condition over a set of value bindings. Each key is associated with an entry that stores the number of items, called counter, in the list with the corresponding key. The key and entry pair are stored in data structures such as B-trees or hash tables. The value of this counter is also used as a statistic metric for adaptive optimisers. For instance, it can be used to dynamically estimate the join selectivity of window buffers. Due to different uses of the indices, the index entries and the list data structures are implemented as a one-way index or a ring index described in the following. The first index is a one-way index as illustrated in Figure 6.3a.

![One-way index](a)

![Ring Index](b)

**Figure 6.3:** Indices for bags of mappings.

This index can only be used to check if there is an item with a certain key in the list. Therefore, it is useful for operators such as duplication elimination, negation and aggregation. The implementation of the index entry can be extended to store aggregate values for each group. For inserting a new mapping into the list, the index needs one lookup to find the entry corresponding to the key of the new mapping, then updates its counter. If the key has not been in the index, then a new entry will be created. The delete operation also needs a lookup to find the corresponding entry in the index to decrease is counter. Such lookups can be avoided by adding a pointer to every mapping in the list for each index. The mappings would have a pointer to point back to its index entry. When a mapping is deleted, we just need to follow the pointer to the entry to decrease its counter. If the number of keys is large and the
deleting rate is high, adding one more pointer to each mapping might improve the throughput, provided that extra memory consumption is not critical. On the other hand, if the number of keys is small and there is a large number of items in the buffer, not having such pointers might be a better solution. Hence, the instantiation of the mappings for the one-way index is dynamically switched at runtime based on the size of the list and the number of keys.

For an operation that needs to retrieve the list of items that match a certain key, we extend the one-way index to a ring index [Bobineau et al., 2000, Golab et al., 2004] as depicted in Figure 6.3b. The ring index links all mappings with the same index key in a ring. The index entry of this key contains the start-of-ring pointer to the first mapping of the ring. This first mapping is the newest mapping of the ring. The method \textit{insert} in Algorithm 5 is used to insert a new mapping to the end of the list with a ring index. If the key of the new mapping $\mu$ is already in the index, the algorithm re-assigns its ring pointer to the new first mapping of the ring (line 4), and increases the counter of the ring index entry (line 5). Otherwise, a new ring index entry is created (line 8). The start-of-ring pointer then points to the inserting mapping $\mu$ (line 10).

\begin{algorithm}
\begin{algorithmic}[1]
\caption{\textit{insert}($\mu, \mathcal{L}$)}
\Statex \textbf{Input:} $\mu$: mapping to be inserted, $\mathcal{L}$: the list with a ring index
\Statex 1 \hspace{1em} $(K) \leftarrow \text{key of } \mu \text{ for the ring index in } \mathcal{L}$;
\Statex 2 \hspace{1em} $\mathcal{E} \leftarrow \mathcal{L}.\text{index.get}((K))$;
\Statex 3 \hspace{1em} \If{$\mathcal{E}$ \text{ is not null}}
\Statex 4 \hspace{2em} $\mu.\text{ring-pointer} = \mathcal{E}.\text{start-of-ring}$;
\Statex 5 \hspace{2em} $\mathcal{E}.\text{increaseCounter}()$;
\Statex 6 \hspace{1em} \EndIf
\Statex 7 \hspace{1em} \Else
\Statex 8 \hspace{2em} $\mathcal{E} \leftarrow \text{create a new ring index entry with counter=1}$;
\Statex 9 \hspace{1em} \EndIf
\Statex 10 \hspace{1em} $\mathcal{E}.\text{start-of-ring} = \mu$;
\Statex 11 \hspace{1em} $\mathcal{L}.\text{insert}($\mu$)$;
\end{algorithmic}
\end{algorithm}

To retrieve a list of mappings that have a particular key, the method \textit{probe} in Algorithm 6 is used. It first gets the ring index entry corresponding to the key. If it is not null, an iterator is created for iterating over all the mappings that have the key by following the ring pointers. Note that the counter in the index entry is used the stop condition for the iterator.

\begin{algorithm}
\begin{algorithmic}[1]
\caption{\textit{probe}(K)}
\Statex \textbf{Input:} key: key to be probed, $\mathcal{L}$: the list with a ring index
\Statex 1 \hspace{1em} $\mathcal{E} \leftarrow \mathcal{L}.\text{index.get}(K)$;
\Statex 2 \hspace{1em} \If{$\mathcal{E}$ \text{ is not null}}
\Statex 3 \hspace{2em} \Return the iterator that iterates over $\mathcal{E}.\text{count}()$ mappings by following the ring pointers from the mapping pointed to by $\mathcal{E}.\text{start-of-ring}$;
\Statex 4 \hspace{1em} \EndIf
\Statex 5 \hspace{1em} \Else
\Statex 6 \hspace{2em} \Return empty iterator;
\Statex 7 \hspace{1em} \EndIf
\end{algorithmic}
\end{algorithm}

When the frequency count of a key in an index reaches zero, this means that the key is not being referred by any mapping anymore. However, deleting such keys in the index might not be efficient. For instance, if a self-balance tree is used for the index, a lazy-deletion approach could avoid a number of
re-balancing operations [Golab et al., 2004]. To enable lazy-deletion, we delay the deletion by still maintaining the keys with zero frequency count. A simple condition to check if a frequency count greater zero must be added in the search function of the index implementation. We maintain a list of keys with zero frequency count, then when the size of the list reaches a certain threshold, a batch deletion operation will be carried out. There might be the case that by the time the batch deletion is started, some of the keys in the list have a frequency count greater the zero. By just ignoring these keys after delaying the deleting operation, we can save unnecessary inserting/deleting operations on them.

We can use balanced trees or hash tables for indexing keys for both one-way and ring indices. The hash tables can only be used for the equality predicates whilst the balanced trees also support range scans. The balanced trees have fixed boundaries of search time defined by their heights. For instance, an AVL tree’s height is \(1.44 \log_2 (n + 2) - 1\) and a Red-black tree’s height is \(2\log_2 (n + 1)\), where \(n\) is number of indexed keys. In principle, the hash table can have search time of \(O(n)\) in the worst case. However, the average search time of a hash table is \(n/k + 1\), where \(k\) is the number of buckets. In our adaptive execution strategy, we favor hash tables for equality predicates because the adaptive executor can trade memory for better speed by increasing the number of buckets to have lower search time. For dynamically reconfiguring a hash table, we implemented expandable hash buckets that can expand when the number of keys reach a certain threshold, called the load factor. The load factor is the ratio of the number of keys to the number of addresses allocated for hash buckets.

### 6.2.4 Experimental Results

To evaluate the performance of our indices on different data structures, we experimented with implementations of AVL trees, red-black trees and hash tables. In the first experiment shown in Figure 6.4, we measured the throughputs of the insert operation on different numbers of keys and window sizes for each implementation of the ring index. In another experiment shown in Figure 6.5, we measured the average probing time of these three implementations of the ring index on window buffers with 1 million mappings with different numbers of distinct keys.

In these experiments, we set the load factor at 75% which is default load factor most of hash table implementations. The experiments ran on a standard workstation configuration, AMD64 2x E5606 Intel Quad-Core Xeon 2.13GHz with 16GB RAM, running OpenJDK Runtime Environment (IcedTea6 1.8.10) OpenJDK 64-Bit Server VM on Debian squeeze1. Figure 6.4 shows that hash table is the winner over AVL trees and red-black trees for the inserting operation. With the throughputs ranging from 400,000 to 800,000 mappings/second for the window size up to 10 million mappings, the ring index meets our requirement of a low maintenance and high throughput indexing mechanism. Figure 6.5 shows that the hash table also delivers faster probing time for the ring index implementation. The average probing time on a 1-million-mapping window is consistently around 600-800 nano seconds. Intuitively, a nano second is the cycle time of a 1GHz processor. This fast and stable search time is the basis for the high throughput operators implemented in the following sections.

Next, we conducted experiments to analyse the behaviour of the inserting and probing operations on the window buffers which use a hash table for the ring index. We varied the number of distinct keys in the windows as well as the window size then recorded the throughput of inserting and probing operations. The number of keys and window sizes range from 10,000 to 1 million. Figure 6.6 and Figure 6.7 show the throughput of inserting and probing operations. Both figures use contour maps to represent the clusters of throughputs. They show that the throughputs are gradually clustered into groups that have the window sizes in a certain range of number of keys. The convergence of throughputs is clearer when

---

1This configuration will be used for the rest of the thesis.
the ratio of the number of keys to the window size is smaller. This might be used by an algorithm to estimate the throughputs of operations by monitoring the number of distinct keys in the windows. The throughput estimation is used in rate-based optimisation algorithms [Viglas and Naughton, 2002] to maximise the output rate in a fluctuating stream rate setting.

Figure 6.4: Insert throughput with different number of keys.
6.2. OPERATOR-AWARE DATA STRUCTURES

Figure 6.5: Probing time of an AVL/Red-black Tree/hash index on a 1 million mapping window.

Figure 6.6: Insert throughput on windows with hash index.
6.3 Incremental Evaluation for Sliding Windows

This section presents an incremental evaluation approach for the operators in the CQELS language. We extend incremental equations [Griffin and Libkin, 1995, Ghanem et al., 2007] to represent incremental semantics for the operators. The incremental evaluation of each operator is based on two equations for handling the events of new mapping arrival and mapping expiration. For a new mapping \( \mu \) arriving to the input buffer \( R \), the operator \( \cup \) is used to indicate this event and the evaluation is denoted as \( R \cup \mu \). The evaluation for the event of expiring mapping \( \mu \) in the input buffer \( R \) is represented as \( R - \mu \).

We first address the stateless operators (i.e., select, project, union), then we investigate how to handle the expiration in the stateful operators (i.e., join, aggregate, distinct, minus) based on our tree-based data structure. Then, we present algorithms for incremental evaluation of stateful operators.

6.3.1 Stateless Operators

Stateless operators do not have to maintain a processing state for incremental evaluation, i.e., they do not need to access any previous input. Incremental equations 6.1, 6.2 and 6.3 show how a new mapping and an expired mapping are processed by the Select(\( \sigma \)), Project(\( \pi \)) and Union(\( \cup \)) operators respectively. Incremental evaluations of nonstateful operators are straightforward, since a new and expired mapping can be processed directly to produce the corresponding new and expired mappings.

Figure 6.7: Probing throughput on windows with hash index.
6.3. INCREMENTAL EVALUATION FOR SLIDING WINDOWS

\[ \sigma_p(R \uplus \mu) = \sigma_p(R) \uplus \sigma_p(\mu); \quad \sigma_p(R \setminus \mu) = \sigma_p(R) - \sigma_p(\mu) \]  \hspace{1cm} (6.1)

\[ \pi_A(R \uplus \mu) = \pi_A(R) \uplus \pi_A(\mu); \quad \pi_A(R \setminus \mu) = \pi_A(R) - \pi_A(\mu) \]  \hspace{1cm} (6.2)

\[ (R^1 \uplus \mu) \cup R^2 = (R^1 \cup R^2) \cup \mu; \quad (R^1 \setminus \mu) \cup R^2 = (R^1 \cup R^2) - \mu \]  \hspace{1cm} (6.3)

6.3.2 Handling the Expiration in Stateful Operators with a Tree-based Data Structure

As presented in Section 2.1.4, incremental evaluation of stateful operators needs to handle the expirations of old stream elements [Golab, 2006]. An expired mapping may cause the removal of one or more items from the result (e.g., aggregation) or the addition of new mappings to the result (e.g., duplicate elimination and minus). There should be mechanisms to signal expiration events. There are two main techniques to signal the expirations: negative tuple [Arasu et al., 2006, Golab, 2006, Golab and Özsu, 2010] and direct timestamp [Arasu et al., 2006, Golab, 2006].

We revisit the two approaches for invalidating expired tuples from Section 2.1.4, by considering the example shown in Figure 6.8. This example represents the query \((W_1 \triangleright W_2 \triangleright W_3)\), where \(W_1\) is a count-based window, and \(W_2\) and \(W_3\) are time-sliding windows. The figure shows how the data enters the windows, how the results are generated, and how the expired mappings should be invalidated.

For the direct-timestamp approach, when each mapping arrives the window at a timestamp \(t\), it will be assigned an expiration timestamp, \(exp\), where \(exp = t + \text{windowlength}\), and \(\text{windowlength}\) is the length of the sliding window. For instance, \(\langle b_1, c_1 \rangle .exp = 1 + 5 = 6\) and \(\langle c_1, d_1 \rangle .exp = 3 + 5 = 8\). When we join these two mapping, we have the result mapping \(\langle b_1, c_1, d_1 \rangle\) with an expiration timestamp equal to \(\min(\langle b_1, c_1 \rangle .exp, \langle c_1, d_1 \rangle .exp) = \min(6, 8) = 6\). To invalidate the expired mappings at each
time point, the operators simply checks the expiration timestamps of the mappings in their processing state.

However, this approach is not applicable to count-based windows like $W_1$. For instance, when the mapping $\langle a_1, b_1 \rangle$ arrives at $W_1$, its expiration time is not known until time point 6, where the mapping $\langle a_3, b_1 \rangle$ arrives. In this case, the negative tuple approach can be used to signal the expirations. For instance, at time point 6, a negative mapping $\langle a_1, b_1, c_1, d_1 \rangle$ is sent as a signal for the invalidation. Therefore, the invalidation has to re-compute the join again with the negative mapping to find the expired mapping $\langle a_1, b_1, c_1, d_1 \rangle$ generated in the final results. Ghanem et al. [2007] proposed a solution to reduce the overhead of re-processing the negative mappings because every mapping eventually expires from its window and generates the corresponding negative tuples. However, this requires extra memory since it adds expiration timestamps and piggy-back flags to intermediate results. Moreover, this approach can only be applied for time sliding windows. A hybrid approach is also proposed in [Krämer and Seeger, 2005], but, the processing and memory overhead still persist.

To address these issues, we propose a novel approach based on our tree-based data structure and its simple expiration checking algorithm (Section 6.2.1). For leaf-mappings with a timestamp, the expiration can be triggered by source inputs or clock ticks. To signal an invalidation to the following operators, the window containing the expired leaf-mapping sends a negative leaf-mapping to the final operator in the query pipeline to trigger the invalidation. The negative leaf-mapping of a leaf-mapping is an expired version of it. It is created by simply changing the timestamp to its negative value. For instance, at time point 6, the third mapping $\langle a_3, b_1 \rangle$ arrives at the window, and thus, the first mapping $\langle a_1, b_1 \rangle$ expires. The negative leaf-mapping $\langle a_1, b_1 \rangle$ is sent to the final operator to check whether $\bowtie_{21}$ and $\bowtie_{22}$ have also expired. In this case, the tree of $\bowtie_{21}$ contains $\langle a_1, b_1 \rangle$, therefore, $\bowtie_{22}$ has also expired.

**Algorithm 7:** Invalidate expired mappings

```plaintext
Input: $\mu^-$: negative leaf-mapping, $R$: an input buffer, $\mathcal{LM}$: a buffer for leaf mappings that generated $R$

1. invalidated ← 0
2. toBeInvalidated ← $\mathcal{LM}$.get($\mu^-$)
3. $\mathcal{EXP} ← \emptyset$
4. for $\mu \in R$ // iterate over $R$ in the inserting order
5. do
6.   if $\mu$ is generated from $\mu^-$ then
7.     $\mathcal{EXP}$.insert($\mu$)
8.     Remove $\mu$ from $R$
9.     invalidated + +
10.   if expired==toBeExpired then
11.     Remove $\mu^-$ from $\mathcal{LM}$
12.     return $\mathcal{EXP}$;
```

In the input buffers of an intermediate operator, the expired mappings trigged by a negative leaf-mapping might not be stored in a consecutive order. For instance, a join operator can generate output mappings in a random order, which are stored in the input buffer of the next operator, e.g., aggregates. In the worst case, the invalidation has to go over all mappings in the buffer to check if any of them has expired. To avoid unnecessary checks, we use an auxiliary buffer with an one-way index to remember how many mappings generated from a leaf-mapping were inserted into the buffer. The index counter is
then used to stop the check operation earlier as shown in Algorithm 7.

### 6.3.3 Multiway Join

A multiway join query can be evaluated by trees of binary, partially blocking, and pipelined join operators. However, this approach is not sufficient for processing streaming inputs that need to dynamically reorganise the evaluation tree in response to the changes of stream data. Therefore, similar to MJoin [Viglas et al., 2003], we introduce a single multiway join that works over more than two input buffers. This multiway join generates and propagates results in a single step without having to pass these results through a multi-stage binary execution pipeline.

Because the multiway join is symmetric, without loss of generality, we extend the incremental equations of the binary join to the n-way join as shown in Equations 6.4 and 6.5 where the inserting and expiring happen in the input buffer $R^1$. In order to employ our ring-index for window buffers, we represent the incremental evaluation of these equations in Equation 6.6 using the select operator $\sigma_{\mu_1}(R^2)$. The operator $\sigma_{\mu_1}(R^2)$ returns all the mappings stored in the input buffer $R^2$ which are compatible with $\mu_1$. As shown in the experimental result of Section 6.2.4, this operator is supported by the high throughput probe method on the data structure that has ring-indexes on the variables to check the compatibility.

$$
(R^1 \cup \mu_1) \bowtie (R^2 \ldots \bowtie R^n) = (R^1 \bowtie R^2 \ldots \bowtie R^n) \cup (\mu_1 \bowtie R^2 \ldots \bowtie R^n) 
$$

Equation 6.4

$$
(R^1 - \mu_1) \bowtie (R^2 \ldots \bowtie R^n) = (R^1 \bowtie R^2 \ldots \bowtie R^n) \setminus (\mu_1 \bowtie R^2 \ldots \bowtie R^n) 
$$

Equation 6.5

$$
\mu_1 \bowtie (R^2 \ldots \bowtie R^n) = (\{\mu_1\} \times \sigma_{\mu_1}(R^2)) \bowtie (R^3 \ldots \bowtie R^n) 
$$

Equation 6.6

The evaluation of a new mapping $\mu_1$ inserted into the input buffer $R^1$ is illustrated in Figure 6.9.

![Figure 6.9: Multiway join process.](image)
When a mapping $\mu_1$ is inserted into the input buffer $R^1$, it will be used to probe one of the other input buffers $R^2 \cdots R^n$. Let us assume that $R^2$ is the next input buffer in the probing sequence. For each mapping $\mu_2$ in $R^2$ that is compatible with $\mu_1$, an intermediate joined mapping in the form $\mu_1 \circ \mu_2$ is generated. Subsequently, $\mu_1 \circ \mu_2$ is recursively used to probe the other input buffers to generate the final mappings. When a buffer that does not return any compatible mapping is found, the probing sequence stops. To avoid the double computation issue of the negative tuple approach in the event of expiration, the outputs of a multiway join are stored in a buffer that is facilitated through invalidate method of Algorithm 7. When the expired mapping arrives, it is used as a parameter to call the invalidate method to find expired outputs. This invalidating operation is usually done by the upper operator of the query pipeline, which consumes the multiway join output as its input buffer.

Algorithm 8 shows our incremental evaluation algorithm for a multiway join with $n$ input buffers. Lines 2-4 handle new mappings and line 6 is for forwarding the negative mappings to the upper operator. Line 4 calls the recursive sub-routine probingPropagate given in Algorithm 9 to initialise the probing sequence. The probing sequence is given in sub-routine findNextProbWin in line 2. Each step of the probing sequence is dynamically identified by the output generated from previous steps. Therefore, the probing sequence plays an important role in the performance of this multiway join algorithm. We will investigate how to optimise the probing sequence in Chapter 7. In this chapter, we only use a simple routine for determining the probing sequence as shown in Algorithm 10. This algorithm provides a fixed probing sequence for processing a new mapping from each input buffer.

---

**Algorithm 8: Multi-Way Join**

**Input:** $n$ input buffers $W_1, \ldots, W_n$

1. if a new mapping $\mu$ arrives at window $W[i]$ then
2. remove expired tuples from all windows
3. $W[i].insert(\mu)$
4. probingPropagate($\mu$, $\{W[1], \ldots, W[n]\} \setminus \{W[i]\}$)
5. else
6. propagate negative mapping to upper operator

---

**Algorithm 9: Probing propagation probingPropagate**

**Input:** $\mu$, $k$ sliding windows $\{W[i_1], \ldots, W[i_k]\}$

1. if $k == 0$ then
2. $i_{next} \leftarrow \text{findNextProbWin}(\mu, \{W[i_1], \ldots, W[i_k]\})$
3. for $\mu^* \in W[i_{next}].probe(\mu)$ do
4. probingPropagate($\mu \circ \mu^*$, $\{W_1, \ldots, W_k\} \setminus \{W[i_{next}]\}$)
5. else
6. dispatch $\mu$

---

**Algorithm 10: Provide fix probing sequence for findNextProbWin**

**Input:** $\mu$: mapping to probe, $k$ sliding windows $\{W[i_1], \ldots, W[i_k]\}$

1. for $j \in [1..k]$ do
2. if $\mu$ shares joining variable with window $W[i_j]$ then
3. return $i_j$;
We evaluate the performance in terms of execution throughput\(^2\) of our multiway join algorithm based on the tree-based data structures against the relation-based algorithms described in Chapter 5. For evaluation purposes, we also compare it to the performance of the join algorithms in ESPER \(^3\), using ad-hoc data structures. These ad-hoc data structures are manually programmed Java classes with attributes in primitive data types, i.e., integer. Each attribute is used to store an encoded version of a binding value of a mapping. Therefore, it is more compact than our tree-based data structure and faster to access its binding value. In the first experiment, we tested a binary join and a 3-way join on window buffers with the same window size. The binary join has 10 distinct values in the join predicate and the 3-way join has 10,000 distinct values in the join predicates. The results are reported in Figure 6.10a and 6.10b.

The results for different window sizes show that our algorithm consistently has the best throughputs

\(^2\)The execution throughput is measured based on the number of mappings that trigger executions of an operator per second. From this point forward, throughput is referred as the execution throughput of an operator.

\(^3\)http://esper.codehaus.org/

![Figure 6.10: Join throughput of binary joins.](image-url)
CHAPTER 6. DATA STRUCTURES AND ALGORITHMS FOR INCREMENTAL EVALUATION
OF TRIPLE-BASED WINDOWING OPERATORS

in comparison to the relation-based and ad-hoc based algorithms. The new data structures and
algorithm improve the join performance by orders of magnitude, in comparison with the relation-
based implementation. However, the slopes of throughput curves in Figure 6.10a and 6.10b differs
considerably in relation to the ratio of the number of distinct values to the window size.

Subsequently, we conducted two experiments on the 3-way join to find out what drives such differences.
In the first experiment, we fixed the ratio of number of distinct values in the joining predicate (1/10)
to the window size and selectivity (1%) of the join predicate. Figure 6.10c shows that there are only
marginal changes for window sizes from 1000 to 1 million mappings. In the second experiment, we
fixed the number of distinct values in the joining predicate and varied the window sizes. Figure 6.10d
shows that the throughputs of all algorithms drop dramatically for window sizes greater than 1000.
Hence, the ratio of the number of distinct values to the window size determines the throughput of
join operators. The results of these two experiments show that the throughput of join operator can be
estimated based on the number of distinct values and the window size. Note that the number of distinct
values in the join predicate is stored in the one-way and ring indices.

6.3.4 Aggregate

An aggregate operator $AGG_{f_1(A_1), f_2(A_2), ..., f_k(A_k)}$ maps each each input mapping to a group $G$ and pro-
duces one output mapping for each non-empty group $G$. The output has the form $<G, Val_1, ..., Val_k>$,
where $G$ is the group identifier and $Val_i$ is the group’s aggregate value of the function $f_i(A_i)$. The
value $Val_i$ is updated whenever the set of mappings in $G$ changes in the case of new and expired
mappings. Both new mappings and expired mappings can result in an update to the value of a group
and the aggregate operator needs to report the new value for that group.

Algorithm 11 shows the incremental evaluation of the aggregate operator. The operator consumes the
input mappings gradually, and multiple input mappings can contribute to the aggregate value of a group
in a clock tick. For example, the aggregate operator consuming input mappings from a multiway join
might have to update the aggregated value of a group several times before delivering the final update to
the upper operator. Therefore, to signal that all input mappings from the lower operator at a particular
clock tick have already been delivered, we use a special mapping, called null mapping. When an upper
operator receives the null mapping, it can deliver its evaluation for that clock tick. In this algorithm, we
use a buffer to store all updates of aggregate values before they are dispatched to the output. Lines 1-3
handle the dispatching with the null mapping. Lines 5-16 handle the event of a new mapping. The
algorithm uses a one-way index for the input buffer with an extension to store aggregate values. The
composite key is used as the identification of a group. Lines 6-11 find a group to update its aggregate
values. The update is done in lines 13-14. Lines 15 and 16 store the updates and adding the new input
mapping to the input buffer, respectively.

As described in Section 6.3.2, the expired mappings are signaled by negative leaf mappings which are
then used as parameter to invalidate the expired mappings as shown in line 18. Similar to updating
aggregate values for new mappings, the updates for expired mappings are done in lines 19-27.
Algorithm 11: Aggregation

**Input:** \( \mu \): dispatched mapping, \( AGG_{f_1(A_1), f_2(A_2), \ldots, f_k(A_k)} \): aggregation functions, \( O \): output buffer to be dispatched, \( \mathcal{E}\mathcal{X}\mathcal{P} \): buffer of input mappings waiting to be invalidated

1. if \( \mu \) is null mapping then
2. dispatch output buffer \( O \)
3. \( O \leftarrow \emptyset \)
4. else
5. if \( \mu \) is a new mapping then
6. \( gKey \) ← generate the composite key on group bindings of \( \mu \)
7. \( E \leftarrow \mathcal{E}\mathcal{X}\mathcal{P}.\text{index}.\text{get}(gKey) \)
8. if \( E \) is not null then
9. \( E.\text{increaseCount}() \)
10. else
11. \( E \leftarrow \text{create new one-way index entry for a new group} \)
12. for \( i \in [1..k] \) do
13. \( A \leftarrow \text{Recompute } f_i(A_i) \text{ for adding new mapping } \mu \)
14. \( O.\text{insert}(E) \)
15. \( \mathcal{E}\mathcal{X}\mathcal{P}.\text{insert}(\mu) \)
16. else
17. for \( \mu^- \in \mathcal{E}\mathcal{X}\mathcal{P}.\text{invalidate}(\mu) \) do
18. \( gKey \) ← generate the composite key on group bindings of \( \mu^- \)
19. \( E \leftarrow \mathcal{E}\mathcal{X}\mathcal{P}.\text{index}.\text{get}(gKey) \)
20. if \( E.\text{count}() > 0 \) then
21. for \( i \in [1..k] \) do
22. \( A \leftarrow \text{Recompute } f_i(A_i) \text{ for removing expired mapping } \mu^- \)
23. \( O.\text{insert}(E) \)
24. else
25. \( \mathcal{E}\mathcal{X}\mathcal{P}.\text{index}.\text{remove}(gKey) \)

We have also evaluated our algorithm against the relation-based and ad-hoc implementations discussed in the multiway join experiments. We evaluated on four aggregate functions, namely COUNT, SUM, AVERAGE and MIN. In the first experiment, the aggregate operators consume input mappings from a window with one million mappings. Figure 6.11 shows the throughput for different numbers of aggregate groups. The results show that our algorithm performs better than the relation-based algorithm by orders of magnitude. However, the ad-hoc algorithm has the best performance in these experiments. The compact data structure and primitive data type of binding attributes enables faster access to aggregate values. Therefore, it can deliver a better performance. Nevertheless, our algorithm delivers marginally lower throughputs than the ad-hoc algorithm.

![Throughput graphs for different aggregate functions](image)

**Figure 6.11:** Aggregate throughput on a window with one million mappings.

Next, we evaluated our algorithm in the case where the aggregate operator consumes the input mappings from a join operator. We tested it with a binary join which joins two windows with one million mappings each. Figure 6.12 reports the throughputs according to the selectivity of the join. When the selectivity is lower than one, our algorithm delivers the best throughput while the ad-hoc algorithm outperforms our aggregate algorithm when the selectivity is greater than one. This effect is caused by the fact that our data structures need to traverse inside the tree to check the expirations. Moreover, a greater number of input mappings due to greater selectivity requires more scan operations for finding the expired mappings in the input buffer of the aggregate operator.
6.3. INCREMENTAL EVALUATION FOR SLIDING WINDOWS

6.3.5 DISTINCT

The DISTINCT operator eliminates duplicates in the input buffer to return the distinct mappings to the output. The incremental Equations 6.7 and 6.8 describe how to incrementally evaluate the DISTINCT operator in the events of inserting new mapping and removing expired mappings. Equation 6.7 states that a new mapping \( \mu \) only produces a new output if there is no duplicates of \( \mu \) in the operator’s input buffer. The opposite applies to expired mappings: the operator only outputs the expired mapping if \( \mu \) is already in the input buffer.

\[
\epsilon(R \uplus \mu) = \begin{cases} 
\epsilon(R) & \text{if } \mu \in R \\
\epsilon(R) \cup \{\mu\} & \text{Otherwise} 
\end{cases}
\]  

(6.7)

\[
\epsilon(R \uplus \mu) = \begin{cases} 
\epsilon(R) & \text{if } \mu \in R \\
\epsilon(R) \setminus \{\mu\} & \text{Otherwise} 
\end{cases}
\]  

(6.8)

Algorithm 12 shows the DISTINCT operator. As defined in Equation 6.7, handling the arrival of a mapping is similar to the case of an aggregate operator (lines 5-14). Lines 17-22 show how to handle expirations corresponding to Equation 6.8.

Figure 6.12: Aggregate throughput with binary join.
Algorithm 12: DISTINCT

Input: $\mu$: dispatched mapping, $O$: output buffer to be dispatched, $\mathcal{E}X\mathcal{P}$: buffer of input mappings waiting to be invalidated

1. if $\mu$ is null mapping then
   2. dispatch output buffer $O$
   3. $O \leftarrow \emptyset$
4. else
   5. if $\mu$ is a new mapping then
      6. gKey $\leftarrow$ generate the composite key for $\mu$
      7. $\mathcal{E} \leftarrow \mathcal{E}X\mathcal{P}.\text{index.get}(gKey)$
      8. if $\mathcal{E}$ is null then
         9. $\mathcal{E} \leftarrow$ create a new entry
        10. gIndex.insert(gKey,$\mathcal{E}$)
        11. $O.insert(\mu)$
      12. else
         13. $\mathcal{E}.\text{increaseCount}()$
        14. $\mathcal{E}X\mathcal{P}.insert(\mu)$
   15. else
      16. for $\mu^- \in \mathcal{E}X\mathcal{P}.\text{invalidate}(\mu)$ do
         17. gKey $\leftarrow$ generate the composite key for $\mu^-$
         18. $\mathcal{E} \leftarrow \mathcal{E}X\mathcal{P}.\text{index.get}(gKey)$
         19. if $\mathcal{E}.\text{count}() > 1$ then
            20. $\mathcal{E}.\text{decreaseCount}()$
         21. else
            22. $\mathcal{E}X\mathcal{P}.\text{index.remove}(gKey)$

Similar to the aggregate operator, we evaluated our tree-based implementation of the DISTINCT operator against relation-based and ad-hoc based versions. We have tested the performance of our DISTINCT operator in two scenarios: In the first scenario, we place the DISTINCT operator after a triple-based window. In the second scenario, the operator consumes the outputs of a binary join. The results are shown in Figure 6.14. In both cases, the tree-based algorithm delivers the best throughput, while the relation-based and re-evaluation implementation give much poorer performance. This explains the drawback of re-evaluation in the DISTINCT operator, because the majority of such re-computations can be ignored if the fast indices for the processing state are used.
6.3. INCREMENTAL EVALUATION FOR SLIDING WINDOWS

6.3.6 Minus

A minus operator between two input buffers, $R^1$ and $R^2$, produces mappings that are in $R^1$ and are not compatible with any mapping in $R^2$. The minus operator $(R^1 \setminus R^2)$ is asymmetric because handling new or expired mappings depends on whether the mapping is from $R^1$ or $R^2$. Its incremental evaluation is represented as four cases, corresponding to four incremental equations, Equations 6.9-6.12. Algorithm 13 presents our tree-based implementation for the minus operator. This algorithm is also similar to the aggregate operator in handling the input mappings. For a new mapping, line 6 checks if it comes from the left or right side. Depending on the result, the mapping is then evaluated using either Equation 6.9 (lines 7-13) or Equation 6.11 (lines 15-17). Similarly, lines 21-22 and 25-29 implement Equation 6.10 and 6.12 respectively. This algorithm uses one-way indices for the input and the output buffers. The indexing keys are generated as composite keys from shared binding values between the two input buffers. These keys are used to check the existence of compatible mappings from each input buffer.

$$
(R^1 \uplus \mu) \setminus R^2 = \begin{cases} 
(R^1 \setminus R^2) \cup \mu & \text{if } \mu \not\equiv R^2 \\
(R^1 \setminus R^2) & \text{Otherwise} 
\end{cases} \tag{6.9}
$$

$$
(R^1 - \mu) \setminus R^2 = \begin{cases} 
(R^1 \setminus R^2) \cup \mu & \text{if } \mu \not\equiv R^2 \\
(R^1 \setminus R^2) & \text{Otherwise} 
\end{cases} \tag{6.10}
$$

$$
R^1 \setminus (R^2 \uplus \mu) = (R^1 \setminus R^2) \setminus \{ \mu' \mid \mu' \in R^1 \land \mu \not\equiv \mu' \} \tag{6.11}
$$

$$
R^1 \setminus (R^2 - \mu) = (R^1 \setminus R^2) \cup \{ \mu' \mid \mu' \in R^1 \land \mu \not\equiv \mu' \} \tag{6.12}
$$

Figure 6.13: Throughputs of the DISTINCT operator.
Algorithm 13: Minus

**Input:** \( \mu \): dispatched mapping, \( O \): output buffer to be dispatched, \( \mathcal{E}\mathcal{X}\mathcal{P}^L \): left buffer of input mappings waiting to be invalidated, \( \mathcal{E}\mathcal{X}\mathcal{P}^R \): right buffer of input mappings waiting to be invalidated

1. if \( \mu \) is null mapping then
   1.1. dispatch output buffer \( O \)
   1.2. \( O \leftarrow \emptyset \)

2. else
3.   if \( \mu \) is a new mapping then
4.     if \( \mu \) is from the left input then
5.        gKey \leftarrow \text{generate the composite key for } \mu
6.        \( E \leftarrow \mathcal{E}\mathcal{X}\mathcal{P}^R . \text{index.get}(gKey) \)
7.        if \( E \) is null then
8.            \( O . \text{insert}(\mu) \)
9.        else
10.           \( E . \text{increaseCount()} \)
11.           \( \mathcal{E}\mathcal{X}\mathcal{P}^L . \text{insert}(\mu) \)
12.     else
13.        gKey \leftarrow \text{generate the composite key for } \mu
14.        \( O . \text{purgeByKey}(gKey) \)
15.        \( \mathcal{E}\mathcal{X}\mathcal{P}^R . \text{insert}(\mu) \)
16.     else
17.        if \( \mu \) is from the left input then
18.           for \( \mu^- \in \mathcal{E}\mathcal{X}\mathcal{P}^L . \text{invalidate}(\mu) \) do
19.              gKey \leftarrow \text{generate the composite key for } \mu^-
20.              \( O . \text{purgeByKey}(gKey) \)
21.        else
22.           for \( \mu^- \in \mathcal{E}\mathcal{X}\mathcal{P}^R . \text{invalidate}(\mu) \) do
23.              gKey \leftarrow \text{generate the composite key for } \mu^-
24.              \( E \leftarrow \mathcal{E}\mathcal{X}\mathcal{P}^R . \text{index.get}(gKey) \)
25.              if \( E \) is null then
26.                 for \( \mu^+ \in \mathcal{E}\mathcal{X}\mathcal{P}^L . \text{probe}(gKey) \) do
27.                   \( O . \text{insert}(\mu^+) \)

Since the minus algorithm is similar to the binary join, we have used the same setup from the binary join experiment to evaluate it. In the first experiment, we have fixed the ratio of the number of distinct values to the window size at 1/10. Figure 6.14a reports the throughputs of our minus algorithm, the adhoc algorithm and the relation-based algorithm. Our minus algorithm outperforms others by orders of magnitude. In the second experiment, we have fixed the window size to 100,000 mappings for both input buffers. We ran three algorithms with different matching selectivities, and reported the throughputs in Figure 6.14b. The results show that our algorithm for the minus operator produces faster throughputs than the others by orders of magnitude.
6.3. INCREMENTAL EVALUATION FOR SLIDING WINDOWS

Throughputs of Minus operator.

(a) Fixed ratio of the number distinct values to the window size

(b) Fixed window size

Figure 6.14: Throughputs of Minus operator.
CHAPTER 6. DATA STRUCTURES AND ALGORITHMS FOR INCREMENTAL EVALUATION OF TRIPLE-BASED WINDOWING OPERATORS
Adaptive Query Optimisation

The multiway join operator proposed in the previous chapter is the most used and most expensive operator in the query pipeline. This chapter presents optimisation algorithms for multiway joins shared among multiple queries which take advantage of the adaptivity of the CQELS execution framework. We first present two adaptive cost-based optimisation algorithms for single multiway join operators. Based on these optimisation algorithms, we describe the algorithm for optimising multiple join queries that share input streams. Experimental evaluations of all algorithms presented are also presented.

7.1 Adaptive Cost-based Optimisation for Multiway Joins

Data stream sources may suffer fluctuations from data arrival, most of them from unpredictable, slow or bursty network traffic [Urhan et al., 1998]. On top of that, due to the streaming nature of the input, the data distribution is unpredictable, thus, making it difficult to have a model for operator selectivity [Ives et al., 1999]. To deal with changes in the selectivity of the join predicates in multiway joins, we exploit our adaptive multiway join algorithm to find the best execution plans at run-time. In Chapter 6 we have shown that the probing sequence can be dynamically chosen for each new mapping arriving at an input window. Therefore, we try to find the best probing sequence for each new mapping at run-time. The criteria to choose the best probing sequence is based on the optimal cost function which is recursively defined in Definitions 7.1 and 7.2.

\[ C(\mu \bowtie R) = C(\sigma_\mu(R)) + C(\{\mu\} \times \sigma_\mu(R)) \]  
(7.1)

\[ C(\mu_1 \bowtie \{R^2 \cdots R^n\}) = \min_{i_2 \in \{2..n\}} \left( C(\mu_1 \bowtie R^{i_2}) + \sum_{\mu_{i_2}^* \in \sigma_{\mu_1}(R^{i_2})} C((\mu_1 \bowtie \mu_{i_2}^*) \bowtie (\{R^2 \cdots R^n\}\{R^{i_2}\})) \right) \]  
(7.2)

Definition 7.1 is the optimal cost for a binary join which has only one possible probing sequence. When a new mapping \( \mu \) arrives at an input buffer of the binary join, the only option to compute new results is to probe the other input buffer \( R \) for the mappings compatible with \( \mu \). The cost includes \( C(\sigma_\mu(R)) \) as the cost of probing the input buffer and \( C(\{\mu\} \times \sigma_\mu(R)) \) as the cost of creating new mappings from \( \mu \) and the output of \( \sigma_\mu(R) \). Definition 7.2 is used to recursively choose the next input buffer for the
probing sequence \( \mu_1 \bowtie \{ R^2 \cdots R^n \} \) that has the minimal cost, provided that the optimal costs of for all possible next probing sequences \( C(\{ \mu_1 \circ \mu^*_i \} \bowtie \{ \{ R^2 \cdots R^n \} \setminus \{ R^{i_2} \} \}) \) are known. However, to compute the value of this cost function, we need to compute \( C \) for all possible probing sequences, which is prohibitively expensive. Therefore, we aim at finding a near optimal probing sequence by approximating this cost function through two light-weight estimation methods described next.

### 7.1.1 One-step Adaptation

As shown by the experimental results of the throughput of the probing operation in Figure 6.7, we can assume that the cost of probing an input buffer can be estimated by a constant \( \gamma_\sigma \), since it does not change dramatically with respect to the sizes of the window buffers. Moreover, we can also assume that the cost of generating a mapping from two mappings is constant \( \gamma_\times \). Therefore, we have the following definitions for \( \gamma_\sigma \) and \( \gamma_\times \).

\[
\gamma_\sigma \approx C(\sigma_\mu(R)) \quad \text{for every } \mu \text{ and } R \tag{7.3}
\]

\[
\gamma_\times \approx \frac{C(\{ \mu \} \times \sigma_\mu(R))}{|\sigma_\mu(R)|} \quad \text{for every } \mu \text{ and } R \tag{7.4}
\]

To adapt to the changes in the input buffers, our one-step adaption method assumes that the costs of the next steps in the possible probing sequences are approximatively the same for each output generated from the current probing step. Definition 7.5 approximates the cost of each next probing sequence \( \{ R^2 \cdots R^n \} \setminus \{ R^{i_2} \} \) as a constant \( \gamma_{i_2} \):

\[
\gamma_{i_2} \approx C((\mu_1 \circ \mu^*_i) \bowtie \{ \{ R^2 \cdots R^n \} \setminus \{ R^{i_2} \} \}) \tag{7.5}
\]

Hence, we have the approximate cost function for a probing sequence as shown below.

\[
C(\mu_1 \bowtie \{ R^2 \cdots R^n \}) \approx \gamma_\sigma + (\gamma_\times + \gamma_{i_2}) \cdot \min_{i_2 \in \{ 2 \cdots n \}} |\sigma_\mu(R^{i_2})| \tag{7.6}
\]

This approximation enables us to build a simple algorithm for finding the potentially best probing sequence by checking the cardinalities of all possible probing operations of a mapping \( \mu \) on \( k \) sliding windows \( \{ W[i_1] \cdots W[i_k] \} \). The algorithm is described in Algorithm 14. Note that this algorithm can use the counters stored in the ring index entries of the keys on the join predicates of the window buffer. Calling the probe function on a window at line 5 will return the index entries for identifying the cardinality of the probing output.
7.1. ADAPTIVE COST-BASED OPTIMISATION FOR MULTIWAY JOINS

Algorithm 14: findNextProbWin_OneStep

Input: $\mu$: mapping to probe, $k$ sliding windows $\{W[i_1], W[i_k]\}$

1. $\text{minCard} \leftarrow \text{MAXINT}$
2. $\text{minProbeIdx} \leftarrow 0$
3. for $j \in [1..k]$ do
4.   if $\mu$ shares joining variable with window $W[i_j]$ then
5.     iterator $\leftarrow W[i_j].\text{probe}(\mu)$
6.     if iterator.size() < $\text{minCard}$ then
7.       $\text{minCard} \leftarrow \text{iterator.size()}$
8.       $\text{minProbeIdx} \leftarrow i_j$
9. return $\text{minProbeIdx}$;

7.1.2 Two-step$^+$ Adaptation

The more probing sequences are possible for $\mu_1 \bowtie \{R^2 \cdots R^n\}$ the harder it gets to compute the cost function given in Definition 7.2. We analyse the number of possible probing sequences of a join query based on its join graph. A join graph consists a set of windows as its vertices. Two vertices are connected if two windows share a join predicate (same mapping variable). Figure 7.1 depicts four join graphs for two 4-way joins and two 6-way joins. Note that each vertex is labeled with name of the shared mapping variable. Join graphs connected as a line as in Figure 7.1a and Figure 7.1c are called linear joins. If a mapping $\mu_1$ comes from both ends of the linear join, there is only one possible probing sequence. For instance, the only possible probing sequence for $\mu_1 \bowtie \{W_2, W_3, W_4\}$ is $\mu_1 \bowtie W_2 \bowtie W_3 \bowtie W_4$. Therefore, it is easy to estimate $C(\mu_1 \bowtie \{R^2 \cdots R^n\})$ in this case. For instance, by monitoring the probing operation with the given probing sequence, its cost can be stored in an extra field of the ring index entry of the index used for the join predicate.

![Join graphs](image)

Figure 7.1: Join graphs.

Another interesting property of linear joins is that the inner nodes have only two connections to two separate linear subjoins. This property can be used to efficiently estimate the probing sequences in a linear join by breaking the graph into smaller linear subjoins. The join graphs that have the star
and bushy topologies like Figure 7.1b and Figure 7.1d are called star join and bushy join, respectively. The higher the connectivity of the graph, the more complicated it becomes to compute the $\min$ part in Definition 7.2. However, we have found a useful property for the high connectivity vertices as shown in the following equation.

\[
\mu \bowtie (\{ R^i_1 \cdots R^i_m \}) = \sigma_\mu (R^i_1) \times \cdots \times \sigma_\mu (R^i_m) \quad \text{if} \quad \text{dom}(R^i_j) \cap \text{dom}(R^i_k) \subseteq \text{dom}(\mu) \quad \forall j, k \in [1..m] \land j \neq k
\] (7.7)

This property says that the Cartesian product of the all outputs of probing operations $\sigma_\mu (R^i_j)$, using the mapping $\mu$ on the input buffers $R^i_j$, is equivalent to the join operation $\mu \bowtie (\{ R^i_1 \cdots R^i_m \})$. The condition, called short-cut condition, is met when $\mu$ contains all shared variables of pairs of distinct input buffers. Based on these properties, we can provide an approximation function $\gamma_{approx}$ for $C((\mu_1 \circ \mu_2^*) \bowtie (\{ R^2 \cdots R^n \} \setminus \{ R^{i_2} \}))$. An improved cost approximation function is shown in Definition 7.8.

\[
C((\mu_1 \bowtie (\{ R^2 \cdots R^n \})) \approx \gamma_{\sigma} + \min_{i_2 \in \{2..n\}} \left( |\sigma_{\mu_1}(R^{i_2})| + \sum_{\mu_2^* \in \sigma_{\mu_1}(R^{i_2})} (\gamma_\times + \gamma_{approx}((\mu_1 \circ \mu_2^*) \bowtie (\{ R^2 \cdots R^n \} \setminus \{ R^{i_2} \}))) \right)
\] (7.8)

Based on this definition, we propose the two-step$^+$ algorithm to enable short-cut joins shown in Algorithm 15. Lines 2-11 check the short-cut condition. When the short-cut-condition is not met, the algorithm calls the sub-routine $\text{findNextProbWin\_MinCost}$ to find the next buffer to probe based on the approximate costs.

**Algorithm 15: probingPropagation\_TwoStep**

**Input:** $\mu$: propagating mapping, $k$ sliding windows $\{W[i_1],...,W[i_k]\}$

1. if $k==0$ then
2. if $\mu \bowtie \{W[i_1]\cdots W[i_k]\}$ satisfies the short-cut condition then
3. for $j \in [1..k]$ do
4. $I_j \leftarrow W[i_j].\text{probe}(\mu)$
5. if $I_j.\text{size}()==0$ then
6. return;
7. cartersianProduct($I_1 \cdots I_k$)
8. else
9. $i_{\text{next}} \leftarrow \text{findNextProbWin\_MinCost}(\mu,\{W[i_1],...,W[i_k]\})$
10. for $\mu^* \in W[i_{\text{next}}].\text{probe}(\mu)$ do
11. probePropagate($\mu \circ \mu^* , \{W_1,...,W_k\} \setminus \{W[i_{\text{next}}]\}$)
12. else
13. dispatch $\mu$

This sub-routine is described in Algorithm 16. It finds the next input buffer that potentially leads to the minimal cost value. Lines 5-19 check if all estimations $\gamma_{approx}$ are available. Then, the best probing
sequence will be determined in lines 17-20. If one of the estimations needed is not available, the algorithm will switch to the one-step algorithm to find the probing sequence as shown in lines 14-16. In the process, line 15 will turn on the operator monitor to measure the time spent for this probing sequence, and then assign this value to $\gamma_{\text{approx}}$ for the next executions. To direct the probing sequence to the short-cut condition, we assign a zero cost for $\gamma_{\text{approx}}(\mu \bowtie \{ W[i_1] \ldots W[i_k] \})$ if $\mu \bowtie \{ W[i_1] \ldots W[i_k] \}$ that meets the short-cut condition.

Algorithm 16: findNextProbWin МинCost

- **Input**: $\mu$: mapping to probe, $k$ sliding windows $\{ W[i_1], \ldots, W[i_k] \}$

1. if $\mu$ only shares variable with one window $W[i_j]$ then
2. return $i_j$;
3. else
4. \hspace{1em} minCost $\leftarrow$ MAXINT
5. \hspace{1em} for $j \in [1..k]$ do
6. \hspace{2em} if $\mu$ shares joining variable with window $W[i_j]$ then
7. \hspace{3em} $I \leftarrow W[i_j].\text{probe}(\mu)$
8. \hspace{3em} if $I.\text{size}() == 0$ then
9. \hspace{4em} return $i_j$
10. \hspace{2em} $C \leftarrow 0$
11. \hspace{2em} for $\mu_{i_j}^* \in I$ do
12. \hspace{3em} if $\gamma_{\text{approx}}((\mu \circ \mu_{i_j}^*) \bowtie \{ W[i_1] \ldots W[i_k] \}) \setminus \{ W[i_j] \})$ is already estimated then
13. \hspace{4em} $C \leftarrow C + I.\text{size}() \times \gamma_{\text{approx}}((\mu \circ \mu_{i_j}^*) \bowtie \{ W[i_1] \ldots W[i_k] \}) \setminus \{ W[i_j] \}))$
14. \hspace{2em} else
15. \hspace{3em} turn on the operator monitor to estimate $\gamma_{\text{approx}}((\mu \circ \mu_{i_j}^*) \bowtie \{ W[i_1] \ldots W[i_k] \}) \setminus \{ W[i_j] \})$
16. \hspace{2em} return findNextProbWin_OneStep($\mu, \{ W[i_1], \ldots, W[i_k] \}$);
17. if $\text{minCost} > C$ then
18. \hspace{1em} $\text{minCost} \leftarrow C$
19. \hspace{1em} $i_{\text{next}} \leftarrow i_j$
20. return $i_{\text{next}}$;

7.1.3 Experimental Results

We have conducted an experiment to evaluate the two algorithms for finding the best probing sequences. We compare them against the fixed probing sequences used in Chapter 6. We also compare them against the ESPER engine which serves as baseline. Similar to previous experiments, we have created an ad-hoc data structure specifically for each window buffer. We have measured the throughputs for the four types of queries given in Figure 7.1 with different window sizes. As shown in Chapter 6, the ratio of the number of distinct values in the join predicates to the window size has a significant impact in the throughput of the join operator. Hence, we fix this ratio in all test runs. To check the adaptability of the algorithms, in each test run, we stream mappings to the window buffers with three stages. In the first stage, we fill the window buffers with mappings to keep the selectivity on the join predicate at 100%. Then, in the next stage, we stream the same number of mappings but with a selectivity of 10% to test if the engines could adapt to the changes of the new selectivity. Finally, in the last stage, we
stream the same number of mappings with the selectivity of 100%.

Figure 7.2 reports on the experimental results. The results show that the two-step\(^+\) algorithm outperforms the other algorithms by orders of magnitude. For the 6-way join with the bushy graph, ESPER reported an out-of-heap-size error for window sizes greater than 100 mappings. In summary, the one-step algorithm performs 175%-400% better than the fixed probing sequence algorithm. The two-step\(^+\) algorithm provides on average a throughput which is twice as high as the one given by the one-step algorithm. The higher the likelihood of short-cut joins, the higher the throughput of the two-step\(^+\). In particular, the 4-way join with star graph and 6-way join with bushy graph produces more than 7 times and 10 times higher throughputs than the one-step-based version, respectively.

\begin{figure}[h]
\centering
\includegraphics[width=\textwidth]{figure7_2}
\caption{Join throughputs of optimisation algorithms.}
\end{figure}

7.2 Optimisation of Shared Window Joins

As highlighted in Chapter 5, the scalability in terms of number of concurrent queries could be ignored if an inter-query optimisation was in place. For different window joins which share join predicates
over the same RDF streams, not having inter-query optimisation results in a waste of resources due to redundant computation. This section utilises the data structures introduced in this chapter to build shared join algorithms to avoid redundant processing. A shared join has a single execution plan for multiple queries and produces multiple output streams for each separate query involved. The output streams are identified by the associated window sizes. The shared join operator consists of two components: the join component and the routing component. The join component produces a single intermediate output stream for all queries, and the routing component routes the valid output items to the corresponding output buffer of each continuous query [Hammad et al., 2003c]. The join component dominates the query load because the join operation is more expensive than the filtering operation of the routing component and the number of queries is usually smaller than the overall input sizes. Next, we present a multiple join operator for the case when the operator has the same set of input buffers for all queries involved. Then we provide a solution for the general case. Finally, we conduct an experiment to evaluate the performance of our algorithms.

7.2.1 Multiple Join Operator

To share the computations and the memory when processing multiple joins that have the same set of input buffers, we modify the multiway join algorithm from Chapter 6 to build a shared join operator, i.e, the multiple join operator. Let us assume \( m \) multiple window joins \( W_1^1 \cdots \bowtie W_n^m \) where \( j=1...k \) and \( W_j^j \) is a window buffer extracted from the RDF stream \( S_i^j \), \( i=1...n \). Let \( W_j^j_{max} \) be the window buffer that has a window size equal to the maximum window size over all \( W_j^j, j=1...k \). Then, the containment property [Hammad et al., 2003c] as shown in Definition 7.9 holds.

\[
W_1^1 \cdots \bowtie W_n^j \subseteq W_1^{max} \cdots \bowtie W_n^{max} \tag{7.9}
\]

Due to this property, the processing of the query \( W_1^{max} \cdots \bowtie W_n^{max} \) produces an output that contains the outputs of all queries \( W_j^j, j=1...k \). Therefore, the join component only has to execute a single multiway query for \( W_1^{max} \cdots \bowtie W_n^{max} \). In the routing component, each resulting mapping has to be routed to the query that takes it as an input. We call the routing component of the multiple join operator router. The router maintains a sorted list of the windows relevant to each join. The windows are ordered by window sizes in increasing order. Each output mapping is traversed to its constituent leaf-mappings to find the mappings which are valid within valid time intervals of the windows. When a mapping satisfies the time condition of a query, it is routed to the query’s output buffer. Figure 7.3 illustrates a multiple join operator for 3 queries over 2 streams \( S^1 \) and \( S^2 \) where \( Q_1 = W_1^1 \bowtie W_2^1 \), \( Q_2 = W_2^2 \bowtie W_2^2 \) and \( Q_3 = W_3^1 \bowtie W_3^2 \). This multiple join operator connects the 2-way join operator \( W_1^{max} \bowtie W_2^{max} \) to its router where \( W_1^{max} = W_3^1 \) and \( W_2^{max} = W_3^2 \). The left-hand side of the figure shows how the router delivers the output mappings from the 2-way join to each query. For instance, when the new mapping \( \langle a_1, b_6 \rangle \) arrives at the stream \( S^1 \), the 2-way join probes the input buffer \( W_2^{max} \) to generate two output mappings \( \langle a_1, b_6, c_1 \rangle \) and \( \langle a_1, b_6, c_5 \rangle \). Based on the window conditions of each query, the router routes \( \langle a_1, b_6, c_5 \rangle \) to \( Q_1 \) and \( Q_2 \) and both \( \langle a_1, b_6, c_1 \rangle \) and \( \langle a_1, b_6, c_5 \rangle \) to \( Q_3 \).
7.2.2 Networks of Multiple Join Operators for Shared Window Joins

In the general case, the concurrent queries registered to the system only share subsets of the streams involved in their queries. Therefore, we create a network of multiple join operators to enable sharing the execution of sub-queries for a group of queries. Each multiple join operator is created for a group of joins that share the same set of streams. Figure 7.4 illustrates a network for 4 queries over 4 streams $S^1$, $S^2$, $S^3$ and $S^4$, where $Q_1 = W_1^1 \bowtie W_1^2 \bowtie W_1^3$, $Q_2 = W_2^1 \bowtie W_2^3$, $Q_3 = W_3^2 \bowtie W_3^3$, and $Q_4 = W_4^2 \bowtie W_4^3$. This network is composed of three multiple join operators $\bowtie_1^M$, $\bowtie_2^M$ and $\bowtie_3^M$, where $\bowtie_1^M$ is for $Q_1$, $\bowtie_2^M$ is for $Q_2$ and $\bowtie_3^M$ for $Q_3$ and $Q_4$.

Figure 7.4: A Network of Multiple Join Operators.
To reduce the amount of memory allocated for the window buffers, the network only uses one buffer for each stream. This buffer contains all window buffers on that stream. For example, the input buffer $W_{max}^2$ on the stream $S^2$ contains the windows $W_1^2, W_2^2, W_3^2$ and $W_4^2$. Therefore, multiple join operators in the network share subsets of all input buffers created for the network. This enables an incremental evaluation for the network in the same way as for multiway joins. Algorithm 17 presents the probing propagation process in the network.

**Algorithm 17:** probingPropagationMultipleJoins

Input: $\mu$: propagating mapping, $k$ sliding windows $\{W[i_1],...,W[i_k]\}$, $m$ multiple join operators

1. for $\bowtie^M_i$ accepts $\mu$ as output do
2. dispatch $\mu$ to the router of $\bowtie^M_i$
3. if there is a multiple join operator among $\bowtie^M_1 \cdots \bowtie^M_m$ that consumes $\mu$ for further processing then
4. $\bowtie^M_{next} ⇝$ the smallest multiple join operator among $\bowtie^M_1 \cdots \bowtie^M_m$ that consumes $\mu$
5. $\mathcal{J} \mathcal{S} ←$ set of input windows of $\bowtie^M_{next}$ and but haven’t involved in generating $\mu$
6. $i_{next} ≜$ findNextProbeWin_MinCost($\mu, \mathcal{J} \mathcal{S}$)
7. for $\mu^* \in W[i_{next}].probe(\mu)$ do
8. probingPropagationMultipleJoins($\mu \circ \mu^*, \{W_1,...,W_k\} \backslash \{W[i_{next}]\}, \bowtie^M_1 \cdots \bowtie^M_m$)

The algorithm recursively propagates the probing operations to find possible outputs in the join component, before forwarding join outputs to the routing component. When a new mapping arrives at an input window, it will trigger a probing graph that contains all probing sequences of the multiple joins that consumes that new mapping. For instance, a new mapping $\mu_2$ in the stream $S^2$ will trigger the following probing sequences: $seq_1 = \mu_2 \bowtie^M_1 \{W_1^1, W_3^3\}$ for $\bowtie^M_1$, $seq_2 = \mu_2 \bowtie^M_2 W_3^3$ for $\bowtie^M_2$ and $seq_3 = \mu_2 \bowtie^M_3 W_4^4$ for $\bowtie^M_3$. They form the probing graph depicted in Figure 7.5, in which they share a probing sub-sequence $seq_2$. Note that the solid lines represent for probing sequences inside the multiway joins and the broken lines represent for routing paths from the multiway joins to the corresponding routers. The output mappings of this shared probing sequence can either be delivered to the router of $\bowtie^M_2$ or be used for further processing in $seq_1$ or $seq_3$.

![Figure 7.5: A probing graph.](image)

At each stage of the probing graph, the algorithm checks if the mapping used for the probing is the output of a multiple join operator. In this case, the mapping is sent to the router of this multiple join operator (lines 1–2). Line 3 is used to check if the propagated mapping can be used to generate joined results by a multiple join operator. If so, line 4 chooses the one that has the smallest set of inputs. This multiple join operator applies the two-step++ algorithm for choosing the best probing sequence on a
The algorithm is only applied to the subset of the input buffers that have been processed on the sequence created by propagated mappings.

### 7.2.3 Experimental Results

We have evaluated the performance of the multiple query optimisation algorithm by testing its scalability when the number of concurrent queries increases. The evaluation includes three cases: The first two cases are 2-way joins and 3-way joins that share the same 2 and 3 streams respectively. In the third case, there is a mixture of three query types, namely 2-way joins, 3-way join, and 4-way joins whose input streams are randomly chosen from 4 streams. The window size of each input stream is chosen randomly between 100,000 and 200,000 mappings. In each case we executed a series of runs for a number of queries ranging from 2 to 1,000. For each test run, we first register its concurrent queries, then we filled up all windows. After all windows were filled, we measured the processing throughput of the engine used. The engine consumes incoming data distributed evenly among 4 streams. In these experiments, we used ESPER with ad-hoc data structures for each stream as the baseline engine and we compared against the CQELS engine equipped with the multiple query optimiser.

Figure 7.6 shows the experimental results. We can see that the throughput of ESPER drops rapidly when the number of queries increases. The sudden stops in the plot line are due to out-of-heap error during the processing. In all tests, the throughput of CQELS only decreases marginally. This stems from the fact that adding more queries only adds small additional loads on the routers, while the number of streams are unchanged. Since the processing load only depends on the maximum window sizes of all query windows on each input stream, this guarantees scalability.
7.2. OPTIMISATION OF SHARED WINDOW JOINS

Figure 7.6: Join throughputs of multiple query optimisation.
Evaluations

We have already evaluated specific functionalities and algorithms of CQELS in the previous chapters. In this chapter, we provide an overall comparison of the complete CQELS system and compare it to the state-of-the-art systems. The evaluations are supported by our Linked Stream Benchmark framework, LSBench [Le-Phuoc et al., 2012a].

8.1 Evaluation Setup

8.1.1 Data Schema and Test Data Generator

For the evaluation, we assume a dataset coming from a social network application as described in Chapter 1. The data in a social network can be divided into two main categories: i) non-timestamped data (static data) containing the data that is not frequently changed or updated, and ii) streaming data containing the data that is updated or arrives frequently. Figure 8.1 shows the two data layers that contain the static data and stream data we will use in our tests.

Static data. The upper layer in Figure 8.1, i.e., the non-timestamped data layer $U_{\text{data}}$ contains user profile information (name, date of birth, location, relationship status, etc.), the relationships among users, and the channels in which users write posts, comments, etc.

Stream data. The lower layer in Figure 8.1, i.e., the stream data layer, contains various sources of streaming data such as posts and comments, photos, and GPS information.

- GPS stream ($S_{gps}$): Inspired by the use case in Live Social Semantics [Alani et al., 2009, Szomszor et al., 2010], we assume that each user has a GPS tracking device for sending updated information about his/her current location to the social network frequently. Each GPS information contains the latitude and longitude of a location and a timestamp.

- Posts and comments stream ($S_{pc}$): There is a large stream of posts and comments in a social network as users start or join discussions. Similar to the availability of the “wall” for each user in Facebook or the “Tweet timeline” for each Twitter user, in our generated social network, each user has his/her own channel for writing posts. The user becomes the moderator and obviously a subscriber of this channel. People who subscribe to this channel (or “follow” the moderator as in Twitter) can read and
reply to the posts and comments created in the channel. Each channel is used for the posting stream of a user. In this stream, we are particularly interested in the stream of “likes” (i.e., people who show their interest in a post), denoted by $S_{pclike}$, the stream of tags (i.e., the set of words representing the content of the discussion), and the stream of IP addresses from which people send the post or comment.

– Photo stream ($S_{fo}$): Uploaded photos and their associated attributes provide a lot of interesting information for discovering user habits, friend relationships as well as attractive spots, etc. In this stream data, we focus on exploiting useful information from the stream of user tags (i.e., list of users who are tagged in a photo), the stream of location information where the photo was taken, the stream of likes per photo ($S_{folike}$).

To simulate the social network data according to this schema, our LSBench provides a data generator, called Stream Social Network Data Generator (S2Gen). It generates data in consideration of the continuous query semantics [Arasu et al., 2006, Bolles et al., 2008, Barbieri et al., 2010a, Calbimonte et al., 2010, Anicic et al., 2011, Le-Phuoc et al., 2011] and various realistic data distributions, such as the skewed distributions of posts/comments. As window operators are primitive operators in a continuous query, the correlations among simulated data have an effect on the data windows over streams. To meet this requirement, S2Gen uses the “window sliding” approach from the structure-correlated social graph generator S3G2 [Minh Duc et al., 2012]. As such, to generate the stream data, S2Gen slides a window of users along all users in the social graph and creates social activities for each user (writing a post/comment, uploading photos, sending GPS tracking information). For creating a particular stream data, e.g., $S_{pc}$, S2Gen extracts all the posts/comments created for all the users, then sorts them according to their timestamps, and finally serializes these data to a file. A stream player is created in order to push the stream data from this file into a streaming engine. $S_{fo}$, $S_{pclike}$, $S_{folike}$, and $S_{gps}$ are created in a similar way.

For the static data, S2Gen generates the user profiles and the friendship information of all the users in order to form $U_{data}$. The details of this step and how to simulate the data correlations in the static data are the same as in S3G2. Note that all the generated stream data is correlated with non-stream data, e.g., user tags in the photo stream are correlated with friendship information. Various realistic
situations are also simulated while generating stream data, e.g., for GPS stream data, around a specific
time, the latitude and longitude sent by those people attending the same event are close to each other
and to the event’s location.
To allow a flexible data generation, S2Gen offers a range of parameters. The main parameters used in
the following experiments are:

− *Generating period*: The period in which the social activities are generated, e.g., 10 days, or one
month. By varying this parameter, one can create streams with different sizes for testing scalability.

− *Maximum number of posts/comments/photos for each user per week*: Each of these parameters
can be adjusted in order to change the amount of data that arrives in a window of time. It thus can
increase/decrease the input rate (e.g., number of triples/seconds) as the stream player pushes the data
according to a window of time. Besides, it also varies the total amount of generated streaming data for
a fixed generating period.

− *Correlation probabilities*: There are various parameters for the data correlations between graph data
and the graph structure, e.g., the probability that users will be connected if they are living in the same
area. They can be customised to specify how data is skewed according to each data attribute. The
tested systems need to recognise these correlation properties in order to optimise their query plan.

### 8.1.2 Experimental Setup

#### Datasets

We use the social network scenario generated by the above data generator. Here, the static data
corresponds to $U_{data}$, and the incoming streams are $S_{gps}$, $S_{pc}$, $S_{plike}$, $S_{fo}$ and $S_{folike}$. The data
generator considers many parameters to produce plausible input data, but for our experimental purpose,
we are interested in the size (number of triples/facts) of the static data and input streams, i.e., $|U_{data}|$,
$|S_{pc}|$, etc.

#### Test queries

We use 12 queries based on this scenario in the evaluations. The CQELS-QL representations of these
queries can be found in Appendix A.

(Q1) Notify of posts posted by a user identified by account id.

(Q2) Notify of posts posted by a user identified by a URI.

(Q3) Notify of posts posted by friends of a user.

(Q4) Notify of recent posts that have the same tag within 2 hours.

(Q5) Notify if a user has been tagged in a photo (within a day that a friend of his/her has liked the
photo).

(Q6) Notify a person that all comments on a post of a channel that he/she is subscribed to, have been
liked by one of his/her friends.

(Q7) Notify a user of all the friends having been tagged in a photo taken close to his current location
within 1 day and the photo has been liked by someone in the last 15 minutes.
(Q8) Notify a person tagged in photo of a person that has liked the photo but is not in the person’s friend list.

(Q9) Notify a user of all the posts and photos liked by friends of his/her friends.

(Q10) Count the number of posts with a certain hash tag within 20 minutes.

(Q11) Update the maximum number of users having liked a post.

(Q12) Get the 10 most used tags within two hours.

These queries can be divided into three groups:

- **Simple patterns**: The simplest query pattern is a filter which can be processed on the fly \((Q_1)\). \(Q_2\) needs to access a static graph with a filter pattern and then join it with a stream. However, this query can be processed as a filter pattern by materialising the intermediate results from sub-queries on the static data.

- **Join patterns**: \(Q_3\)–\(Q_7\) have joins on streams and static data with different number of joins. \(Q_8\) combines joins and a negation filter.\(^1\) \(Q_9\) includes a UNION operator in the query patterns.

- **Aggregation patterns**: \(Q_{10}\) is a simple counting aggregation. \(Q_{11}\) is more complicated with an aggregation combined with join and nestedness. \(Q_{12}\) is a top-k query with an aggregation pattern.

**Evaluated systems**

The setup to evaluate an engine \(E\) with a stream query \(Q\) is depicted in Figure 8.2. Suppose that \(Q\) requires as input a non-empty set of finite streams \(S_Q = \{S_1, \ldots, S_m\}, \ m \geq 1\), and possibly static data. Let \(R = \{r_1, \ldots, r_m\}\) be a set of stream rates (elements/sec) such that each \(S_i\) is fed into \(E\) at rate \(r_i\). We expect as the output a sequence of elements \(O(E, Q, R) = o_1, \ldots, o_n\), abbreviated by \(O_E\), when \(Q\) and \(R\) are clear from the context. We then record the outputs for computing correctness and throughputs.

![Figure 8.2: Evaluation design.](http://www.w3.org/TR/sparql11-query/#negation)

This evaluation design is general enough to capture the characteristics of stream engines. In our tests, \(E\) can be CQELS, C-SPARQL, and EP-SPARQL.\(^2\) In these tests, the static data is a set of RDF graphs, and each \(S_i\) is a Linked Stream Data, i.e., a sequence of RDF triples, and the output is a sequence of triples or SPARQL results.

\(^1\) [http://www.w3.org/TR/sparql11-query/#negation](http://www.w3.org/TR/sparql11-query/#negation)

\(^2\) The SPARQL\textsubscript{stream} implementation does not support native RDF data (confirmed by the main developer).
Note that EP-SPARQL is a wrapper of the Prolog-based stream engine ETALIS. When testing EP-SPARQL, we observed that it suffered considerably heavy loads from parsing big RDF graphs. Moreover, it does not support multi-threading to easily control the rates of input streams. Recently, JTALIS has been developed as a Java wrapper for ETALIS which does not exhibit the above parsing problems, as it works with facts and Prolog rules. Furthermore, using Java makes it very convenient to control input rates via multi-threading. The static data is a set of ground facts. Each $S_i$ as well as the output is a sequence of ground facts, and queries can be formalised as sets of Prolog rules. We thus compare CQELS, C-SPARQL, and JTALIS in the evaluations. All experiments are reproducible and recorded, and details are available at [http://code.google.com/p/lsbench/](http://code.google.com/p/lsbench/).

The engines were run on a Debian squeeze i5 3450 4x Intel Core i3 2.60GHz with 16GB RAM, running OpenJDK Runtime Environment (IcedTea6 1.8.10) OpenJDK 64-Bit Server VM, and SWI-Prolog 5.10.1.

### 8.2 Evaluations

#### 8.2.1 Functionality Tests

To evaluate query expressiveness support, we used the 12 queries described in the last section which cover different features and aspects of stream queries. Table 8.1 reports on the query patterns in detail and our observation of which queries can be successfully executed by which engines. We can see that a number of desired features are not yet satisfactorily covered by some engines. While CQELS supports all 12 queries, C-SPARQL reports syntax errors on $Q_7$ (complicated numeric filter), $Q_8$ (negation), and encounters runtime error on $Q_9$ (most complex query). Regarding JTALIS, patterns in $Q_5$-$Q_9$ are theoretically supported, but the queries produced no output. Also, there is no support for explicit representations of aggregations that works with timing windows. With the kind help from the JTALIS team, we encoded the simple counting aggregation in $Q_{10}$ by recursive rules, but left out the more complicated aggregations from $Q_{11}$ and $Q_{12}$.

<table>
<thead>
<tr>
<th>Patterns covered</th>
<th>$S$</th>
<th>$N_P$</th>
<th>$N_S$</th>
<th>Engines</th>
</tr>
</thead>
<tbody>
<tr>
<td>$Q_1$</td>
<td>✓</td>
<td>1</td>
<td>1</td>
<td>✓ ✓ ✓</td>
</tr>
<tr>
<td>$Q_2$</td>
<td>✓</td>
<td>✓ 2</td>
<td>1</td>
<td>✓ ✓ ✓</td>
</tr>
<tr>
<td>$Q_3$</td>
<td>✓</td>
<td>✓ 3</td>
<td>1</td>
<td>✓ ✓ ✓</td>
</tr>
<tr>
<td>$Q_4$</td>
<td>✓ ✓</td>
<td>✓ 4</td>
<td>1</td>
<td>✓ A ✓</td>
</tr>
<tr>
<td>$Q_5$</td>
<td>✓</td>
<td>✓ 3</td>
<td>2</td>
<td>✓ ✓ ✓ 0</td>
</tr>
<tr>
<td>$Q_6$</td>
<td>✓</td>
<td>✓ 5</td>
<td>2</td>
<td>✓ ✓ ✓ 0</td>
</tr>
<tr>
<td>$Q_7$</td>
<td>✓ ✓</td>
<td>✓ 10</td>
<td>3</td>
<td>✓ □ 0</td>
</tr>
<tr>
<td>$Q_8$</td>
<td>✓ ✓</td>
<td>✓ 3</td>
<td>2</td>
<td>✓ □ 0</td>
</tr>
<tr>
<td>$Q_9$</td>
<td>✓ ✓</td>
<td>✓ 4</td>
<td>2</td>
<td>✓ E 0</td>
</tr>
<tr>
<td>$Q_{10}$</td>
<td>✓ ✓</td>
<td>✓ 1</td>
<td>1</td>
<td>✓ ✓ ✓</td>
</tr>
<tr>
<td>$Q_{11}$</td>
<td>✓ ✓</td>
<td>✓ 1</td>
<td>1</td>
<td>✓ ✓ ✓ x</td>
</tr>
<tr>
<td>$Q_{12}$</td>
<td>✓ ✓</td>
<td>✓ 1</td>
<td>1</td>
<td>✓ ✓ ✓ x</td>
</tr>
</tbody>
</table>

Legend: **Agg**: Aggregate, **Nested**: Nested query, **S**: uses static data, $N_P$: number of patterns, $N_S$: number of streams, □: syntax error, A: ambiguous syntax, E: error, ∅: return no answer, x: not supported.

Table 8.1: Queries classification.

---

3We normalise the outputs to compare sets of facts and SPARQL result sets.
8.2.2 Basic Throughput Tests

As shown in [Le-Phuoc et al., 2012b], comparing input/output throughputs of CQELS, C-SPARQL, and JTALIS are invalid, thus, “comparable maximum execution throughput”\(^4\) is defined as a metric for comparing performance among these engines. The comparing throughputs are computed and validated with the LSBench validation tool. Table 8.2 shows the experimental results on the queries which are supported at least by two engines. In all the queries tested, CQELS performs faster than all other engines by several orders of magnitude. In particular, the maximum execution throughput of CQELS is over 10,000 times higher than the throughput of C-SPARQL and is 40-1000 times higher than the throughput of JTALIS.

<table>
<thead>
<tr>
<th></th>
<th>Q1</th>
<th>Q2</th>
<th>Q3</th>
<th>Q4</th>
<th>Q5</th>
<th>Q6</th>
<th>Q10</th>
<th>Q11</th>
<th>Q12</th>
</tr>
</thead>
<tbody>
<tr>
<td>CQELS</td>
<td>118924</td>
<td>96789</td>
<td>88467</td>
<td>60467</td>
<td>52890</td>
<td>44391</td>
<td>103698</td>
<td>37953</td>
<td>23965</td>
</tr>
<tr>
<td>C-SPARQL</td>
<td>10</td>
<td>1.68</td>
<td>1.63</td>
<td>10</td>
<td>1.72</td>
<td>1.71</td>
<td>10</td>
<td>2.7</td>
<td>1.3</td>
</tr>
<tr>
<td>JTALIS</td>
<td>3790</td>
<td>3857</td>
<td>1062</td>
<td>99</td>
<td>—</td>
<td>—</td>
<td>87</td>
<td>—</td>
<td>—</td>
</tr>
</tbody>
</table>

Table 8.2: Maximum execution throughput of CQELS, C-SPARQL and JTALIS for different queries.

8.2.3 Scalability Tests

In this section, we evaluate how the CQELS engine scales when we increase the static data size, window size and number of current queries.

The static data size experiment

For testing how the engines perform when the static data size increases, we choose 4 queries that involved the static dataset, i.e., Q2, Q3, Q5 and Q6. In this test, we increased the size of the static data from 100,000 triples to 100 million triples, then measure the throughputs of the engines. The results are shown in Figure 8.3. Note that JTALIS does not support query Q5 and Q6, thus, there is no data for these queries. In all queries, the CQELS engine scales well up to 100 million triples while other engines return out-of-memory error or do not return any result in the output streams for static data sizes greater than 1 million triples. The throughput of CQELS only decreases slightly when the static data size increases. This behaviour comes from the fact that when the size of the whole static dataset increases, the speed of accessing the relevant intermediate results for the query does not increase proportionally due to the caches and indexes.

---

\(^4\)In the following, “throughput” is always meant the refer to comparable maximum execution throughout.
Window size experiment

In this test, we varied the size of the windows involved in the queries. We modified queries Q3, Q6, Q10 and Q11 by changing their windows to count-based windows to have a fixed numbers of data items in a window during each test run. As JTALIS does not support count-based windows, we only compared the CQELS engine against the C-SPARQL engine. In all queries, we use the windows with the same size in each test run. We increased the window size from 100,000 to 10 million, then, we measured the throughput of the engines for each query per window size. Figure 8.4 shows the results of these experiments. The C-SPARQL engine did not return any result for tests on window sizes greater than 100,000 for queries Q3 and Q10 and the tests on window sizes greater than 10,000 with queries Q6 and Q11. On the other hand, the CQELS engine can perform stably on window sizes up to 10 million. The throughput of CQELS only decreases marginally when the window size increases.
CHAPTER 8. EVALUATIONS

<table>
<thead>
<tr>
<th>Number of queries experiment</th>
</tr>
</thead>
</table>

To test how the engines perform when the number of concurrent queries registered to the engine increases, we choose the queries Q3, Q4 and Q6 and Q11 as templates to generate concurrent queries to be registered to the engines. These queries contain multiway joins where the CQELS engine can benefit from the multiple query optimisation algorithms introduced in Chapter 7. Figure 8.5 shows the throughputs of the CQELS, C-SPARQL and JTALIS engines for 1 to 1000 concurrent queries. Note that JTALIS does not support queries Q6 and Q11, thus, there is no data for JTALIS on these queries. In all queries, other engines either do not return any results on the output stream or their throughputs decrease linearly when the number of queries increases. On the other hand, the throughput of the CQELS engine only decreases slightly when more current queries are registered to the engine.

![Figure 8.4: Scalability when varying the window size.](image-url)
8.2.4 Impact of Dictionary Throughput on The Performance

From the experimental results in the above evaluations, it can be seen that there are some gaps between the throughput of the test queries and the query operators implemented in Chapter 6. Table 8.3 summarises the minimum, average and maximum throughputs of all experiments on evaluation queries and incremental query operators conducted in Chapters 6, 7 and 8. In most cases, the average and maximum throughputs are noticeably higher than the throughput of the queries evaluated. As discussed in Section 5.1.2, the overhead of the encoding and decoding operations might contribute significantly to the query load for simple and very fast queries.

Therefore, we now investigate how the dictionary implementation contributes to the query execution time by analysing the time spent on encoding and decoding RDF nodes. We observed the behaviour of encoding and decoding operations by streaming different types of stream data. We use simulated streams including a post stream, a photo stream and a GPS stream. These three simulated streams have different ratios of RDF nodes in each stream. In particular, the post stream is dominated by literal string
nodes whilst the photo stream is dominated by URI nodes. The majority of data of the GPS stream are GPS coordinates, therefore, this stream contains more inline nodes than others. We also used the real dataset from Billion Triple Challenge\(^5\), by streaming it as a stream. The detailed figures of these experiments can be found in Appendix B. The encoding and decoding throughputs are summarised in last two columns in Table 8.3. These figures show that the minimum encoding/decoding throughputs are considerably lower than the average/maximum throughputs of the query operators. Hence, the dictionary might have a significant impact in the throughput of some simple queries. This explains the gap between the maximum throughput of the query operators and the evaluated queries.

### 8.3 Summary

The evaluations show that our CQELS engine is better than current state-of-art systems in all comparison aspects. For the query expensiveness in terms of SPARQL 1.1 query pattern support, our CQELS-QL covers all query patterns proposed by LSBench and the CQELS engine successfully processed them in all test runs. In general, our CQELS engine is faster than other engines by several orders of magnitude. Furthermore, the CQELS engine scales perfectly in all execution settings. For the handling large static data sizes, the CQELS engine can process datasets up to 100 millions triples without suffering the dramatic performance deterioration of other systems. While other systems can not handle the test queries with window sizes of more than 100,000 stream elements, the CQELS engine was able to scale up to 10 million stream elements with marginal changes in the execution throughput. When processing concurrent queries that possibly share computation, adding more queries to the CQELS engine only slightly decreases its throughput. However, the throughput of other systems degraded linearly with respect to the number of concurrent queries registered to these systems.

---

\(^5\)http://km.aifb.kit.edu/projects/btc-2011/

<table>
<thead>
<tr>
<th></th>
<th>Query</th>
<th>Join</th>
<th>Aggregate</th>
<th>Distinct</th>
<th>Minus</th>
<th>Encoding</th>
<th>Decoding</th>
</tr>
</thead>
<tbody>
<tr>
<td>Minimum throughput</td>
<td>23k</td>
<td>1k</td>
<td>1k</td>
<td>100k</td>
<td>80k</td>
<td>25k</td>
<td>37k</td>
</tr>
<tr>
<td>Average throughput</td>
<td>82k</td>
<td>300k</td>
<td>350k</td>
<td>400k</td>
<td>500k</td>
<td>93k</td>
<td>110k</td>
</tr>
<tr>
<td>Maximum throughput</td>
<td>110k</td>
<td>800k</td>
<td>400k</td>
<td>900k</td>
<td>1000k</td>
<td>400k</td>
<td>800k</td>
</tr>
</tbody>
</table>

**Table 8.3:** Summary of throughputs (triples/second): CQELS queries, query operators (join, aggregate, distinct, minus), and encoding/decoding operations (1000 triples/sec).
Chapter 9

Conclusion and Future work

9.1 Conclusion

This thesis has addressed a number of research questions for enabling highly performant processing of Linked Stream Data and Linked Data:

To address the issue of a query language suitable for processing Linked Stream Data and Linked Data, this thesis contributes an adaptive processing model for Linked Stream Data. Based on this processing model we propose CQELS-QL, one of the first continuous query languages for expressing queries over Linked Stream Data and Linked Data. The query language is defined using an abstract syntax based on the grammar of SPARQL 1.1 to make it compatible with the SPARQL query fragments. The semantics of CQELS-QL is formally defined using denotational semantics. The language overcomes several limitations of other continuous query languages: Changes on Linked Data are supported in the data model of CQELS-QL and the continuous semantics of CQELS-QL is formally defined. Furthermore, the ambiguities of other languages identified in Chapter 3 are addressed in CQELS-QL.

The drawbacks of using unmodified DSMSs and triple storages for Linked Stream Data and Linked Data following a blackbox architecture severely harm the throughput of stream processing engines. By analysing the performance issues of blackbox architectures, we proposed a whitebox architecture with an adaptive execution framework to overcome these problems. Our execution framework does not suffer from the overhead of data transformation and provides full control of the query execution process to achieve low-latency responses. Having full control of the query execution process also allows us to gain several orders of magnitudes in the performance in each execution phase compared to the state of the art.

Based on this adaptive execution framework, the challenges of physical organisation for Linked Stream Data and Linked Data are addressed by a hybrid approach which enables fast access on both large RDF datasets and high update throughput on RDF streams to meet the requirement of low-latency response. This approach carefully balances techniques from triple storages and DSMSs. By adopting a dictionary encoding technique for normalising RDF nodes in RDF streams and RDF datasets, the system allows operators to operate on small and fixed size integers which saves considerable amounts of memory and I/O load in comparison to using lexical versions of RDF nodes. For RDF datasets that do not fit into main memory, a caching mechanism exploits slow update rates and the persistent nature of continuous query patterns to pre-compute intermediate results of sub-queries on RDF datasets. The caching component of our execution framework also employs comprehensive indexing strategies taken from
triple storages to enable fast probing operations over pre-computed results. Our proposed execution framework shows significant performance gains against blackbox Linked Stream Data processing engines. It also overcomes the scalability issues of large RDF data sets that other systems could not cope with in our experiments.

For the challenge of building high throughput physical operators, we proposed novel operator-aware data structures associated with efficient incremental evaluation algorithms to deal with unusual properties of RDF stream data and query patterns. These data structures are designed to handle small data items and intermediate mappings contained in the processing state. The data structures include various low-maintenance cost indexes to support high throughput in the probing operations that are used in the different operator implementations. Based on such data structures, we propose several algorithms to enable incremental evaluation of basic operators such as join, duplicate elimination, and aggregation. These algorithms overcome typical issues of incremental evaluation of windowing operators identified in this thesis. They also show their advantage against other popular approaches such as direct-timestamp, negative tuple, and punctuation in terms of memory footprint, especially with a processing state containing exceptionally long lists of very small data items. Our experiments show a performance gain of orders of magnitude in all operator implementations with our data structures.

Our execution framework also provides optimisation solutions to further improve the performance of the execution engine. We propose adaptive cost-based optimisation algorithms for continuously re-ordering the join order of a multiway join operator to adapt to changes in the input streams. A solution for optimising multiple queries by extending the data structures for window buffers is also provided. Our experiments confirm that these solutions can improve the performance of the CQELS engine by orders of magnitude.

A general contribution of the thesis is that it provides systematic survey and extensive experiments enabling comparisons of to Linked Data Stream processing engines. The results of the system evaluations presented in thesis are the first evaluation datasets, a benchmarking system, and an evaluation methodology for Linked Data Stream processing engines.

9.2 Future work

In a next step, we will enable the CQELS engines to efficiently handle potential memory overflow issues by investigating how to extend our data structures to support spilling window buffers on disk, when memory consumption reaches a certain threshold. It would also be interesting to study how to adopt techniques and algorithms from adaptive caching for continuous queries [Chandrasekaran and Franklin, 2004, Dougdis et al., 2004, Babu et al., 2005b, Dougdis et al., 2004, Golab et al., 2006], partitioning sliding windows [Folkert et al., 2005], and stream warehouse [Golab and Özsu, 2010] to deal with disk-based RDF streams.

Distributing the stream processing across a cluster of machines to deal with high-speed data streams, large datasets and a large amount of registered queries, is another desirable feature we would like to support in a future version of the CQELS framework. We are looking into generic distributed real-time computing platforms such as S4\(^1\), Kafka\(^2\), Storm\(^3\) and Scribe\(^4\) to distribute our query operators over networked computing environments. Distributing computing over multiple computers has significant

\(^1\)http://incubator.apache.org/s4/
\(^2\)http://incubator.apache.org/kafka/
\(^3\)https://github.com/nathanmarz/storm
\(^4\)https://github.com/facebook/SCRIBE
9.2. FUTURE WORK

performance costs because network bandwidth and latency are several orders worse than RAM. On the other end, network latency is naturally much lower than disk latency. Therefore, the performance cost of storing and retrieving data on other nodes in a network is comparable to the cost of using disk (far less than with random access). However, on typical server hardware, the sequential access to disk is comparably faster than completely random access to RAM [Jacobs, 2009]. Therefore, finding a parallelising strategy that can efficiently utilizes RAM and disks on network computers to enable better scalability is interesting but challenging.

Moreover, there is an emerging trend of using the Cloud infrastructure, e.g. Amazon EC2, Google Cloud, for establishing an elastic cluster of computing nodes. Building a Linked Stream Data processing engine running on such an elastic cluster enables the engine to adapt the change of the processing load by adding or releasing processing nodes in the cluster at runtime. This elasticity is vital processing stream data due to its fluctuating streaming rate and the unpredictable computing complexity and the possibility of changing the number of concurrent query online. Therefore, enabling the elasticity for distributed version of CQELS is another feature we would like to support in the future. Combining the adaptability of CQELS query optimiser with the elasticity of Cloud version of CQELS is an interesting idea that is worth investigating as well.

For optimisation, we plan to adopt further optimisation techniques from DSMSs in our execution model. By evaluating their cost models and algorithms on different query shapes and data distribution, we will derive how to engineer and combine them in order to design more sophisticated optimisation strategies. On top of that, two important aspects of the computing environment that effect the performance of CQELS engines are CPU usage and memory consumption. Therefore, we will also study how to define related cost metrics to include them in the cost models of the optimisation algorithms. We also consider this optimisation problem in the distributed setting for the distributed version of CQELS above.
Evaluation Queries for Social Network Scenario

SELECT ?p ?o
WHERE {
STREAM <http://deri.org/poststream> [RANGE 1s] {
<http://www.ins.cwi.nl/sib/user/u984> ?p ?o.}
}

Query Q1: Notify of posts posted by a user identified by account id

PREFIX sioc: <http://rdfs.org/sioc/ns#>
SELECT ?post
WHERE {
?user sioc:account_of <http://www.ins.cwi.nl/sib/person/p984>.}

Query Q2: Notify of posts posted by a user identified by a URI

PREFIX foaf: <http://xmlns.com/foaf/0.1/>
PREFIX sioc: <http://rdfs.org/sioc/ns#>
SELECT ?friend ?post
WHERE {

Query Q3: Notify of posts posted by friends of a user

PREFIX sib: <http://www.ins.cwi.nl/sib/vocabulary/>
PREFIX sioc: <http://rdfs.org/sioc/ns#>
SELECT ?post1 ?post2 ?tag
WHERE {
STREAM <http://deri.org/poststream> [RANGE 15s] {
STREAM <http://deri.org/poststream> [RANGE 15s] {
FILTER(?post1 !=?post2)
}

Query Q4: Notify the recent posts that have the same tag within 2 hours
APPENDIX A. EVALUATION QUERIES FOR SOCIAL NETWORK SCENARIO

```
PREFIX sib: <http://www.ins.cwi.nl/sib/vocabulary/>
PREFIX foaf: <http://xmlns.com/foaf/0.1/>

SELECT ?friend1 ?friend2 ?photo
WHERE {
  STREAM <http://deri.org/likedphotostream> [RANGE 15 minutes]
  { ?friend2 sib:like ?photo. }
  STREAM <http://deri.org/photostream> [RANGE 1 days]
  { ?photo sib:usertag ?friend1. }
}
```

Query Q5: Notify if a user has been tagged in a photo (within a day that a friend of his/her has liked the photo)

```
PREFIX sib: <http://www.ins.cwi.nl/sib/vocabulary/>
PREFIX foaf: <http://xmlns.com/foaf/0.1/>
PREFIX sioc: <http://rdfs.org/sioc/ns#>

SELECT ?user ?friend ?comment ?channel
WHERE {
  STREAM <http://deri.org/poststream> [RANGE 1m]
  STREAM <http://deri.org/likedpoststream> [RANGE 100ms]
  { ?friend sib:like ?comment. }
}
```

Query Q6: Notify a person that all comments on a post of a channel that he/she is subscribed to have been liked by one of his/her friends

```
PREFIX sib: <http://www.ins.cwi.nl/sib/vocabulary/>
PREFIX foaf: <http://xmlns.com/foaf/0.1/>

SELECT ?friend ?photo
WHERE {
  STREAM <http://deri.org/photostream> [RANGE 2 hours]
  STREAM <http://deri.org/likedphotostream> [RANGE 15 minutes]
  { ?someone sib:like ?photo}
  STREAM <http://deri.org/gpsstream> [60 seconds]
  GRAPH <http://deri.org/userprofile/>
  { ?user foaf:name "$name$". ?user foaf:knows ?friend.}
  FILTER (((?lat1-?lat2)*(?lat1-?lat2)+(?lon1-?lon2)*(?lon1-?lon2)<$distance*$distance$)}
}
```

Query Q7: Notify a user of all the friends having been tagged in a photo taken close to his current location within 1 day and the photo has been liked by someone in the last 15 minutes
Query Q8: Notify a person tagged in photo of a person that has liked the photo but is not in the person’s friend list

```
PREFIX sib: <http://www.ins.cwi.nl/sib/vocabulary/>
PREFIX foaf: <http://xmlns.com/foaf/0.1/>
SELECT ?friend1 ?friend2
WHERE {
  STREAM <http://deri.org/photostream> [RANGE 2 hours]
    { ?photo sib:usertag ?friend1}
  STREAM <http://deri.org/ikedphotostream> [RANGE 5 minutes]
    { friend2 sib:like ?photo}
  FILTER NOT EXISTS {?friend2 foaf:knows ?friend1}
}
```

Query Q9: Notify a user of all the posts and photos liked by friends of his/her friends

```
PREFIX sib: <http://www.ins.cwi.nl/sib/vocabulary/>
PREFIX foaf: <http://xmlns.com/foaf/0.1/>
SELECT ?post
WHERE {
  STREAM <http://deri.org/photostream> [RANGE 20 minutes] {?friendofriend sib:like ?post}
  UNION
  STREAM <http://deri.org/poststream> [RANGE 20 minutes] {?friendofriend sib:like ?post}
  GRAPH <http://deri.org/userprofile/>
    {me foaf:knows ?friend. ?friend :knows ?friendofriend.}
}
```

Query Q10: Count the number of posts with a certain hash tag within 20 minutes

```
PREFIX sib: <http://www.ins.cwi.nl/sib/vocabulary/>
PREFIX dc: <http://purl.org/dc/elements/1.1/>
SELECT ?hashTag count(?post)
WHERE {
  STREAM <http://deri.org/poststream> [RANGE 20 minutes]
    {?post sib:hashTag ?hashTag}
}
```

Query Q11: Update the maximum number of users having liked a post

```
PREFIX sib: <http://www.ins.cwi.nl/sib/vocabulary/>
PREFIX dc: <http://purl.org/dc/elements/1.1/>
SELECT ?creator MAX(?userlike)
WHERE {
  SELECT ?creator ?post (COUNT(?user) AS ?userlike)
  WHERE {
    STREAM <http://deri.org/poststream> [RANGE 2 hours] 
      {?post dc:creator ?creator}
    STREAM <http://deri.org/likedpoststream> [RANGE 2 hours] 
      {?user sib:like ?post}
  }
  GROUP BY ?creator, ?post
}
GROUP BY ?creator
```
Query Q12: Get the 10 most used tags within two hours
This appendix reports the experiments conducted to analyse the time spent on encoding and decoding RDF nodes. We tested the behaviour of encoding and decoding operations by streaming different types of stream data. We used simulated streams including a post stream, a photo stream and a GPS stream. These three simulated streams have different ratios of RDF nodes in each stream. In particular, the post stream is dominated by literal string nodes whilst the photo stream is dominated by URI nodes. The majority of data of the GPS stream are GPS coordinates, therefore, this stream contains more inline nodes than others. We also used the real dataset from Billion Triple Challenge\(^1\), by streaming it as a stream. The detailed results are shown in follows.

\[\text{Figure B.1: Encoding time of simulated post stream.}\]

\(^1\)http://km.aifb.kit.edu/projects/btc-2011/
Figure B.2: Decoding time of simulated post stream.

Figure B.3: Encoding time of simulated Photo stream.
Figure B.4: Decoding time of simulated Photo stream.

Figure B.5: Encoding time of simulated GPS stream.
APPENDIX B. DICTIONARY THROUGHPUT EVALUATIONS

Figure B.6: Decoding time of simulated GPS stream.

Figure B.7: Encoding time of Billion Triple Challenge 2011 dataset.
Figure B.8: Decoding time of Billion Triple Challenge 2011 dataset.


P. Minh Duc, P. A. Boncz, and O Erling. S3g2: A Scalable Structure-Correlated Social Graph Generator. 2012.


