

# CONTINUOUS QUERIES OVER DATA STREAMS

## – SEMANTICS AND IMPLEMENTATION



Dissertation  
zur Erlangung des Doktorgrades  
der Naturwissenschaften  
(Dr. rer. nat.)

dem Fachbereich Mathematik und Informatik  
der Philipps-Universität Marburg  
vorgelegt von

Jürgen Krämer  
aus Alsfeld-Leusel

Marburg an der Lahn 2007

Vom Fachbereich Mathematik und Informatik  
der Philipps-Universität Marburg  
als Dissertation am 18. Juni 2007 angenommen.

Erstgutachter: Prof. Dr. Bernhard Seeger

Zweitgutachter: Prof. Dr. Bernd Freisleben

Drittgutachter: Prof. Dr. M. Tamer Özsu

Tag der mündlichen Prüfung am 22. Juni 2007

# Abstract

Recent technological advances have pushed the emergence of a new class of data-intensive applications that require continuous processing over sequences of transient data, called data streams, in near real-time. Examples of such applications range from online monitoring and analysis of sensor data for traffic management and factory automation to financial applications tracking stock ticker data. Traditional database systems are deemed inadequate to support high-volume, low-latency stream processing because queries are expected to run continuously and return new answers as new data arrives, without the need to store data persistently. The goal of this thesis is to develop a solid and powerful foundation for processing continuous queries over data streams. Resource requirements are kept in bounds by restricting the evaluation of continuous queries to sliding windows over the potentially unbounded data streams. This technique has the advantage that it emphasizes new data, which in the majority of real-world applications is considered more important than older data. Although the presence of continuous queries dictates rethinking the fundamental architecture of database systems, this thesis pursues an approach that adapts the well-established database technology to the data stream computation model, with the aim to facilitate the development and maintenance of stream-oriented applications. Based on a declarative query language inheriting the basic syntax from the prevalent SQL standard, users are able to express and modify complex application logic in an easy and comprehensible manner, without requiring the use of custom code. The underlying semantics assigns an exact meaning to a continuous query at any point in time and is defined by temporal extensions of the relational algebra. By carrying over the well-known algebraic equivalences from relational databases to stream processing, this thesis prepares the ground for powerful query optimizations. A unique time-interval based stream algebra implemented with efficient online algorithms allows for processing data in a push-based fashion. A performance analysis, along with experimental studies, confirms the superiority of the time-interval approach over comparative approaches for the predominant set of continuous queries. Based upon this stream algebra, this thesis addresses architectural issues of an adaptive and scalable runtime environment that can cope with varying query workload and fluctuating data stream characteristics arising from the highly dynamic and long-running nature of streaming applications. In order to control the resource allocation of continuous queries, novel adaptation techniques are investigated, trading off answer quality for lower resource requirements. Moreover, a general migration strategy is developed that enables the query processing engine to re-optimize continuous queries at runtime. Overall, this thesis outlines the salient features and operational functionality of the stream processing infrastructure *PIPES* (*Public Infrastructure for Processing and Exploring Streams*), which has already been applied successfully in a variety of stream-oriented applications.



# Zusammenfassung

Der rapide technologische Fortschritt der vergangenen Jahre hat die Entwicklung einer neuen Klasse von Anwendungen begünstigt, die sich dadurch auszeichnen, dass enorme Datenmengen in Form von Datenströmen bereitgestellt und kontinuierlich verarbeitet werden müssen, um zeitnah wichtige Informationen und Kennzahlen zu ermitteln. Beispiele für Anwendungen finden sich in den unterschiedlichsten Bereichen, die sich von der Überwachung und Auswertung von Sensordaten im Verkehrsmanagement oder der Fabrikautomation bis hin zur Trenderkennung in Börsenkursen erstrecken. Konventionelle Datenbanksysteme sind für die erforderliche kontinuierliche Anfrageverarbeitung, bei der die eintreffenden Daten möglichst direkt und ohne vollständige Zwischenspeicherung verarbeitet werden müssen, nicht ausgelegt. Das Ziel dieser Arbeit besteht darin, eine solide Grundlage zur adäquaten Verarbeitung kontinuierlicher Anfragen auf Datenströmen bereitzustellen. Um die Ressourcenanforderungen bei der Verarbeitung zu begrenzen, beziehen sich kontinuierliche Anfragen auf gleitende Fenster über den potentiell unbeschränkt großen Datenströmen. Dieses Vorgehen bietet den Vorteil, dass sich die Ergebnisse einer Anfrage stets auf die aktuellen Daten beziehen, die üblicherweise für die Anwendungen von höherer Relevanz sind. Damit Anwender einen einfachen Zugang zu den neu entwickelten Verfahren finden können, orientiert sich diese Arbeit an bewährter Datenbanktechnologie und adaptiert diese auf das Datenstrommodell. Eine deklarative, eng an den weit verbreiteten SQL-Standard angelehnte Anfragesprache erlaubt es Anwendern, komplexe Applikationslogik auf einfache Weise auszudrücken. Die zu Grunde liegende, aussagekräftige Anfragesemantik basiert auf temporalen Erweiterungen der relationalen Algebra. Darauf aufbauend ist es gelungen, die aus Datenbanken bekannten algebraischen Äquivalenzen auf kontinuierliche Anfragen über Datenströmen zu übertragen, wodurch eine hervorragende Grundlage zur Anfrageoptimierung geschaffen wurde. Für die datengetriebene Verarbeitung sorgt eine bislang einzigartige zeitintervallbasierte Datenstromalgebra umgesetzt durch effiziente Online-Algorithmen. Mittels einer Performanzanalyse gestützt durch experimentelle Studien wird belegt, dass der Zeitintervall-Ansatz für einen Großteil der Anfragen konkurrierenden Ansätzen deutlich überlegen ist. Darüber hinaus widmet sich die Arbeit architekturellen Gesichtspunkten einer adaptiven und skalierbaren Laufzeitumgebung, die in der Lage ist, sich einer variierenden Anfragelast sowie sich über die Zeit ändernden Datenstromcharakteristika anzupassen. Insbesondere werden neue Verfahren vorgestellt, um die Ressourcenallokation von Anfragen zu steuern und Anfragen zur Laufzeit zu optimieren. Die im Rahmen dieser Arbeit entwickelte Funktionalität bildet den Kern der Softwareinfrastruktur *PIPES* (*Public Infrastructure for Processing and Exploring Streams*), die sich bereits in diversen Anwendungsbereichen als ein mächtiges und nützliches Werkzeug zur Verarbeitung von Datenströmen bewährt hat.



# Acknowledgments

First of all, I am extremely grateful to my adviser, Prof. Dr. Bernhard Seeger, for his invaluable guidance throughout my doctoral studies. In spite of a busy schedule, he has been readily available for advice, reading, or simply a word of encouragement. I learned a lot from him about good research practice and what it takes to achieve this goal. I would also like to thank all the members of my committee for the time and energy they have devoted to reading my work.

I express my gratitude to all the database group members for their great interest in the topic of my thesis, their critical feedback and suggestions, and the pleasant working atmosphere. In particular, I am grateful to Michael Cammert, Christoph Heinz, Tobias Riemenschneider, and Sonny Vaupel for all the help and encouragement they have given me, including the many fruitful discussions on reasonable query semantics and implementation design. Special thanks also go to Michael Cammert, Heike and Patrick Seitz, and Ben Mills for proof-reading this thesis.

I am grateful to have had the opportunity to work with Yin Yang and Prof. Dr. Dimitris Papadias on the dynamic plan migration problem. In addition, I would like to thank Prof. Dr. Thomas Penzel and Prof. Dr. Richard Lenz for the inspiring, multidisciplinary discussions about data stream processing in sleep medicine. I am thankful to Ralph Langner and his team for providing us with the commercial i-Plant enterprise version, along with professional support at no charge, so that we are able to employ our stream processing infrastructure PIPES in highly automated manufacturing environments.

Finally, I would not have reached this point in my academic career without the support and unconditional love of my parents and my sister. I am forever indebted to my wonderful wife Birgit and my son Julian for their love, understanding, patience, and encouragement when it was most required. Last, but not least, I owe my thanks to all my friends who believed in me and kept me smiling.

The research described in this thesis was part of the project “Anfrageverarbeitung aktiver Datenströme” supported by grants No. SE-553/4-1 and SE-553/4-3 from the German Research Foundation.



# Contents

<b>List of Figures</b>	<b>xvii</b>
<b>List of Tables</b>	<b>xix</b>
<b>List of Algorithms</b>	<b>xxi</b>
<b>I Introduction</b>	<b>1</b>
<b>1 A New Class of Data Management Applications</b>	<b>3</b>
1.1 Data Stream Model . . . . .	5
1.2 Differences between DBMSs and DSMSs . . . . .	5
<b>2 Research Challenges</b>	<b>9</b>
2.1 Query Formulation . . . . .	9
2.2 Semantics for Continuous Queries . . . . .	9
2.3 Stream Algorithms . . . . .	10
2.4 Adaptive Query Execution . . . . .	10
<b>3 Contributions</b>	<b>13</b>
<b>4 Thesis Outline</b>	<b>17</b>
<b>II Query Semantics and Implementation</b>	<b>19</b>
<b>5 Introduction</b>	<b>21</b>
<b>6 Preliminaries</b>	<b>25</b>
6.1 Time . . . . .	25
6.2 Tuples . . . . .	25
6.3 Stream Representations . . . . .	25
6.3.1 Raw Streams . . . . .	26
6.3.2 Logical Streams . . . . .	27
6.3.3 Physical Streams . . . . .	27
6.4 Value-Equivalence . . . . .	29
6.5 Stream Transformations . . . . .	29
6.5.1 Raw Stream To Physical Stream . . . . .	29
6.5.2 Raw Stream to Logical Stream . . . . .	30

6.5.3	Physical Stream To Logical Stream . . . . .	30
6.6	Query Processing Steps . . . . .	31
6.7	Running Example . . . . .	31
<b>7</b>	<b>Query Formulation</b>	<b>33</b>
7.1	Graphical Interface . . . . .	33
7.2	Query Language . . . . .	33
7.2.1	Characteristics . . . . .	33
7.2.2	Registration of Source Streams . . . . .	34
7.2.3	Continuous Queries . . . . .	35
7.2.4	Registration of Derived Streams . . . . .	37
7.2.5	Nested Queries . . . . .	38
7.2.6	Comparison to CQL . . . . .	39
<b>8</b>	<b>Logical Operator Algebra</b>	<b>41</b>
8.1	Standard Operators . . . . .	42
8.1.1	Filter . . . . .	42
8.1.2	Map . . . . .	42
8.1.3	Union . . . . .	43
8.1.4	Cartesian Product . . . . .	43
8.1.5	Duplicate Elimination . . . . .	44
8.1.6	Difference . . . . .	45
8.1.7	Grouping . . . . .	46
8.1.8	Scalar Aggregation . . . . .	47
8.1.9	Derived Operators . . . . .	48
8.2	Window Operators . . . . .	49
8.2.1	Time-based Sliding Window . . . . .	49
8.2.2	Count-based Sliding Window . . . . .	50
8.2.3	Partitioned Window . . . . .	52
<b>9</b>	<b>Logical Plan Generation</b>	<b>55</b>
9.1	Window Placement . . . . .	55
9.2	Default Windows . . . . .	55
9.3	Examples . . . . .	56
<b>10</b>	<b>Algebraic Query Optimization</b>	<b>59</b>
10.1	Snapshot-Reducibility . . . . .	59
10.2	Equivalences . . . . .	60
10.3	Transformation Rules . . . . .	60
10.3.1	Conventional Rules . . . . .	61
10.3.2	Window Rules . . . . .	61
10.4	Applicability of Transformation Rules . . . . .	62
10.4.1	Snapshot-Reducible Subplans . . . . .	63
10.4.2	Non-Snapshot-Reducible Subplans . . . . .	64
10.4.3	Extensions to Multiple Queries . . . . .	64

<b>11 Physical Operator Algebra</b>	<b>65</b>
11.1 Basic Idea . . . . .	65
11.2 Operator Properties . . . . .	66
11.2.1 Operator State . . . . .	66
11.2.2 Nonblocking Behavior . . . . .	67
11.2.3 Ordering Requirement . . . . .	67
11.2.4 Temporal Expiration . . . . .	68
11.3 SweepAreas . . . . .	68
11.3.1 Abstract Data Type SweepArea . . . . .	68
11.3.2 Use of SweepAreas . . . . .	70
11.3.3 Implementation Considerations . . . . .	71
11.4 Notation . . . . .	72
11.4.1 Algorithm Structure . . . . .	72
11.4.2 Specific Syntax and Symbols . . . . .	73
11.5 Standard Operators . . . . .	73
11.5.1 Filter . . . . .	73
11.5.2 Map . . . . .	74
11.5.3 Union . . . . .	75
11.5.4 Cartesian Product / Theta-Join . . . . .	76
11.5.5 Duplicate Elimination . . . . .	80
11.5.6 Difference . . . . .	82
11.5.7 Scalar Aggregation . . . . .	87
11.5.8 Grouping with Aggregation . . . . .	93
11.6 Window Operators . . . . .	97
11.6.1 Time-Based Sliding Window . . . . .	97
11.6.2 Count-Based Sliding Window . . . . .	98
11.6.3 Partitioned Window . . . . .	101
11.7 Split and Coalesce . . . . .	105
11.7.1 Split . . . . .	105
11.7.2 Coalesce . . . . .	107
<b>12 Physical Query Optimization</b>	<b>111</b>
12.1 Equivalences . . . . .	111
12.2 Coalesce and Split . . . . .	111
12.2.1 Semantic Impact . . . . .	112
12.2.2 Runtime Effects . . . . .	112
12.3 Physical Plan Generation . . . . .	113
12.3.1 Standard Operators . . . . .	113
12.3.2 Window Operators . . . . .	114
12.3.3 Split Omission . . . . .	116
12.3.4 Expiration Patterns . . . . .	116
<b>13 Query Execution</b>	<b>119</b>
13.1 Design Goals . . . . .	119
13.2 Query Graph . . . . .	119

13.3 Query Processing Fundamentals . . . . .	120
13.4 Runtime Environment . . . . .	120
13.4.1 Metadata . . . . .	121
13.4.2 Scheduling . . . . .	121
13.4.3 Resource Management . . . . .	121
13.4.4 Query Optimization . . . . .	122
13.5 Dealing with Liveness and Disorder . . . . .	122
13.5.1 Time Management . . . . .	122
13.5.2 Problem Definition . . . . .	123
13.5.3 Heartbeats in Query Plans . . . . .	123
13.5.4 Stream Ordering . . . . .	123
13.6 Auxiliary Functionality . . . . .	124
13.7 Development Status of PIPES . . . . .	124
<b>14 Implementation Comparison</b>	<b>127</b>
14.1 The Positive-Negative Approach . . . . .	127
14.2 Features . . . . .	127
14.2.1 Window Operators . . . . .	128
14.2.2 Stateless Operators . . . . .	128
14.2.3 Join and Duplicate Elimination . . . . .	129
14.2.4 Grouping, Aggregation, and Difference . . . . .	130
14.2.5 General System Overhead . . . . .	131
14.3 Experimental Evaluation . . . . .	131
14.3.1 Queries with Stateless Operators . . . . .	133
14.3.2 Queries with Stateful Operators . . . . .	135
14.3.3 Count-based Sliding Windows . . . . .	138
14.3.4 Physical Optimizations . . . . .	139
14.4 Lessons Learned . . . . .	142
<b>15 Expressiveness</b>	<b>145</b>
15.1 Comparison to CQL . . . . .	145
15.1.1 Query Translation . . . . .	145
15.1.2 Formal Reasoning . . . . .	146
15.2 Beyond CQL . . . . .	148
<b>16 Related Work</b>	<b>151</b>
16.1 Use of Timestamps . . . . .	151
16.2 Stream Algebras . . . . .	152
16.3 Extended Relational and Temporal Algebra . . . . .	154
16.4 Stream Processing Systems . . . . .	156
16.4.1 STREAM . . . . .	156
16.4.2 TelegraphCQ . . . . .	157
16.4.3 Aurora . . . . .	157
16.4.4 Gigascope . . . . .	159
16.4.5 StreamMill . . . . .	159

16.4.6 Nile . . . . .	159
16.4.7 Tapestry . . . . .	160
16.4.8 Tribeca . . . . .	161
16.4.9 Cape . . . . .	161
16.4.10 Cayuga . . . . .	161
16.4.11 XML Stream Processing . . . . .	161
16.5 Further Related Approaches . . . . .	163
16.5.1 Sequence Databases . . . . .	163
16.5.2 Materialized Views . . . . .	163
16.5.3 Active Databases . . . . .	163
16.5.4 Sensor Databases and Networks . . . . .	164
<b>17 Extensions</b>	<b>167</b>
17.1 Window Advance . . . . .	167
17.1.1 Time-based Sliding Window . . . . .	168
17.1.2 Count-based Sliding Window . . . . .	169
17.1.3 Partitioned Window . . . . .	172
17.2 CQL Support . . . . .	172
17.2.1 Istream . . . . .	172
17.2.2 Dstream . . . . .	173
17.3 Converters . . . . .	174
17.4 Relations . . . . .	175
17.4.1 Conversion into Streams . . . . .	176
17.4.2 Provision of Specific Operators . . . . .	176
17.4.3 Output Relations . . . . .	177
<b>18 Conclusions</b>	<b>179</b>
<b>III Cost-Based Resource Management</b>	<b>181</b>
<b>19 Introduction</b>	<b>183</b>
<b>20 Granularity Conversion</b>	<b>185</b>
20.1 Time Granularity . . . . .	185
20.2 Query Language Extensions for Time Granularity Support . . . . .	185
20.3 Semantics . . . . .	186
20.4 Implementation . . . . .	187
20.5 Plan Generation . . . . .	187
20.6 Semantic Impact . . . . .	188
20.7 Physical Impact . . . . .	189
<b>21 Adaptation Approach</b>	<b>191</b>
21.1 Overview . . . . .	191
21.2 Query Language Extensions for QoS constraints . . . . .	191
21.3 Adaptation Techniques . . . . .	193

21.3.1 Adjusting the Window Size . . . . .	193
21.3.2 Adjusting the Time Granularity . . . . .	193
21.4 Runtime Considerations . . . . .	194
21.4.1 Granularity Synchronization . . . . .	194
21.4.2 QoS Propagation . . . . .	195
21.4.3 Change Histories . . . . .	195
21.5 Semantic Guarantees . . . . .	196
<b>22 Cost Model</b>	<b>197</b>
22.1 Purpose . . . . .	197
22.2 Model Parameters . . . . .	197
22.2.1 Meaning . . . . .	198
22.2.2 Initialization . . . . .	198
22.2.3 Updates . . . . .	199
22.2.4 Justification for the Choice of Model Parameters . . . . .	199
22.3 Parameter Estimation . . . . .	200
22.3.1 Preliminaries . . . . .	201
22.3.2 Operator Formulas . . . . .	201
22.4 Estimating Operator Resource Allocation . . . . .	207
22.4.1 Filter and Map . . . . .	207
22.4.2 Union . . . . .	207
22.4.3 Join . . . . .	208
22.4.4 Duplicate Elimination . . . . .	210
22.4.5 Scalar Aggregation . . . . .	213
22.4.6 Grouping with Aggregation . . . . .	215
22.4.7 Split . . . . .	217
22.4.8 Granularity Conversion . . . . .	219
22.4.9 Time-Based Sliding Window . . . . .	220
22.4.10 Count-Based Sliding Window . . . . .	220
22.4.11 Partitioned Window . . . . .	221
<b>23 Adaptive Resource Management</b>	<b>223</b>
23.1 Runtime Adaptivity . . . . .	223
23.1.1 Changes in Query Workload . . . . .	223
23.1.2 Changes in Stream Characteristics . . . . .	223
23.2 Adaptation Effects . . . . .	224
23.2.1 Changes to the Window Size . . . . .	224
23.2.2 Changes to the Time Granularity . . . . .	224
23.3 Adaptation Strategy . . . . .	225
23.3.1 Optimization Problem . . . . .	225
23.3.2 Heuristic Solution . . . . .	225
23.4 Extensions and Future Work . . . . .	225
<b>24 Experiments</b>	<b>227</b>
24.1 Adjustments to the Window Size . . . . .	227

24.2 Adjustments to the Time Granularity . . . . .	229
24.3 Window Reduction vs. Load Shedding . . . . .	229
24.4 Scalability and Adaptivity . . . . .	231
<b>25 Related Work</b>	<b>235</b>
25.1 Costs Models . . . . .	235
25.2 Estimation of Resource Usage . . . . .	236
25.3 Load Shedding and Approximate Queries . . . . .	236
25.4 Scheduling . . . . .	237
25.5 Applicability . . . . .	237
<b>26 Conclusions</b>	<b>239</b>
<b>IV Dynamic Plan Migration</b>	<b>241</b>
<b>27 Introduction</b>	<b>243</b>
<b>28 Problems of the Parallel Track Strategy</b>	<b>245</b>
28.1 Parallel Track Strategy . . . . .	245
28.2 Problem Definition . . . . .	245
<b>29 A General Strategy for Plan Migration</b>	<b>249</b>
29.1 Logical View . . . . .	249
29.1.1 GenMig Strategy . . . . .	249
29.1.2 Correctness . . . . .	249
29.2 Physical View . . . . .	250
29.2.1 Implementation of GenMig for the Time-Interval Approach . . . . .	250
29.2.2 Correctness . . . . .	252
29.3 Performance Analysis . . . . .	254
29.4 Optimizations . . . . .	255
29.4.1 Reference Point Method . . . . .	255
29.4.2 Shortening Migration Duration . . . . .	256
29.5 Implementation of GenMig for the Positive-Negative Approach . . . . .	256
29.6 Extensions . . . . .	256
<b>30 Experimental Evaluation</b>	<b>259</b>
<b>31 Related Work</b>	<b>263</b>
<b>32 Conclusions</b>	<b>265</b>
<b>V Summary of Conclusions and Future Research Directions</b>	<b>267</b>
<b>33 Summary of Conclusions</b>	<b>269</b>

*Contents*

---

<b>34 Future Research Directions</b>	<b>271</b>
<b>Bibliography</b>	<b>275</b>

# List of Figures

1.1	Comparison between query processing in DBMSs and DSMSs . . . . .	6
6.1	Overview of different stream representations . . . . .	26
6.2	An example for logical and physical streams . . . . .	28
6.3	Steps from query formulation to query execution . . . . .	30
6.4	Conceptual schema of NEXMark auction data . . . . .	32
7.1	BNF grammar excerpt for extended <code>FROM</code> clause and window specification .	35
9.1	Logical Plans for NEXMark queries . . . . .	56
9.2	Logical Plans for NEXMark queries continued . . . . .	57
10.1	Snapshot-reducibility . . . . .	59
10.2	Algebraic optimization of a logical query plan . . . . .	63
11.1	Inverse intersection . . . . .	83
11.2	Computing the difference on time intervals . . . . .	86
11.3	Incremental computation of aggregates . . . . .	91
12.1	Physical plan optimization using coalesce . . . . .	113
12.2	Physical plan with windowed subquery . . . . .	115
13.1	An architectural overview of PIPES . . . . .	120
13.2	PIPES Visualizer . . . . .	125
14.1	Total runtimes of NEXMark queries for the Time-Interval Approach (TIA) and the Positive-Negative Approach (PNA) . . . . .	132
14.2	Currency conversion query ( $Q_1$ ) . . . . .	133
14.3	Selection query ( $Q_2$ ) . . . . .	134
14.4	Memory allocation of time-based window operator for PNA . . . . .	135
14.5	Short auctions query ( $Q_3$ ) . . . . .	136
14.6	Highest bid query ( $Q_4$ ) . . . . .	137
14.7	Closing price query ( $Q_5$ ) . . . . .	138
14.8	Latency of TIA for count-based windows . . . . .	139
14.9	Adjusting SweepArea implementations to expiration patterns . . . . .	140
14.10	Priority queue sizes and output rates for a selective join . . . . .	141
14.11	Priority queue sizes and output rates for an expansive join . . . . .	142
21.1	Adaptive resource management architecture . . . . .	192

*List of Figures*

---

22.1	Model parameters . . . . .	198
22.2	Use of stream characteristics . . . . .	200
22.3	Estimation of $l'$ for the Cartesian product . . . . .	202
22.4	Estimation of $d'$ and $l'$ for the scalar aggregation (case $d < l$ ) . . . . .	204
24.1	Changing the window size . . . . .	228
24.2	Changing the time granularity . . . . .	228
24.3	Savings in memory usage . . . . .	229
24.4	Savings in processing costs . . . . .	230
24.5	Output quality . . . . .	230
24.6	Scalability of adaptation techniques and cost model . . . . .	232
28.1	Plan migration with join and duplicate elimination . . . . .	246
29.1	GenMig strategy . . . . .	251
30.1	Output stream characteristics of Parallel Track and GenMig . . . . .	259
30.2	Memory usage of Parallel Track and GenMig . . . . .	260
30.3	Performance comparison of Parallel Track, GenMig, and GenMig with reference point optimization . . . . .	261

# List of Tables

1.1	Differences between DBMSs and DSMSs . . . . .	7
8.1	Examples of logical streams . . . . .	41
8.2	Filter over logical stream $S_1$ . . . . .	42
8.3	Map over logical stream $S_1$ . . . . .	43
8.4	Union of logical streams $S_1$ and $S_2$ . . . . .	44
8.5	Cartesian product of logical streams $S_1$ and $S_2$ . . . . .	44
8.6	Duplicate elimination on logical stream $S_1$ . . . . .	45
8.7	Difference of logical streams $S_1$ and $S_2$ . . . . .	45
8.8	Grouping on logical stream $S_1$ . . . . .	46
8.9	Scalar aggregation over logical stream $S_1$ . . . . .	47
8.10	Time-based window over logical stream $S_1$ . . . . .	50
8.11	Logical stream $S_3$ . . . . .	51
8.12	Count-based window over stream $S_3$ . . . . .	51
8.13	Partitioned window over logical stream $S_3$ . . . . .	52
11.1	Example input streams . . . . .	73
11.2	Filter over physical stream $S_1$ . . . . .	74
11.3	Map over physical stream $S_1$ . . . . .	74
11.4	Union of physical streams $S_1$ and $S_2$ . . . . .	77
11.5	Overview of parameters used to tailor the ADT SweepArea to specific operators	78
11.6	Equi-join of physical streams $S_1$ and $S_2$ . . . . .	80
11.7	Duplicate elimination over physical stream $S_1$ . . . . .	82
11.8	Difference of physical streams $S_1$ and $S_2$ . . . . .	87
11.9	Scalar aggregation over physical stream $S_1$ . . . . .	93
11.10	Grouping with aggregation over physical stream $S_1$ . . . . .	97
11.11	Chronon stream $S_3$ and time-based window of size 50 time units over $S_3$ . .	98
11.12	Count-based window of size 1 over chronon stream $S_3$ . . . . .	100
11.13	Partitioned window of size 1 over chronon stream $S_3$ . . . . .	104
11.14	Split with output interval length $l_{out} = 1$ over physical stream $S_1$ . . . . .	107
11.15	Coalesce with maximum interval length $l_{max} = 20$ over physical stream $S_2$ . .	109
12.1	Impact of physical operators on order of end timestamps in their output stream and conditions under which the output stream of a physical operator is a chronon stream . . . . .	116
17.1	Time-based window over $S_3$ with window size 50 and advance 10 time units	169
17.2	Count-based window with a size and advance of 2 elements over $S_3$ . . . . .	171

*List of Tables*

---

17.3	Istream over physical stream $S_2$	173
17.4	Dstream over physical stream $S_2$	174
20.1	Granularity conversion with granule size 2 over logical stream $S_1$	186
20.2	Granularity conversion with granule size 2 over physical stream $S_1$	187
22.1	Variables specific to join costs	208
22.2	Variables specific to duplicate elimination costs	211
22.3	Variables specific to scalar aggregation costs	213
22.4	Variables specific to grouping with aggregation costs	215
22.5	Variables specific to split costs	218
22.6	Variables specific to count-based window costs	220
22.7	Variables specific to partitioned window costs	221
24.1	Plan generation	232

# List of Algorithms

1	ADT SweepArea . . . . .	69
2	Filter ( $\sigma_p$ ) . . . . .	73
3	Map ( $\mu_f$ ) . . . . .	74
4	Union ( $\cup$ ) . . . . .	75
5	TRANSFER . . . . .	76
6	Cartesian Product ( $\times$ ) / Theta-Join ( $\bowtie_\theta$ ) . . . . .	78
7	Duplicate Elimination ( $\delta$ ) . . . . .	81
8	Difference ( $-$ ) . . . . .	84
9	Scalar Aggregation ( $\alpha_{f_{agg}}$ ) . . . . .	90
10	Grouping with Aggregation ( $\gamma_{f_{group}, f_{agg}}$ ) . . . . .	95
11	UPDATE . . . . .	96
12	Time-Based Sliding Window ( $\omega_w^{\text{time}}$ ) . . . . .	97
13	Count-Based Sliding Window ( $\omega_N^{\text{count}}$ ) . . . . .	99
14	Partitioned Window ( $\omega_{f_{group}, N}^{\text{partition}}$ ) . . . . .	102
15	Split ( $\zeta_{l_{out}}$ ) . . . . .	105
16	SPLITUP . . . . .	106
17	Coalesce ( $\zeta_{l_{max}}$ ) . . . . .	108
18	Time-based Sliding Window with Advance Parameter ( $\omega_{w,a}^{\text{time}}$ ) . . . . .	169
19	Count-based Sliding Window with Advance Parameter ( $\omega_{N,A}^{\text{count}}$ ) . . . . .	170
20	UPDATESWEEPAREA . . . . .	171
21	TERMINATE . . . . .	171
22	Istream . . . . .	173
23	Dstream . . . . .	174
24	Granularity Conversion ( $\Gamma_g$ ) . . . . .	187
25	GenMig Strategy . . . . .	250
26	Fork . . . . .	252
27	Fuse . . . . .	253



# **Part I**

# **Introduction**



# 1 A New Class of Data Management Applications

The commercial success of relational *database management systems* (DBMSs) in the past decades illustrates the importance of data management in todays information technology, in particular in business applications. DBMSs are designed to evaluate complex queries over large *persistent* datasets efficiently. In order to make the data accessible to users or applications, it has to be loaded into a database first. Queries posed to the DBMS, e. g., expressed in a query language such as *SQL* [GUW00], enable the user or application to retrieve and process the stored data. For a given query, the DBMS computes the query answer over the current state of the database and returns the results. Data records remain valid in the database, unless these are explicitly removed. Usually, the number of queries exceeds the number of data manipulations, i. e., the datasets can be considered to be relatively static [GÖ03b].

Traditional DBMSs have proven to be well-suited to the organization, storage, and retrieval of finite datasets. In recent years, however, new data-intensive applications have emerged that need to process data which is continuously arriving at the system in the form of potentially unbounded, time-varying sequences of data items, termed *data streams*. Examples belonging to this new class of stream-oriented applications can be found in a diversity of application domains including sensor monitoring, finance, transaction log analysis, and network security [SCZ05, GÖ03b, BBD<sup>+</sup>02, SQR03]:

- *Sensor networks* are used in a variety of applications for monitoring physical or environmental conditions, for instance, in traffic management [MF02, CHKS03, ACG<sup>+</sup>04], location tracking and battlefield surveillance [CCC<sup>+</sup>02, SCZ05], supply chain management based on the upcoming RFID technology [WDR06, GHLK06, BTW<sup>+</sup>06], medical monitoring [BSS05, KSPL06, HS06b], and manufacturing processes [CHK<sup>+</sup>06]. The measurements generated by sensors, e. g., temperature readings, can be modeled as continuous data streams. Typical stream queries involve the detection of unusual conditions with the aim to activate alarms. In general, the complexity of these queries is beyond the evaluation of simple filter predicates checking whether certain sensor values are within pre-defined bounds. For instance, joins are required to combine data from multiple sources, while aggregation is useful to compensate for individual sensor failures.
- *Network traffic management* involves online monitoring and analysis of network packets to find out information on traffic flow patterns for routing system analysis, bandwidth usage statistics, and network security [CJSS03, SH98]. In particular, intrusion detection requires real-time responses, for instance, to prevent denial-of-service attacks. An element in a data stream typically corresponds to a network

packet and is composed of the corresponding packet header information including the packet source, destination, length, and time at which the packet header was recorded [BBD<sup>+</sup>02].

- *Electronic trading* often relies on the online analysis of financial data obtained from stock tickers and news feeds. Specific goals include discovering correlations, identifying trends, and forecasting stock prices [ZS02, GÖ03b, DGP<sup>+</sup>07]. Electronic trading requires high-volume processing of feed data with minimal latency because more current results maximize arbitrage profits [SCZ05].
- *Transaction logging* is performed in many applications, generating huge volumes of data, for example, web server logs and click streams [CFPR00], telephone call records [CJSS03, GKMS01], user account logging, firewall logs, or event logs in online auction systems such as eBay [TTPM02]. Transaction logging naturally produces data streams as sequences of log entries. Queries over these streams could be used to collect information about customer behavior, detect suspicious access patterns that could indicate fraud or attacks, or identify performance bottlenecks with the aim to improve service reliability.

This new class of emerging applications has gained in importance because of the advances in micro-sensor technologies, the increasing ubiquity of wireless computing devices, the popularity of the World Wide Web along with the steadily growing number of web applications, and the dramatic escalation in feed volumes. For example, every day WalMart records 20 million sales transactions, Google handles 70 million searches, AT&T generates 275 million call records, and eBay logs 10 million bids [DH01]. Market data feeds can generate multiple thousands of messages per second [SCZ05]. In the future, data stream volumes are likely to increase due to the steady technological progress. While storing these massive data sets is possible to a certain extent, extracting valuable information from the resultant histories is often extremely expensive because the overwhelming data volumes accumulated over months or even years cannot be searched and analyzed in acceptable time [DH01]. As a consequence, the archived data is often discarded after the retention time has elapsed, without any prior analysis but to free space for new data. Even in applications with less restrictive time requirements it is thus advisable to preprocess data streams in order to extract relevant information and reduce data volumes.

Stream-based applications necessitate queries to be evaluated over data that continuously arrives as potentially unbounded, rapid, and time-varying data streams. Because these queries remain in the system for longer times, they are called *continuous queries*. Conventional DBMSs cannot provide adequate support for the new data management needs of stream-oriented applications because they are not designed for the continual evaluation of queries over transient data. Therefore, several research groups have pursued the design and implementation of a novel system type, called *data stream management system* (DSMS).

## 1.1 Data Stream Model

We can extract the following *characteristics* of data streams and *processing requirements* from the above applications (see [BBD<sup>+</sup>02, GÖ03b, SCZ05] for further information):

- A data stream is a potentially unbounded sequence of data items generated by an active data source. A single data item is called stream element.
- Stream elements arrive continuously at the system, pushed by the active data source. The system neither has control over the order in which stream elements arrive nor over their arrival rates. Stream rates and ordering could be unpredictable and vary over time.
- A data source transmits every stream element only once. As stream elements are accessed sequentially, a stream element that arrived in the past cannot be retrieved unless it is explicitly stored. The unbounded size of a stream precludes a full materialization.
- Queries over data streams are expected to run continuously and return new results as new stream elements arrive.

The ordering of stream elements may be *implicit*, i. e., defined by the arrival time at the system, or *explicit* if stream elements provide an application timestamp indicating their generation time. Complementary to the pure stream model, some applications need to combine data streams with stored data.

DSMSs are geared to meet the demanding requirements of the data stream model and the associated stream-oriented applications. In recent years, several prototypical DSMSs have been developed by database research groups, including *Aurora* [CCC<sup>+</sup>02, ACC<sup>+</sup>03], *Borealis* [AAB<sup>+</sup>05], *CAPE* [RDS<sup>+</sup>04], *Gigascope* [CJSS03], *Nile* [HMA<sup>+</sup>04], *PIPES* [KS04], *STREAM* [MWA<sup>+</sup>03], and *TelegraphCQ* [CCD<sup>+</sup>03]. Stream processing engines have also gained recognition as commercial products. *StreamBase Systems Inc.* [Str07] is a university spin-off from M.I.T., Brown University, and Brandeis University based on findings from the Aurora and Borealis project, Gigascope is used at AT&T.

## 1.2 Differences between DBMSs and DSMSs

Let us briefly compare query processing in traditional DBMSs with the general challenges for DSMSs arising from the data stream model. Figure 1.1 illustrates query processing in DBMSs and DSMSs, while Table 1.1 summarizes the most important differences.

**Data Sources** DBMSs operate on *passive, persistent* data sources, namely, finite relations stored on disk. In contrast, DSMSs operate on *active* data sources that continuously push data into the system as possibly unbounded, rapid, and *transient* data streams. Due to the stringent response time requirements of many streaming applications, data management primarily takes place in main memory, whereas DBMSs excessively make use of external memory [Vit01]. Moreover, it is unfeasible for a DSMS to store an entire stream due to the

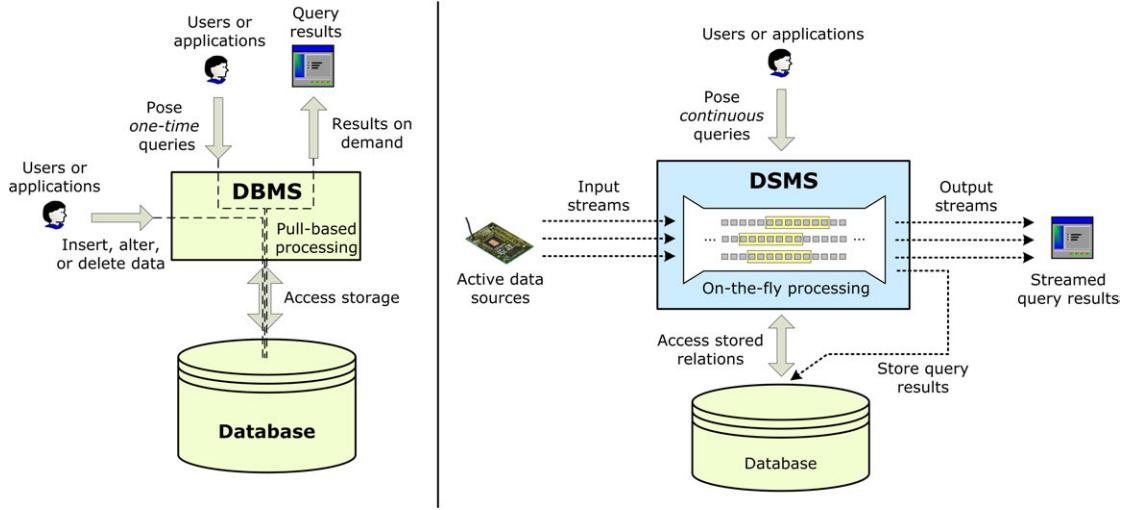


Figure 1.1: Comparison between query processing in DBMSs and DSMSs

unknown and potentially unbounded stream size. Even if larger stream fragments were written to disk, operating on this vast amount of data would drop system performance drastically and, thus, would conflict with fast response times. At most, fragments of query results may be stored in streaming applications whenever these need to support ad-hoc queries referencing the past. Access to databases is also necessary if applications need to combine relations with streams.

**Query Types** While DBMSs execute *one-time queries* over persistent data, DSMSs execute *continuous queries* over transient data. Whenever the user issues a query, the DBMS computes and outputs the results for the current snapshot of the relations. After that, the processing for this query is completed. In a DSMS, however, queries are long-running, i.e., they remain active in the system for a long period of time. Once registered at the DSMS, a query generates results continuously on arrival of new stream elements until it is deregistered.

**Query Answers** DBMSs always produce *exact* query answers for a given query, whereas continuous queries usually provide *approximate* query answers. The reasons are: (i) Many continuous queries are not computable with a finite amount of memory, e.g., the Cartesian product over two infinite streams. (ii) Some relational operators such as group-by are blocking because they must have seen the entire input before they are able to produce a result. (iii) The data can accumulate faster than the system can process the data. In general, high quality approximate answers are acceptable for users or applications. Moreover, recently arrived data is considered more accurate and useful.

**Processing Methodology** A DBMS employs a *demand-driven* computation model, where processing is initiated when a query is issued. Tuples are typically read from

Aspect	DBMS	DSMS
Data Sources	Passive, finite relations stored on disk	Active data sources generating continuous, potentially unbounded, transient data streams
Query Types	One-time queries evaluated over the current state of the database	Long-running queries that continuously produce results when new stream elements arrive or time progresses
Query Answer	Exact	Often only approximate
Processing Methodology	Pull-based processing initiated on demand; random access	Push-based processing initiated by arriving stream elements; sequential access
Query Optimization	Static optimization prior to query execution	Static optimization plus re-optimizations at runtime for adaptation purposes

Table 1.1: Differences between DBMSs and DSMSs

relations in a *pull-based* manner using scan-based or index-based access methods [Gra93]. Conversely, query processing in a DSMS is *data-driven* because the query answer is computed incrementally on arrival of new stream elements. Hence, the underlying active data sources trigger processing in a *push-based* fashion. Without explicit buffering, DSMSs have to access stream elements sequentially in arrival order, whereas DBMSs have random access to tuples.

**Query Optimization** DBMSs optimize queries prior to execution. First, the optimizer generates a set of semantically equivalent query plans but with different performance characteristics. Based on a cost model incorporating metadata about the underlying data and system conditions, the optimizer then selects the plan with the lowest estimated costs. While this *static optimization* is adequate for one-time queries, the long-running nature of continuous queries entails DSMSs requiring further *query optimizations at runtime* to adapt to changing stream characteristics and system conditions. Data distributions and arrival rates of streams and also query workload may vary over time. Without runtime adaptations, plan and system performance may degrade significantly during the lifetime of a continuous query.

Despite all the differences, one might endeavor to augment traditional DBMSs with specific functionality to support continuous queries over data streams [ACG<sup>+</sup>04, SCZ05]. Encoding triggers [WC96, PD99, HCH<sup>+</sup>99] or employing existing techniques for data expiration [Tom03, SJS06] and incremental maintenance of materialized views [BLT86, JMS95, TGNO92] tend to be promising starting points. However, the downside of this augmentation approach is twofold. First, current DBMSs do not scale for a large number of triggers [ACG<sup>+</sup>04] and, second, query optimization would be limited severely because the

system has no means to assess the novel functionality outside the DBMS core [ACC<sup>+</sup>03].

The remainder of this introductory part consists of three chapters. In Chapter 2, we outline research challenges relevant to the construction of a fully fledged DSMS. The specific contributions of this thesis are highlighted in Chapter 3. Finally, Chapter 4 provides an overview of the organization of this thesis.

## 2 Research Challenges

Processing continuous queries over data streams raises various research challenges. While this chapter briefly summarizes important challenges for building a DSMS, the interested reader is referred to [BBD<sup>+</sup>02, GÖ03b, SCZ05] for excellent surveys of models and issues in data stream management.

### 2.1 Query Formulation

Today, many applications dealing with continuous queries over data streams rely on hand-coded queries to directly process elements at delivery. Custom coding typically entails high development and maintenance costs and is also inflexible with regard to enhancements. The latter implies that changing the application logic is difficult and often necessitates extensive testing and debugging. It is nearly impossible to modify queries at runtime. Rather than expressing queries in custom code or scripts, the use of a high-level query language would facilitate the development and maintenance of complex applications drastically. A *declarative query language* such as the practically approved SQL would permit the formulation and modification of application logic in an intuitive and concise manner by expressing and altering query statements. However, SQL is meant for one-time queries over relations and not for continuous queries over data streams. For this reason, SQL has to be enhanced appropriately to support the new stream data type along with stream operations. This involves identifying what kind of new functionality is required in SQL and extending the grammar specification accordingly. Nevertheless, changes to the SQL standards should be minimal so that the enriched language is still attractive for people with SQL experience. Alternatively, query formulation could be based on a graphical user interface (GUI) allowing users to specify data flow by combining operators from a stream algebra in a *procedural* manner.

### 2.2 Semantics for Continuous Queries

Query formulation only makes sense if the query semantics is precisely defined. To guarantee predictable and repeatable query results, continuous query semantics should be defined independent of how the system operates internally. Unlike traditional one-time queries, continuous queries produce output over time. Therefore, it is important to incorporate the *notion of time* into the semantics. Since queries expressed in some query language are translated into a query plan composed of operators from a *stream algebra*, the semantics of a query results from the semantics of the contributing operators. Hence, a suitable set of stream operators has to be identified and defined first. This raises the question to which extent the stream algebra can profit from relational operators and

their well-known semantics. However, enhancing relational operators with temporal support can be a complex task as known from temporal databases. Furthermore, temporal extensions alone are not sufficient for stream processing because queries are long-running and data streams can be unbounded. As some queries would require an unbounded amount of memory to evaluate precisely, the stream algebra must support *novel constructs* to restrict the range of queries to finite subsets over the input streams. It has to be clarified (i) how these novel constructs can be integrated into the algebra in a coherent manner, (ii) how they affect query semantics, and (iii) what algebraic optimizations the stream algebra allows.

## 2.3 Stream Algorithms

The data stream model introduces new challenges for the implementation of queries. First, algorithms no longer have random access to their input, only *sequential* access. Second, algorithms that need to store some state information from stream elements that have been seen previously, e. g., the join and aggregation, must be computable within a *limited amount of space*, while the streams themselves are unbounded. This necessitates approximation techniques that trade output accuracy for memory usage and opens up the question of which reliable guarantees can be given for the output. Third, some implementations of relational operators are blocking. Since streams may be infinite, blocking operators cannot be applied in the stream computation model because they would produce no output. Examples are the difference that computes results by subtracting the second input from the first input, or the aggregation with SUM, COUNT, MAX, etc. Because queries with blocking operators, in particular with aggregation, are extremely common in applications, *nonblocking* analogs have to be found that still produce sound results. Fourth, algorithms should process incoming elements *on-the-fly* and generate output continuously over time. This implies that amortized processing time per stream element should be kept small.

## 2.4 Adaptive Query Execution

Due to the long-running nature of queries, DSMSs have to be designed to execute thousands of continuous queries concurrently. Moreover, a DSMS must be able to adapt to its highly dynamic and unpredictable environment. The fundamental architecture of DSMSs should regard the following objectives. First, it should provide mechanisms to reduce load in cases of saturation to guarantee *high-availability*. Not only increasing query workload but also bursts in stream rates may cause a system to become overloaded. Second, although users in many stream applications tolerate approximate answers, the system should attempt to *maximize output accuracy*. Due to changing system conditions, resources have to be redistributed among queries and operators at runtime. Third, the system should be able to exploit *multi-query optimization* and *optimization at runtime* to save system resources and improve scalability. It should provide means for sharing the execution of queries with common subexpressions as well as strategies for gradually replacing an inefficient plan with a more efficient plan at runtime without stalling query execution. Fourth, the system should provide *near real-time capabilities* to meet the

response time requirements of applications. Overall, the objectives necessitate the system to monitor its performance continually. Based on these runtime statistics, appropriate adaptation strategies have to be developed.



## 3 Contributions

This chapter points out the main contributions of this thesis and places them in the context of the research challenges introduced in the previous chapter. As part of the research for this thesis, we have designed and developed a general-purpose software infrastructure for data stream management called *PIPES* (*Public Infrastructure for Processing and Exploring Streams*) [KS04]. This thesis describes the salient features of PIPES, including the query language, its semantic foundations, algorithms and implementation concepts, and the adaptive runtime environment.

As already mentioned, it is not always possible to compute exact answers for continuous queries because streams may be unbounded whereas system resources are limited. Since high-quality approximate answers are acceptable in lieu of exact answers in the majority of applications, we employ *sliding windows* as approximation technique. By imposing this technique on data streams, the range of a continuous queries is restricted to finite sliding windows capturing the most recent data from the streams, rather than the entire past history. We choose the sliding window approach because of the following advantages: (i) it emphasizes recent data, which in most applications is considered to be more important and relevant than older data, and (ii) query semantics can be defined precisely so that queries produce deterministic answers.

### Query Language

This thesis presents our declarative query language to express continuous queries over data streams. Because our language inherits the basic syntax from SQL, it is familiar and attractive for users with SQL experience. We show that slight modifications to SQL standards are sufficient to incorporate data streams and stream-oriented operations. For the specification of sliding windows, we reuse and enhance a subset of the window constructs definable in SQL:2003 for OLAP functions. Our query language even allows users to define and include derived streams as well as complex nested queries with aggregates or quantifiers.

### Query Semantics

In analogy to traditional database systems, we distinguish between a logical and physical operator algebra. This thesis precisely defines our semantics for continuous queries by means of a *logical stream algebra* modeling data streams as temporal multisets. The operator algebra contains a stream-counterpart for every operator in the extended relational algebra, except for sorting. In addition, it provides novel window operators to define the scope of operations. Rather than integrating windows directly into operators, we separated the functionalities to avoid redundancy and facilitate the exchange of window types. The combination of window operators with our stream variants of the relational operators creates their windowed analogs. Overall, our logical algebra assigns an exact meaning to any continuous query at any point in time. As our logical algebra reveals the temporal

properties of stream operations, it represents a valuable tool for exploring and validating equivalences.

#### **Stream Algorithms**

The implementation of continuous queries is specified by our unique *physical stream algebra*. This operator algebra consists of push-based, nonblocking, stream-to-stream algorithms. Stream elements are modeled as tuples tagged with time intervals. The time interval indicates the validity of a tuple according to the specified windows. We describe how physical operators process stream elements, in particular, how they modify the time intervals, to produce semantically equivalent results to their logical counterparts. We propose and employ specific data structures for operator state maintenance that allow for efficient probing and eviction of stream elements.

#### **Plan Generation and Query Optimization**

This dissertation not only outlines how queries expressed in our query language are compiled into logical query plans, but also shows how logical plans are mapped to physical query plans. Moreover, we prove that the well-known transformation rules from relational databases are equally applicable in our stream algebra. We enrich this extensive set of *algebraic equivalences* with novel rules for the window constructs. Besides these logical optimizations, we investigate several *physical optimizations* dealing with expiration patterns, stream ordering, and time granularity. We identify vital properties of subplans that enable the system to apply these optimizations. Altogether, our approach establishes a solid and powerful foundation for the optimization of continuous queries.

#### **Adaptation Techniques**

This thesis investigates an approach to *adaptive resource management* that adjusts window sizes and time granularities to keep system resource usage within bounds. When posing a continuous query, we enable the user to specify quality of service constraints to restrict window sizes and time granularities to tolerated ranges. Because the resource manager has to adhere to these constraints, we can ensure that the user receives query answers with at least the minimal required accuracy. These two novel techniques differ from standard load shedding approaches based on sampling because we can give strong semantical guarantees on the query results, even under query re-optimization. We explain the semantics of both adaptation techniques, discuss their impact on query plans, and validate their effectiveness and performance in experimental studies.

#### **Cost Model**

This dissertation presents an extensive cost model for our physical algebra that allows for *estimation of resource consumption* at operator-, query-, and system-level based on *stream characteristics*. Such a cost model is crucial to any adaptive runtime component in a DSMS because it can be used to quantify the effects of changes to query plans on the resource usage in advance. On the one hand, our resource manager employs the cost model to quantify the effects of adaptation techniques on query plans in order to determine to which extent window and granularity sizes need to be adjusted to keep resource usage in bounds. On the other hand, our query optimizer uses the cost model to select the best execution plan, namely the plan with the lowest estimated cost, from a set of semantically

---

equivalent plans. Furthermore, the cost model enables the system to decide whether sufficient resources are available to accept a new query or not.

### Plan Migration

In this thesis, we develop a general strategy for the dynamic plan migration problem tackling re-optimizations of query plans at runtime. During plan migration, the old plan, which has become inefficient over time due to changes in the data characteristics, is replaced with a new, more efficient plan. Our migration strategy treats the old and new plan as *black boxes* that have to produce semantically equivalent outputs. We not only show that our approach overcomes the limitations of existing solutions and prove its correctness, but also analyze its performance. Our experiments confirm that our migration strategy exhibits lower memory and CPU overhead and finishes migration earlier than comparative approaches.

### Query Execution

The principle design, operational functionality, and architecture of PIPES originate from the findings and methods developed in this thesis. With the development of PIPES, we pursue a library approach rather than building a monolithic system because we are convinced that it is almost unfeasible to develop a general-purpose and still highly efficient DSMS for the plethora of streaming applications. Instead, PIPES provides powerful and generic building blocks for the construction of a fully functional DSMS which can be tailored to the specific needs of an application domain. This thesis describes the multi-threading enabled architecture of PIPES, explains how PIPES supports sharing of computation across multiple query plans, and outlines the frameworks for the adaptive runtime components along with their features.

### Publications

The main results of the thesis have been published in refereed international journals, conferences, and workshops.

- Michael Cammert, Christoph Heinz, Jürgen Krämer, Martin Schneider, and Bernhard Seeger. A Status Report on XXL – a Software Infrastructure for Efficient Query Processing. *IEEE Data Engineering Bulletin*, 26(2):12–18, 2003.
- Michael Cammert, Christoph Heinz, Jürgen Krämer, and Bernhard Seeger. Datenströme im Kontext des Verkehrsmanagements. In *Mobilität und Informationssysteme, Workshop des GI-Arbeitskreises „Mobile Datenbanken und Informationssysteme“*, 2003.
- Jürgen Krämer and Bernhard Seeger. PIPES - A Public Infrastructure for Processing and Exploring Streams. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 925–926, 2004.
- Michael Cammert, Christoph Heinz, Jürgen Krämer, and Bernhard Seeger. Anfrageverarbeitung auf Datenströmen. *Datenbank Spektrum*, 11:5–13, 2004.
- Jürgen Krämer and Bernhard Seeger. A Temporal Foundation for Continuous Queries over Data Streams. In *Proceedings of the International Conference on Management of Data (COMAD)*, pages 70–82, 2005.

### 3 Contributions

---

- Michael Cammert, Christoph Heinz, Jürgen Krämer, and Bernhard Seeger. Sortierbasierte Joins über Datenströmen. In *Proceedings of the Conference on Database Systems for Business, Technology, and the Web (BTW)*, pages 365–384, 2005.
- Christoph Heinz, Jürgen Krämer, Bernhard Seeger, and Alexander Zeiss. Datenstromverarbeitung in Production Intelligence Software. In *Workshop der GI-Fachgruppe „Datenbanken“ zum Thema „Business Intelligence“*, 2005.
- Jürgen Krämer, Bernhard Seeger, Thomas Penzel, and Richard Lenz. *PIPES<sub>med</sub>*: Ein flexibles Werkzeug zur Verarbeitung kontinuierlicher Datenströme in der Medizin. In *51. Jahrestagung der Deutschen Gesellschaft für Medizinische Informatik, Biometrie und Epidemiologie (GMDS)*, 2006.
- Michael Cammert, Christoph Heinz, Jürgen Krämer, Tobias Riemenschneider, Maxim Schwarzkopf, Bernhard Seeger, and Alexander Zeiss. Stream Processing in Production-to-Business Software. In *Proceedings of the IEEE International Conference on Data Engineering (ICDE)*, pages 168–169, 2006.
- Michael Cammert, Jürgen Krämer, Bernhard Seeger, and Sonny Vaupel. An Approach to Adaptive Memory Management in Data Stream Systems. In *Proceedings of the IEEE International Conference on Data Engineering (ICDE)*, pages 137–139, 2006.
- Jürgen Krämer, Yin Yang, Michael Cammert, Bernhard Seeger, and Dimitris Papadias. Dynamic Plan Migration for Snapshot-Equivalent Continuous Queries in Data Stream Systems. In *Proceedings of the International Conference on Extending Data Base Technology (EDBT) Workshops*, pages 497–516, 2006.
- Michael Cammert, Jürgen Krämer, and Bernhard Seeger. Dynamic Metadata Management for Scalable Stream Processing Systems. In *Proceedings of the First International Workshop on Scalable Stream Processing Systems (SSPS)*, 2007. (Co-located with IEEE ICDE).
- Michael Cammert, Christoph Heinz, Jürgen Krämer, Bernhard Seeger, Sonny Vaupel, and Udo Wolske. Flexible Multi-Threaded Scheduling for Continuous Queries over Data Streams. In *Proceedings of the First International Workshop on Scalable Stream Processing Systems (SSPS)*, 2007. (Co-located with IEEE ICDE).
- Yin Yang, Jürgen Krämer, Dimitris Papadias, and Bernhard Seeger. HybMig: A Hybrid Approach to Dynamic Plan Migration for Continuous Queries. *IEEE Transactions on Knowledge and Data Engineering (TKDE)*, 19(3):398–411, 2007.
- Michael Cammert, Jürgen Krämer, Bernhard Seeger, and Sonny Vaupel. A Cost-Based Approach to Adaptive Resource Management in Data Stream Systems. *IEEE Transactions on Knowledge and Data Engineering (TKDE)*, 2007.

The implementation of our stream processing infrastructure PIPES is freely available under the terms of the GNU LGPL license at the project website [XXL07].

## 4 Thesis Outline

In this first part of the thesis, Part I, we introduced the problem of data management for a new class of data-intensive applications that require to process data from potentially unbounded, transient streams. We showed that continuous queries necessitate a new processing methodology for which traditional DBMSs turn out to be inadequate. We then discussed open issues in state of the art continuous query processing and gave an overview of the main contributions of this thesis. The remainder of this dissertation consists of the following four parts:

- Part II establishes a solid foundation for continuous query processing over data streams and represents the core of the thesis. The structure reflects the series of tasks carried out in a DSMS, ranging from query translation to query execution. We define our logical and physical data stream models and the respective operator algebras, specify our SQL-style query language and plan generation, derive transformation rules from algebraic equivalences, point out novel physical optimizations, and outline the architecture of our stream processing infrastructure PIPES.
- Part III deals with adaptive resource management. We propose two methods of adaptations: adjusting the window sizes or time granularities. For both methods, we clarify the semantics, implementation, and impact on the resource consumption of query plans. Furthermore, we develop an extensive cost model for estimating operator resource utilization. Based on this cost model and a heuristic strategy for applying our adaptation techniques, our resource manager keeps system resources within bounds.
- Part IV focuses on the problem of query re-optimizations at runtime. We first formulate general objectives for plan migration strategies. We then demonstrate the deficiencies of existing migration strategies. After that, we present our novel plan migration strategy, prove its correctness, and analyze its runtime costs.
- Part V concludes the thesis by giving a short summary of the main contributions along with an outlook on future research directions.

Notice that each of the three intermediate parts (i) starts with a more focused introduction to the particular subject, including a detailed summary of contributions, (ii) contains a thorough discussion of related work, and (iii) ends with a chapter drawing conclusions.



## **Part II**

# **Query Semantics and Implementation**



## 5 Introduction

Continuous queries over unbounded data streams have emerged as an important query type in a variety of applications, e. g., financial analysis, network and traffic monitoring, sensor networks, and complex event processing [BBD<sup>+</sup>02, GÖ03b, CCC<sup>+</sup>02, SH98, CJSS03, WZL03, DGP<sup>+</sup>07]. Traditional database management systems are not designed to provide efficient support for the continuous queries posed in these data-intensive applications [BBD<sup>+</sup>02]. For this reason, novel techniques and systems dedicated to the challenging requirements in stream processing have been developed. There has been a great deal of work on system-related topics such as adaptive resource management [BSW04, TCZ<sup>+</sup>03, CKSV06], scheduling [BBD<sup>+</sup>04, CCZ<sup>+</sup>03, CHK<sup>+</sup>07], query optimization [VN02, AN04, ZRH04, KYC<sup>+</sup>06], and on individual stream operations, e. g., user-defined aggregates [LWZ04], windowed stream joins [KNV03, GÖ03a], and windowed aggregation [YW01, LMT<sup>+</sup>05]. However, an operative data stream management system needs to unify all this functionality. This constitutes a serious problem because most of the approaches rely on different semantics which makes it hard to merge them. The problem gets even worse if the underlying semantics is not specified properly but only motivated through illustrative examples, for instance, in some informal, declarative query language. Moreover, such queries tend to be simple and the semantics of more complex queries often remains unclear.

To develop a complete DSMS, it is crucial to identify and define a basic set of operators to formulate continuous queries. The resulting stream operator algebra must have a precisely-defined and reasonable semantics so that at any point in time, the query result is clear and unambiguous. The algebra must be expressive enough to support a wide range of streaming applications, e. g., the ones sketched in [SQR03]. Therefore, the algebra should support windowing constructs and stream analogs of the relational operators. As DSMSs are designed to run thousands of continuous queries concurrently, a precise semantics is also required to enable subquery sharing in order to improve system scalability, i. e., common subplans are shared and thus computed only once [CDTW00]. Furthermore, it has to be clarified whether query optimization is possible and to what extent. The relevance of this research topic is also confirmed in the survey [BBD<sup>+</sup>02]:

...perhaps the most interesting open question is that of defining extensions of relational operators to handle stream constructs, and to study the resulting “stream algebra” and other properties of these extensions. Such a foundation is surely key to developing a general-purpose well-understood query processor for data streams.

Besides the semantic aspects, it is equally important for realizing a DSMS to have an efficient implementation that is consistent with the semantics. To date little work has been published that combines semantic findings with execution details in a transparent

manner. Therefore, Part II specifies our general-purpose stream algebra not only from the logical but also from the physical perspective. In addition, it reveals how to adapt the processing steps for one-time queries, common in conventional DBMSs, to continuous queries.

The notion of time plays an important role in the majority of streaming applications [BBD<sup>+</sup>02, GÖ03b, SQR03]. Usually, stream tuples are tagged with a timestamp. Any stream algebra should consequently incorporate operators that exploit the additionally available temporal information. Extending the relational algebra towards a temporal algebra over data streams is non-trivial for a number of reasons, including the following.

- **Timestamps:** It is not apparent how timestamps are assigned to the operator results. Assume we want to compute a join over two streams whose elements are integer-timestamp pairs. Is a join result tagged with the minimum or maximum timestamp of the qualifying elements, or both? What happens in the case of cascading joins? While some approaches prefer to choose a single timestamp, e.g., [BBDM03, GHM<sup>+</sup>07], others suggest keeping all timestamps to preserve the full information [GÖ03a, ZRH04]. To the best of our knowledge, currently there exists no consensus.
- **Resource Limitations:** Due to the unbounded stream size, the system can run out of resources if it executes long-running queries whose evaluation plans involve stateful operators. Stateful operators like the join or duplicate elimination need to store past elements to produce correct results. DSMSs apply windowing and approximation techniques in order to restrict the amount of resources required for evaluating a query [ABB<sup>+</sup>02, GÖ03b]. For this reason, it is essential to analyze the use and impact of these constructs on logical and physical query plans. Be aware that things can become complex quickly, especially if continuous queries include subqueries that are possibly windowed.
- **Asynchronous Inputs:** Streams arriving at the DSMS are typically not synchronized with each other because elements are generated by remote data sources and the network conveying the elements may not guarantee in-order transmission [SW04]. Independent of the arrival order across data sources, query results should be deterministic and equivalent in some way. Hence, an appropriate definition for plan equivalence must be found, which, at best, is invariant under re-optimizations at runtime [ZRH04, KYC<sup>+</sup>06].
- **Language Extensions:** In DBMSs it is common to formulate queries via SQL rather than directly composing query plans at operator-level. However, the novel constructs specific to stream processing necessitate extensions to SQL. An appropriate syntax with an intuitive meaning plus the language grammar modifications have to be identified. Overall, queries should be compact and easy to write.

While all operators of our logical algebra can handle infinite streams, those operators of the physical algebra keeping state information usually cannot produce exact answers for unbounded input streams with a finite amount of memory [ABB<sup>+</sup>02]. A prevalent technique that overcomes this problem is to restrict the operator range to a finite,

---

*sliding window* over each input stream. As a result, the most recent data is emphasized, which is viewed to be more relevant than the older data in the majority of real-world applications [BBD<sup>+</sup>02, GÖ03b]. Moreover, the query answer is exact with regard to the window specification. Another class of techniques, differing from the approach being taken in this thesis, aims at maintaining compact *synopses* as approximations of the entire operator state [BBD<sup>+</sup>02, GÖ03b]. Synopsis construction can be based on random sampling [AGPR99, CMN99, Vit85], sketching techniques [AMS96], histograms [IP99], and wavelets [CGRS00, GKMS01, HS05]. Using those approximations instead of the full state information leads to approximate query answers. Unfortunately, it is often not possible to give reliable guarantees for the results of approximate queries, in particular for sampling-based methods, which prevents query optimization [CMN99]. Approaches based on synopses are altogether complementary to the work presented in this thesis.

As only a few papers published so far address the design and implementation issues of a general-purpose algebra for continuous query processing, we focus on continuous sliding window queries, their semantics, implementation, and optimization potential in this work. To summarize the contributions of Part II:

- We define a concrete logical operator algebra for data streams with precise query semantics. The semantics are derived from the well-known semantics of the extended relational algebra [DGK82, GUW00] and its temporal enhancement [SJS01]. To the best of our knowledge, such a semantic representation over multisets does not exist, neither in the stream community nor in the field of temporal databases.
- We discuss the basic implementation concepts and algorithms of our physical operator algebra. Our novel stream-to-stream operators are designed for push-based, incremental data processing. Our approach is the first that uses time intervals to represent the validity of stream elements according to the window specifications in a query. We show how time intervals are initialized and adjusted by the individual operators to produce correct query results. Although it was not possible to apply research results from temporal databases to stream processing directly, our approach profits from the findings in [SJS00, SJS01].
- We illustrate query formulation, reveal plan generation, and outline the core functionality of the runtime components operational in our stream processing infrastructure PIPES [KS04, CHK<sup>+</sup>06]. Altogether, Part II surveys our practical approach and makes the successive tasks from query formulation to query execution seamlessly transparent – an aspect often missing in papers. Moreover, we discuss the expressive power of our approach and compare it to the *Continuous Query Language (CQL)*.
- We adapted the temporal concept of snapshot-equivalence for the definition of equivalences at data stream and query plan level. These equivalences do not only validate the concordance between our physical and logical operator algebra, but also establish a solid basis for logical and physical query optimization as we were able to carry over most transformation rules from conventional and temporal databases to stream processing.

- A thorough discussion, along with experimental studies, confirms the superiority of our unique time-interval approach over the positive-negative approach [ABW06, GHM<sup>+</sup>07, AAB<sup>+</sup>05] for time-based sliding window queries, the predominant type of continuous queries.

This part is structured as follows. Chapter 6 gives important definitions and introduces our running example. Chapter 7 explains how to formulate continuous queries. The logical operator algebra defined in Chapter 8 reveals our continuous query semantics. While Chapter 9 discusses logical plan generation, Chapter 10 identifies algebraic equivalences and describes their applicability. Our physical operator algebra is introduced in Chapter 11, followed by Chapter 12 dealing with physical plan generation and query optimization. Chapter 13 briefly outlines the overall architecture and functionality of PIPES. We compare our time-interval approach with the positive-negative approach in Chapter 14. Chapter 15 substantiates that our query language is at least as expressive as CQL. Related work is surveyed in Chapter 16. Chapter 17 presents useful enhancements of our approach. Finally, Chapter 18 summarizes the major contributions of this part.

# 6 Preliminaries

This chapter defines a formal model of data streams. Sections 6.1 and 6.2 first characterize the terms *time* and *tuple* respectively. Section 6.3 then formalizes our different types of streams, namely, *raw streams*, *logical streams*, and *physical streams*. After that, Section 6.4 defines equivalence relations for streams. Stream transformations are presented in Section 6.5. Section 6.6 briefly outlines the processes involved in optimizing and evaluating a continuous query. Eventually, we introduce our running example in Section 6.7.

## 6.1 Time

Let  $\mathbb{T} = (T, \leq)$  be a discrete time domain with a total order  $\leq$  as proposed in [BDE<sup>+</sup>97]. A *time instant* is any value from  $T$ . We use  $\mathbb{T}$  to model the notion of *application time*, not system time. Note that our semantics and the corresponding implementation only require any discrete, ordered domain. For the sake of simplicity, let  $T$  be the non-negative integers  $\{0, 1, 2, 3, \dots\}$ .

## 6.2 Tuples

We use the term *tuple* to describe the *data portion* of a stream element. In a mathematical sense, a tuple is a finite sequence of objects, each of a specified type. Let  $\mathcal{T}$  be the composite type of a tuple. Let  $\Omega_{\mathcal{T}}$  be the set of all possible tuples of type  $\mathcal{T}$ . This general definition allows a *tuple* to be a relational tuple, an event, a record of sensor data, etc. We do not want to restrict our stream algebra to the relational model as done in [ABW06] because our operations are parameterized with functions and predicates that can be tailored to arbitrary types. For the case of the relational model, the type would be a relational schema and the tuples would be relational tuples over that schema. As we want to show that our approach is expressive enough to cover CQL, we point out how our operations can be adapted to the relational model whenever necessary.

## 6.3 Stream Representations

Throughout the thesis, we use different representations for data streams. *Raw streams* denote input streams registered at the system. Hence, these streams provide the system with data. In our internal representation, we use different stream formats. We distinguish between a logical and a physical algebra in analogy to traditional DBMSs. The logical operator algebra relies on *logical streams*, whereas its physical counterpart is based on *physical streams*. Figure 6.1 gives an overview of the different stream representations and the formats of stream elements.

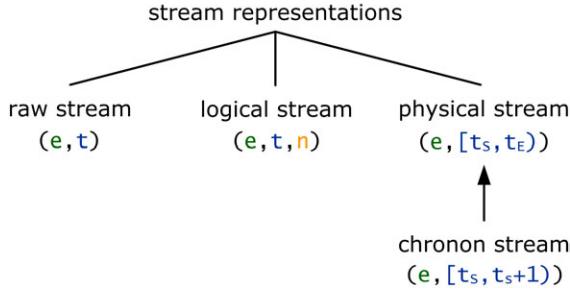


Figure 6.1: Overview of different stream representations

### 6.3.1 Raw Streams

*Raw streams* constitute external streams provided to our DSMS. In the majority of real-world streams, tuples are tagged with a timestamp [GÖ03b, ABW06, ACC<sup>+</sup>03, ACG<sup>+</sup>04, SQR03].

**Definition 6.1** (Raw Stream). A *raw stream*  $S^r$  of type  $\mathcal{T}$  is a potentially infinite sequence of elements  $(e, t)$ , where  $e \in \Omega_{\mathcal{T}}$  is a *tuple* of type  $\mathcal{T}$  and  $t \in T$  is the associated *timestamp*. A raw stream is non-decreasingly ordered by timestamps.

A stream element  $(e, t)$  can be viewed as an event, i. e., as an instantaneous fact capturing information that occurred at time instant  $t$  [JCE<sup>+</sup>94]. A measurement obtained from a sensor is an example for such an event. Note that the timestamp is not part of the tuple, and hence does not contribute to the stream type. There can be zero or multiple tuples with the same timestamp in a raw stream. We only require the number of tuples having the same timestamp to be finite. This weak condition generally holds for real-world data streams. In the following, let  $\mathbb{S}_{\mathcal{T}}^r$  be the set of all raw streams of type  $\mathcal{T}$ .

### Base Streams and Derived Streams

Raw streams can be converted to logical and physical streams using the transformations given in Section 6.5. We use the terms *base streams* or *source streams* for the resultant streams because these streams represent the actual inputs of the logical and physical plans respectively. In contrast, we term the output streams produced by logical and physical operators *derived streams*.

### System Timestamps

If the tuples arriving at the system are not equipped with a timestamp, the system assigns a timestamp to each tuple by default using the system's local clock. While this process generates a raw stream with system timestamps that can be processed like a regular raw stream with application timestamps, the user should be aware that application time and system time are not necessarily synchronized.

### 6.3.2 Logical Streams

A *logical stream* is the order-agnostic multiset representation of a raw or physical stream used in our logical algebra. It shows the validity of tuples at time-instant level. Logical and physical streams are sensitive to duplicates, i. e., tuples valid at the same point in time, because (i) raw streams may already deliver duplicates, and (ii) some operators like projection or join may produce duplicates during runtime, even if all elements in the input streams are unique.

**Definition 6.2** (Logical Stream). A *logical stream*  $S^l$  of type  $\mathcal{T}$  is a potentially infinite multiset (*bag*) of elements  $(e, t, n)$ , where  $e \in \Omega_{\mathcal{T}}$  is a *tuple* of type  $\mathcal{T}$ ,  $t \in T$  is the associated *timestamp*, and  $n \in \mathbb{N}$ ,  $n > 0$ , denotes the *multiplicity* of the tuple.

A stream element has the following meaning: tuple  $e$  is valid at time instant  $t$  and occurs  $n$  times at this instant. A logical stream  $S^l$  satisfies the following condition:

$$\forall (e, t, n), (\hat{e}, \hat{t}, \hat{n}) \in S^l. (e = \hat{e} \wedge t = \hat{t}) \Rightarrow (n = \hat{n}).$$

The condition prevents a logical stream from containing multiple elements with identical tuple and timestamp components. Let  $\mathbb{S}_{\mathcal{T}}^l$  be the set of all logical streams of type  $\mathcal{T}$ .

### Relational Multisets

We enhanced the well-known multiset notation for the extended relational algebra proposed by Dayal et al. [DGK82] for logical streams by incorporating the notion of time. We use this novel representation to convey our query semantics in an elegant yet concrete way. The snapshot perspective facilitates the understanding of the temporal properties of our stream operators.

### Abstraction

At the logical level, dealing with potentially infinite streams does not constitute a problem. The logical stream representation assumes all tuples, timestamps, and multiplicities to be available. Hence, our logical algebra and logical streams provide an abstraction from physical-level issues such as blocking operators, out-of-order element delivery, and streams not being coordinated with each other. Since our query semantics is not affected by these physical-level issues, such an abstraction is valuable for (i) presenting the meaning of a continuous query in a clear and precise fashion and (ii) discovering and reasoning about equivalences as a rationale for query optimization.

### 6.3.3 Physical Streams

Operators in the physical algebra process so-called *physical streams*. Instead of sending positive and negative tuples through a query pipeline as done in [ABW06, GHM<sup>+</sup>07], we propose a novel approach that assigns a time interval to every tuple representing its validity. To the best of our knowledge, we are the first in the stream community to pursue such an approach [KS05]. Conceptually, a physical stream can be viewed as a

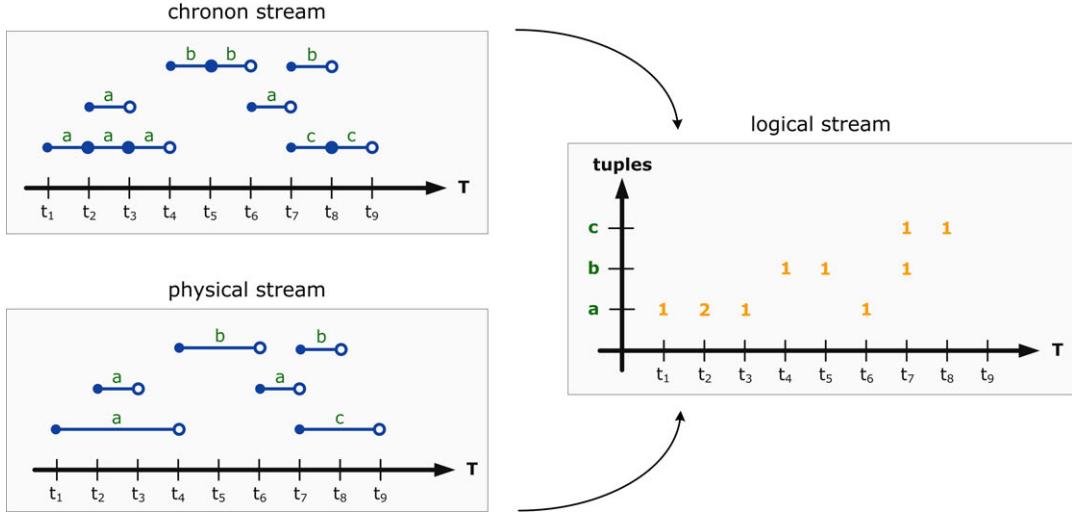


Figure 6.2: An example for logical and physical streams

more compact representation of its logical counterpart that (i) coalesces identical tuples with consecutive timestamps into a single tuple with a time interval and (ii) brings the resulting stream elements into ascending order by start timestamps.

**Definition 6.3** (Physical Stream). A *physical stream*  $S^p$  of type  $\mathcal{T}$  is a potentially infinite, ordered multiset of elements  $(e, [t_S, t_E])$ , where  $e \in \Omega_{\mathcal{T}}$  is a *tuple* of type  $\mathcal{T}$ , and  $[t_S, t_E]$  is a half-open *time interval* with  $t_S, t_E \in T$ . A physical stream is non-decreasingly ordered by start timestamps.

The meaning is as follows: tuple  $e$  is valid during the time period given by  $[t_S, t_E]$ , where  $t_S$  denotes the start and  $t_E$  the end timestamp. According to our definition, a physical stream may contain duplicate elements, and the ordering of elements is significant. Note that we do not enforce any order for stream elements with identical start timestamps. Let  $\mathbb{S}_{\mathcal{T}}^p$  be the set of all physical streams of type  $\mathcal{T}$ .

### Chronon Streams

**Definition 6.4** (Chronon Stream). A *chronon stream* of type  $\mathcal{T}$  is a specific physical stream of the same type whose time intervals are *chronons*, i. e., non-decomposable time intervals of fixed, minimal duration determined by the time domain  $\mathbb{T}$ .<sup>1</sup>

Throughout this thesis, the terms *time instant* or simply *instant* denote single points in time, whereas the terms *chronon* and *time unit* denote the time period between consecutive points in time (at finest time granularity).

**Example 6.1.** Figure 6.2 gives an example of two physical streams that map to the same logical stream. The physical stream at the top is a chronon stream. The x-axis shows the

<sup>1</sup>see [JCE<sup>+</sup>94] for the definition of the term *chronon*

time line. The letters indicate tuples. The numbers in the drawing of the logical stream denote the multiplicity of the corresponding tuples shown on the y-axis.

### Time Granularity

Time intervals model the validity independent from the granularity of time. In Part II, query semantics and thus query answers refer to the finest time granularity available in the system, determined by the time domain  $\mathbb{T}$ . However, sometimes the user is not interested in query answers at finest time granularity, e. g., the average temperature of a sensor within the last minute at millisecond granularity. For this purpose, we proposed a novel operator to change the time granularity of streams while maintaining sound query semantics [CKSV06]. The impact of the granularity conversion on our algorithms is discussed in Part III.

## 6.4 Value-Equivalence

*Value-equivalence* is a property that characterizes two stream elements independent of the stream representation. Informally, we denote two stream elements to be *value-equivalent* if their tuples are equal.

**Definition 6.5** (Value Equivalence). Let  $S_1^x \in \mathbb{S}_{\mathcal{T}}^x$ ,  $S_2^x \in \mathbb{S}_{\mathcal{T}}^x$ , where  $x \in \{r, p, l\}$ , be two streams whose tuples have a common super type  $\mathcal{T}$ . Let  $s_1$  and  $s_2$  be two stream elements from  $S_1^x$  and  $S_2^x$ , respectively, with tuples  $e_1$  and  $e_2$ . We denote both elements to be *value-equivalent* iff  $e_1 = e_2$ .

We use the term *value-equivalence* instead of *tuple-equivalence* to be compatible with the consensus glossary of temporal database concepts [JCE<sup>+</sup>94].

## 6.5 Stream Transformations

To make streams available in either the logical or physical algebra, raw streams have to be converted into the respective stream model first. In addition to these input stream transformations, we specify the conversion from a physical stream into its logical analog. The latter transformation is required to prove the semantic equivalence of query answers returned by logical and physical plans.

### 6.5.1 Raw Stream To Physical Stream

Let  $S^r$  be a raw stream of type  $\mathcal{T}$ . The function  $\varphi^{r \rightarrow p} : \mathbb{S}_{\mathcal{T}}^r \rightarrow \mathbb{S}_{\mathcal{T}}^p$  takes a raw stream as argument and returns a physical stream of the same type. The conversion is achieved by mapping every stream element of the raw stream into a physical stream element as follows:

$$(e, t) \mapsto (e, [t, t + 1]).$$

The time instant  $t$  assigned to a tuple  $e$  is converted into a chronon. Therefore, the resultant physical stream is a *chronon* stream. Since a raw stream is ordered by timestamps, the

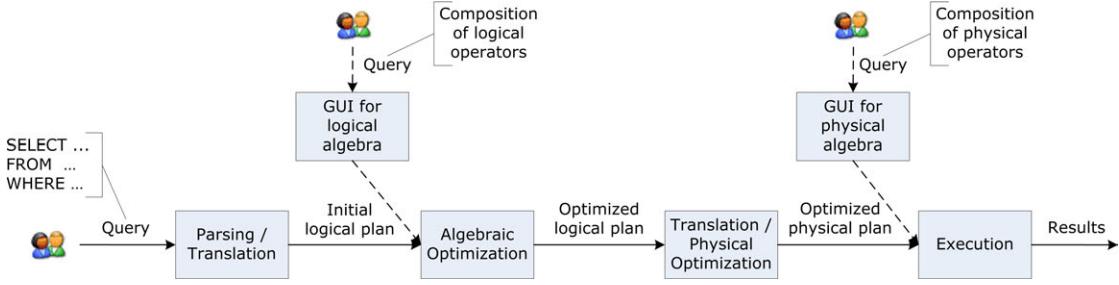


Figure 6.3: Steps from query formulation to query execution

resultant physical stream is properly ordered by start timestamps. The number of elements in a physical stream having identical start timestamps is finite because only a finite number of elements in a raw stream are allowed to have the same timestamp.

### 6.5.2 Raw Stream to Logical Stream

The function  $\varphi^{r \mapsto l} : \$_{\mathcal{T}}^r \rightarrow \$_{\mathcal{T}}^l$  transforms a raw stream  $S^r$  into its logical counterpart.

$$\varphi^{r \mapsto l}(S^r) := \{(e, t, n) \in \Omega_{\mathcal{T}} \times T \times \mathbb{N} \mid n = |\{(e, t) \in S^r\}| \wedge n > 0\} \quad (6.1)$$

The logical stream subsumes value-equivalent elements of the raw stream. Hence, the multiplicity  $n$  corresponds to the number of duplicate tuples  $e$  occurring in  $S^r$  at time instant  $t$ .

### 6.5.3 Physical Stream To Logical Stream

The function  $\varphi^{p \mapsto l} : \$_{\mathcal{T}}^p \rightarrow \$_{\mathcal{T}}^l$  transforms a physical stream  $S^p$  into its logical counterpart.

$$\begin{aligned} \varphi^{p \mapsto l}(S^p) := \{(e, t, n) \in \Omega_{\mathcal{T}} \times T \times \mathbb{N} \mid \\ n = |\{(e, [t_S, t_E]) \in S^p \mid t \in [t_S, t_E]\}| \wedge n > 0\} \end{aligned} \quad (6.2)$$

Recall that a logical stream describes the validity of a tuple  $e$  at time-instant level. For this reason, the time intervals assigned to tuples in the physical stream need to be split into single time instants. The multiplicity  $n$  of a tuple  $e$  at a time instant  $t$  corresponds to the number of elements in the physical stream whose tuple is equal to  $e$  and whose time interval intersects with  $t$ .

**Remark 6.1.** Often when there is no ambiguity we omit the superscript of a stream variable indicating the stream representation. For instance, throughout our logical and physical algebra, we label streams with  $S$  instead of  $S^l$  and  $S^p$  respectively.

## 6.6 Query Processing Steps

The structure of Part II follows the processes involved in optimizing and evaluating a query. Figure 6.3 shows this series of actions. The system receives a query language statement as input. First, this query string is parsed and translated into an initial logical query plan. A logical query plan is composed of operators from our logical algebra. In the next step, the query optimizer generates a number of equivalent logical query plans by applying transformation rules according to a plan enumeration algorithm. Due to the long-running queries, the optimizer also takes subplans of running queries into account to benefit from subquery sharing. Based on a cost model, the optimizer selects the best plan for processing. Thereafter, the optimizer translates the final logical plan into a physical query plan. This is accomplished by replacing every operator in the logical plan with a suitable counterpart of the physical algebra. Note that a logical operator can have several physical counterparts. For instance, a logical join can be substituted for a physical hash or nested-loops join. Finally, the physical plan is registered at the execution engine. The execution engine integrates the plan into the system's global query graph, which consists of all operative query plans, and initiates data processing. Alternatively, we permit the user to pose queries directly by constructing operator plans from either the logical or physical algebra via a graphical user interface.

## 6.7 Running Example

Examples throughout Part II are drawn from the *NEXMark benchmark* for data stream systems developed in the Niagara project [NDM<sup>+</sup>01]. The benchmark provides schema and query specifications from an online auctions system such as eBay. At any point in time, hundreds of auctions for individual items are open in such a system. The following actions occur continuously over time: (i) new people register, (ii) new items are submitted for auction, and (iii) bids arrive for items.

We use the schema and continuous queries published at the *Stream Query Repository* [SQR03] in the following; full details can be found in the original specification of the benchmark [TPPM02]. The schema consists of three relations – `Category`, `Item`, and `Person`, and three streams – `OpenAuction`, `ClosedAuction`, and `Bid`. Figure 6.4 shows the abbreviated conceptual schema of NEXMark. It illustrates the different types of tuples, their structure and dependencies. The NEXMark benchmark generator provides the data in a data-centric XML format. In our implementation, we utilized the Castor XML data binding framework [Exo99] for converting the tuples from XML format into Java objects.

As this work deals with continuous queries over data streams, the examples in the following sections disregard the persistent relations until Section 17.4 reveals how we incorporate relations. In our terminology, all streams in the NEXMark benchmark are raw streams since tuples are associated with timestamps. For example, the `Bid` stream provides tuples of the form  $(itemID, bid\_price, bidderID)$  tagged with a timestamp.

**Remark 6.2.** Some applications may already provide physical streams as inputs to the system, i.e., they associate a tuple with a time period rather than a timestamp [MFHH05, DGH<sup>+</sup>06, DGP<sup>+</sup>07]. In this case, we omit the transformation from the raw to the physical

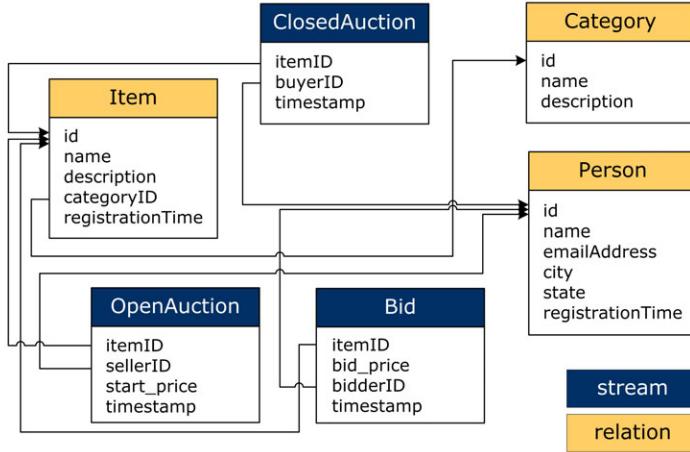


Figure 6.4: Conceptual schema of NEXMark auction data

stream and connect the input streams directly with the physical operator plans. The logical source streams are obtained from converting every physical input stream into a logical one. Even in the NEXMark scenario, it would be possible and meaningful to use a physical stream as input if a single auction stream is used instead of two separate streams indicating auction starts and endings (**OpenAuction** and **ClosedAuction**). As a result, the time interval associated to an auction would correspond to the auction duration. For compliance reasons, however, we stick firmly to the suggested conceptual schema of the NEXMark benchmark.

# 7 Query Formulation

PIPES supports two different ways to formulate a continuous query: (i) via a *graphical interface* or (ii) via a *query language*. Section 7.1 illustrates the use of graphical interfaces for expressing continuous queries, whereas Section 7.2 presents our declarative query language.

## 7.1 Graphical Interface

A graphical interface [CCC<sup>+</sup>02, KS04] is a convenient method to let users express a query. This interface allows the user to build query plans from our operator algebra manually, by connecting the individual operators in an appropriate manner. The graphical interface may either rely on the logical or the physical operator algebra.

- *Formulation with logical operators*: The user specifies the query semantics but does not influence query execution. When the user constructs a logical query plan, the query optimizer is responsible for plan optimization and the translation into a physical query plan. Moreover, the optimizer takes into account already running queries to enable subquery sharing. Finally, the optimizer integrates the new plan into the global, operative query graph.
- *Formulation with physical operators*: The user directly connects the physical operators and hence is responsible for query efficiency. Furthermore, the user has to install the plan correctly into the global query graph executed by the system. An automated query optimization does not take place. Since this method gives more responsibility to the user, it is only suited for experienced ones, which know how to profit from the manifold implementation facets.

## 7.2 Query Language

A declarative query language is another prevalent way to express queries. In recent years several extensions to SQL have been proposed that address the specific requirements of continuous queries [TGNO92, Sul96, CJSS03, ABW03]. Although other query languages can be set up on top of our logical operator algebra, we favor a slight extension of standard SQL with a subset of the window constructs definable in SQL:2003.

### 7.2.1 Characteristics

The following aspects caused us to keep our query language for data streams as close as possible to native SQL:

- The standardized language syntax and semantics are well-understood. On the one hand, this simplifies the development and maintenance of complex applications. On the other hand, the familiar language notation and meaning improves the progress of acceptance by a large user population.
- The language provides a high-level of abstraction due to the principle of logical and physical *data independence*. This means that the underlying data, its representation, and management is cleanly separated from the applications that use the data. Data independence enables the system to perform query optimizations. As data independence hides implementation details from the user, the exchange of internal data structures and algorithms does not necessitate any changes to the existing applications. Another advantage is that the user neither has to deal with low-level implementation details nor has to consider available physical resources, their allocation and distribution. In general, the optimization potential ranges from exploiting algebraic equivalences via transformation rules to high-performance distributed execution strategies. All those optimizations are aimed at achieving efficient query execution plans with low response times.
- Although the expressive power of standard SQL is not sufficient for processing potentially unbounded streams of ordered data [LWZ04], SQL can easily be enriched with stream-oriented operations, for instance, windowing constructs to define the scope of an operation or user-defined aggregates specifying incremental evaluation steps [ABW03, LWZ04, BTW<sup>+</sup>06]. Extensions also exist for handling stream disorder and imperfections [ACC<sup>+</sup>03].
- Since many applications require access and manipulation of both streaming and historical data [SQR03], the language should support constructs to effectively deal with conventional relations as well as streams. SQL natively provides rich functionality to define and manipulate persistent data. Our novel language extensions allow the user to manage and process streams as well. The operations from the extended relational algebra form the basic set of operations for both, streaming and persistent data.

### 7.2.2 Registration of Source Streams

Recall that a source stream is obtained from mapping a raw stream into our internal stream representation. To register a source stream in the system, it is important to declare its name and schema first. This can be done with the `CREATE STREAM` clause (see Example 7.1). Registering a source stream adds an entry to the input stream catalog of the system. In analogy to the `CREATE TABLE` command in SQL, the name declaration is followed by the schema definition. The `SOURCE` clause is a new feature of our query language and consists of two parts. The first part is a user-defined function that establishes the connection to the actual underlying data source, e. g., a raw stream obtained through a network connection. The second part indicates the attribute according to which the stream is ordered. It starts with the keywords `ORDERED BY` and expects a single attribute name.

**Example 7.1.** Let us consider the following statement for the registration of a raw stream.

```

<from clause> ::= FROM <stream reference> [<>window specification>]
                  [{, <stream reference> [<>window specification>]}...]

<window specification> ::= WINDOW([<window partition clause>]
                                <window frame clause> [<slide clause>])
<window partition clause> ::= PARTITION BY <column list>
<window frame clause> ::= <window frame units> <window frame start>
<window frame units> ::= ROWS | RANGE
<window frame start> ::= <value specification> | UNBOUNDED
<value specification> ::= <positive integer> [<units>]
<slide clause> ::= SLIDE <value specification>

```

Figure 7.1: BNF grammar excerpt for extended FROM clause and window specification

```

CREATE STREAM OpenAuction(itemID INT, sellerID INT,
                           start_price REAL, timestamp TIMESTAMP)
  SOURCE establishConnection('port34100', 'converter')
  ORDERED BY timestamp;

```

The user-defined function *establishConnection* takes a port number and a converter as arguments. The converter transforms the sequence of bytes arriving at port 34100 into a raw stream. PIPES provides special converters based on the *Castor XML* framework [Exo99] to transform a data-centric XML stream into a sequence of Java objects. Thus, we model the elements of the raw stream *OpenAuction* as Java objects composed of a tuple component with the form *(itemID, sellerID, start\_price)* and a separate timestamp attribute.

**Remark 7.1** (Timestamp Attribute). Be aware that, although the timestamp attribute is explicitly defined, it is not part of the stream schema. As a consequence, it cannot be referenced in queries. However, it is required in the stream definition to identify input streams with *explicit timestamps*. As some streams might provide multiple timestamp attributes, the ORDERED BY clause determines the attribute according to which the stream is ordered. If no ORDERED BY clause is specified, the system assigns an *implicit timestamp* upon arrival of an element using the system's internal clock.

The command `DROP STREAM` followed by the stream name removes a registered source stream from the system's catalog if all dependent queries have been stopped before. Otherwise an exception is thrown indicating that the source stream is still required for the evaluation of continuous queries.

### 7.2.3 Continuous Queries

The specification of a continuous query in our language resembles the formulation of one-time queries in native SQL using the common `SELECT`, `FROM`, `WHERE`, and `GROUP BY` clauses. While being as close to SQL as possible, our query language supports most of the powerful query specification functionality provided by standard SQL (SQL:1992), for instance, control over duplicates, nested subqueries, aggregates, quantifiers, and disjunctions [Day87].

## Window Specification in FROM clause

As windowing constructs play an important role in stream processing, we slightly enhanced the `FROM` clause with a `WINDOW` specification. The BNF grammar for this specification is shown in Figure 7.1. To define a window over a stream, the window specification has to follow the stream reference in the `FROM` clause. Rather than defining an entirely new syntax for windows from scratch, we reused the `WINDOW` specification of SQL:2003. Windows in SQL:2003 are restricted to built-in OLAP functions and thus are only permitted in the `SELECT` clause. Conversely, we aimed to let windows slide directly over any stream, base and derived streams, e. g., to express windowed stream joins. For this reason, we needed to extend the `FROM` clause. A detailed comparison with the window expression in SQL:2003 reveals two further changes. First, we solely consider preceding windows as only these window types make sense in stream processing. The keyword `PRECEDING` then becomes superfluous and, therefore, we dropped it. Second, we added an optional `SLIDE` clause to specify the window advance, which, for instance, is required to express tumbling windows.

### Default Window

The BNF grammar for the extended `FROM` clause allows the user to omit the window specification. In that case, the default window is a time-based window of size 1, which covers all elements of the referenced input stream valid at the current time instant. We call this special type *now-window*. The following sections will show that now-windows can be omitted in the query formulation and also in query plans because evaluating now-windows produces the identity.

**Example 7.2.** Let us consider some examples of continuous queries formulated in our enriched SQL language.

**Currency Conversion Query** – Convert the prices of incoming bids from U.S. dollars into euros.

```
SELECT itemID, DolToEuro(bid_price), bidderID  
FROM Bid;
```

`DolToEuro` is a user-defined function to convert a dollar price to euros. The function is invoked on every incoming bid. The query does not specify a windowing construct, hence a now-window is applied by default.

**Selection Query** – Select all bids on a specified set of 5 items.

```
SELECT Bid.*  
FROM Bid  
WHERE itemID = 1007 OR itemID = 1020  
      OR itemID = 2001 OR itemID = 2019  
      OR itemID = 1087;
```

The filter condition is expressed in the `WHERE` clause listing the item identifiers of interest.

**Short Auctions Query** – Report all auctions that closed within 5 hours of their opening.

```
SELECT OpenAuction.*  
FROM OpenAuction O WINDOW(RANGE 5 HOURS),  
      ClosedAuction C  
WHERE O.itemID = C.itemID;
```

This query is an example for a *windowed stream join*. It takes two streams as input, namely the `OpenAuction` and `ClosedAuction` stream. A time-based window of range 5 hours is defined over the `OpenAuction` stream, and a now-window is applied to the `ClosedAuction` stream by default. The join condition checks equality on item identifiers.

#### 7.2.4 Registration of Derived Streams

The result of a continuous query is a data stream. Depending on the operator algebra used, this derived stream is either a logical or physical stream. Besides the functionality to register a source stream, it also possible to make derived streams available as inputs for other continuous queries. Like the `CREATE VIEW` mechanism in SQL, a derived stream can be defined by

```
CREATE STREAM <stream name> AS  
    <query expression>.
```

**Example 7.3. Closing Price Query** – Report the closing price and seller of each auction.

```
CREATE STREAM CurrentPrice AS  
    SELECT P.itemID, P.price, O.sellerID AS sellerID  
    FROM ((SELECT itemID, bid_price AS price  
            FROM Bid WINDOW(RANGE 2 DAYS))  
          UNION ALL  
        (SELECT itemID, start_price AS price  
            FROM OpenAuction WINDOW(RANGE 2 DAYS))) P,  
              ClosedAuction C,  
              OpenAuction O WINDOW(RANGE 2 DAYS)  
    WHERE P.itemID = C.itemID AND C.itemID = O.itemID;  
  
CREATE STREAM ClosingPriceStream AS  
    SELECT itemID, sellerID, MAX(P.price) AS price  
    FROM CurrentPrice P,  
    GROUP BY P.itemID, P.sellerID;
```

The NEXMark benchmark generator sets the maximum auction duration to two days, hence, we use this timespan in the windowing constructs. Furthermore, it is assumed that the closing price in an auction is the price of the maximum bid, or the starting price of the auction in cases there were no bids [SQR03]. We choose this continuous query to demonstrate the following language features: the use of *derived streams*, *time-based sliding window joins*, and *windowed grouping with aggregation*. Furthermore, the query is relatively complex and offers optimization potential.

**Remark 7.2.** If we modeled the online auction application with a single auction stream using time intervals to express an auction's duration, the closing price would be much easier to compute (see also remark in Section 6.7). This approach makes sense since the auction duration is usually known in advance. In this case, it would be satisfactory to define a now-window on the Bid stream.

### Query Catalog

The system maintains a query catalog indexing all running queries. Whenever the user poses a continuous query, an entry is added to this catalog. The entry contains the query name and the schema of the output stream. In the case of a derived stream, the query name equals the user-defined name for the output stream. Otherwise, if the user just specified a SELECT statement, a unique query name is generated automatically and handed over to the user prior to query execution. The query name enables a user to stop the execution of a continuous query and to remove it from the system afterwards. The prerequisite for removing a query physically is that it is not shared by other still running queries. Consequently, all dependent queries have to be stopped and removed first. The language syntax for removing a derived stream is identical to the removal of a source stream.

#### 7.2.5 Nested Queries

It is obvious that our query language supports the formulation of nested queries because derived streams are allowed as inputs for continuous queries. The declaration and use of derived streams makes the language more appealing, as it is easier and more intuitive to express complex queries. However, the query result would be unchanged if all names of derived streams were replaced by the corresponding query specification. Besides the simple type of subqueries standing for derived streams, our query language also supports the more complicated types of subqueries in SQL like nested queries with aggregates or quantifiers [Day87].

**Example 7.4. Highest Bid Query** – Return the highest bid(s) in the last 10 minutes.

```
SELECT itemID, bid_price
FROM   Bid WINDOW(RANGE 10 MINUTES),
WHERE  bid_price = (SELECT MAX(bid_price)
                    FROM BID WINDOW(RANGE 10 MINUTES));
```

We choose this query as an example for a *subquery with aggregation*. The nested query in the WHERE clause determines the maximum bid price over a sliding window of ten minutes. The outer query applies the same window (identical FROM clause) and checks whether the price associated to a bid is equal to the maximum.

**Hot Item Query** – Select the item(s) with the most bids in the past hour.

```
SELECT itemID
FROM  (SELECT    B1.itemID AS itemID, COUNT(*) AS num
       FROM      Bid [RANGE 60 MINUTES] B1
       GROUP BY B1.itemID)
```

```
WHERE num >= ALL(SELECT COUNT(*)
                  FROM Bid [RANGE 60 MINUTES] B2
                  GROUP BY B2.itemID);
```

The example demonstrates a query with *universal quantification*. The nested query in the WHERE clause computes the number of bids received for each item over the last hour. The subquery in the FROM clause computes the same aggregate but assigns it to the corresponding item identifier. The outer query eventually checks whether the number of bids associated to an item identifier by the subquery in the FROM clause is equal to or greater than *all* values returned by the subquery in the WHERE clause.

### 7.2.6 Comparison to CQL

The previous examples illustrate that our query language is very close to native SQL. Hence, we can profit from the manifold advantages mentioned above. At the end of this section, we want to compare our language with CQL [ABW03, ABW06], which is closest in spirit to ours. A bunch of continuous queries in various application domains have been formulated in CQL and made available to the public [SQR03]. All these queries can be transformed into our query language since it is at least as expressive as CQL (see Chapter 15). We have already successfully demonstrated this property for the above examples drawn from the online auction scenario.

#### Similarities

At first glance, the specification of continuous queries in our language looks similar to that of CQL because both are largely based on SQL. They rely on the same windowing constructs, including support for time-based, count-based, and partitioned windows. Although [ABW06] does not define window slides formally, many of the exemplary queries in the stream query repository contain them [SQR03]. Their syntax and semantics are explained in [Ara06].

#### Differences

The attentive reader might have observed that our query language does not make use of the keywords `Istream`, `Dstream`, and `Rstream`, heavily used in CQL queries. CQL requires these relation-to-stream operators due to its *abstract semantics*. An apparent drawback of CQL and its abstract semantics is that even simple queries consisting only of a single operation, like a filter or map over a stream, require three successive steps.

1. A stream-to-relation operator converts the stream into a time-varying relation. Any windowing construct is a stream-to-relation operator in CQL.
2. A relation-to-relation operator performs the corresponding operation on the time-varying relation, i. e., a relational filter or map adapted to time-varying relations.
3. A relation-to-stream operator transforms the time-varying result relation back into a stream. `Istream`, `Dstream`, and `Rstream` are relation-to-stream operators.

In contrast to CQL, our query language is purely stream-based. Instead of employing the indirect route of using the relational algebra to process streams, our operators take one or multiple streams directly as input and generate a stream as output. Hence, there is no need to transform streams into time-varying relations and vice versa. Because of this, we distinguish only between the following two classes of stream-to-stream operators in our algebra.

1. *Window operators* are unary operators modeling the windowing constructs. They replace the stream-to-relation operators of CQL.
2. *Standard operators* are those operators derived from the extended relational algebra. We provide stream-to-stream counterparts for each relational operator.

The following reasons inspired us to develop a stream-only query language on top of a stream-to-stream operator algebra.

- It is quite natural to process streams directly rather than turning them into time-varying relations and back into streams.
- CQL can still be used as query language on top of our logical and physical operator algebra. Details on the algebraic expressiveness of our approach will be discussed in Chapter 15.
- Our logical operator algebra defines operator semantics in a precise and direct manner. It clearly shows the temporal effects on individual operators. In our opinion, this is important since the semantics of continuous queries is defined over time. The abstract semantics defined for CQL hides those temporal effects because it does not explicitly specify how time-varying relations are processed.
- The concrete rather than abstract semantics makes the relationship between a logical operator and its physical counterpart transparent and is useful to understand stream and plan equivalences.
- The motivation behind CQL is to reuse the well-understood relational semantics. In comparison to this, we exploited the large body of work on temporal databases to establish a semantic foundation for stream processing because (i) time plays an important role in stream processing and (ii) this research field seemed to already provide promising approaches that enrich the relational model with temporal functionality.
- The relation-to-stream operators unnecessarily complicate the language, especially when applied to nested queries.

One might argue that CQL permits continuous queries over streams and time-varying relations. The latter is not possible in our language so far. Section 17.4 points out ways to include updatable relations. A further difference between CQL and our language is that CQL does not specify how to declare streams and how to remove continuous queries from the system.

## 8 Logical Operator Algebra

This chapter presents the operators of our logical algebra. Section 8.1 discusses the *standard operators*, i. e., those stream operators that have an analog in the extended relational algebra, whereas Section 8.2 defines the *window operators*. Rather than integrating windows into standard operators as done for instance in [KNV03, GÖ03a], we separated the functionalities. This is an important step towards identifying a basic set of stream operators. It avoids the redundant specification of windowing constructs in the various operators and facilitates exchanging window types. In our case, the combination of window operators with standard operators creates their windowed analogs.

**Example 8.1.** The examples throughout this section illustrating the semantics of our operators are based on the two following logical streams:

$$S_1 := \{(c, 1, 1), (a, 2, 3), (a, 3, 3), \\ (b, 3, 1), (a, 4, 3), (b, 4, 1), \\ (c, 4, 1), (b, 5, 2), (b, 6, 2)\}$$

and

$$S_2 := \{(b, 2, 2), (b, 3, 2), (a, 4, 1), \\ (b, 4, 1), (c, 4, 1), (a, 5, 2), \\ (b, 5, 1), (a, 6, 1), (c, 6, 2)\}$$

Table 8.1 shows the two logical streams from the snapshot (time instant) perspective in tabular form. Let the tuples of both streams belong to the same type  $\mathcal{T}$ . A table *row* indicates that the multiset of tuples listed in the second column is valid at the time instant specified in the first column. For the ease of presentation, we denote a multiset by

(a) Stream $S_1$		(b) Stream $S_2$	
T	Multiset	T	Multiset
1	$\langle c \rangle$	1	$\langle \rangle$
2	$\langle a, a, a \rangle$	2	$\langle b, b \rangle$
3	$\langle a, a, a, b \rangle$	3	$\langle b, b \rangle$
4	$\langle a, a, a, b, c \rangle$	4	$\langle a, b, c \rangle$
5	$\langle b, b \rangle$	5	$\langle a, a, b \rangle$
6	$\langle b, b \rangle$	6	$\langle a, c, c \rangle$

Table 8.1: Examples of logical streams

listing the tuples including duplicates in  $\langle \rangle$  brackets, instead of using a set notation with tuple-multiplicity pairs.

## 8.1 Standard Operators

We first introduce our basic set of operators consisting of: filter ( $\sigma$ ), map ( $\mu$ ), union ( $\cup$ ), Cartesian product ( $\times$ ), duplicate elimination ( $\delta$ ), difference ( $-$ ), grouping ( $\gamma$ ), and scalar aggregation ( $\alpha$ ). Section 8.1.9 reports on composite operators built of the basic ones, e. g., the join.

### 8.1.1 Filter

Let  $\mathbb{P}_{\mathcal{T}}$  be the set of all filter predicates over tuples of type  $\mathcal{T}$ . The *filter*  $\sigma : \mathbb{S}_{\mathcal{T}}^l \times \mathbb{P}_{\mathcal{T}} \rightarrow \mathbb{S}_{\mathcal{T}}^l$  returns all stream elements of a logical stream whose tuple satisfies a given *filter predicate*. A filter predicate  $p \in \mathbb{P}_{\mathcal{T}}$  is a function  $p : \Omega_{\mathcal{T}} \rightarrow \{\text{true}, \text{false}\}$ . The argument predicate is expressed as subscript. The definition of  $\sigma_p$  indicates that the input and output stream have the same type.

$$\sigma_p(S) := \{(e, t, n) \in S \mid p(e)\} \quad (8.1)$$

$\sigma_p(S_1)$	
T	Multiset
1	$\langle c \rangle$
2	$\langle a, a, a \rangle$
3	$\langle a, a, a \rangle$
4	$\langle a, a, a, c \rangle$

Table 8.2: Filter over logical stream  $S_1$

**Example 8.2.** Table 8.2 shows the results of a filter over logical stream  $S_1$  (see page 41) with filter predicate

$$p(e) := \begin{cases} \text{true} & \text{if } e = a \vee e = c, \\ \text{false} & \text{otherwise.} \end{cases}$$

### 8.1.2 Map

Let  $\mathbb{F}_{\text{map}}$  be the set of all mapping functions that map tuples of type  $\mathcal{T}_1$  to tuples of type  $\mathcal{T}_2$ . The *map* operator  $\mu : \mathbb{S}_{\mathcal{T}_1}^l \times \mathbb{F}_{\text{map}} \rightarrow \mathbb{S}_{\mathcal{T}_2}^l$  applies a given *mapping function*  $f \in \mathbb{F}_{\text{map}}$  to the tuple component of every stream element. The argument function is expressed as subscript.

$$\begin{aligned} \mu_f(S) := & \{(\hat{e}, t, \hat{n}) \mid \exists X \subseteq S. X \neq \emptyset \wedge \\ & X = \{(e, t, n) \in S \mid f(e) = \hat{e}\} \wedge \hat{n} = \sum_{(e, t, n) \in X} n\} \end{aligned} \quad (8.2)$$

The mapping function can be a higher-order function. It is therefore sufficient to have only a single mapping function. This definition of the map operator is more powerful than its relational counterpart as it allows to create tuples of an entirely different type as output. The projection operator of the extended relational algebra can only project to attributes and add new attributes by evaluating arithmetic expressions over the existing attributes. Note that the mapping function does not affect the timestamp.

$\mu_f(S_1)$	
T	Multiset
1	$\langle c \rangle$
2	$\langle a, a, a \rangle$
3	$\langle a, a, a, x \rangle$
4	$\langle a, a, a, x, c \rangle$
5	$\langle x, x \rangle$
6	$\langle x, x \rangle$

Table 8.3: Map over logical stream  $S_1$

**Example 8.3.** Table 8.3 displays the output stream of the map operator applied to stream  $S_1$  (see page 41) with mapping function  $f : \Omega_{\mathcal{T}} \rightarrow \Omega_{\mathcal{T}}$  defined by

$$f(e) := \begin{cases} x & \text{if } e = b, \\ e & \text{otherwise,} \end{cases}$$

where  $x \in \Omega_{\mathcal{T}}$ . All occurrences of tuple  $b$  are replaced by tuple  $x$ .

### 8.1.3 Union

The *union*  $\cup : \mathbb{S}_{\mathcal{T}}^l \times \mathbb{S}_{\mathcal{T}}^l \rightarrow \mathbb{S}_{\mathcal{T}}^l$  merges two logical streams of compatible types. The multiplicity of a tuple  $e$  at time instant  $t$  in the output stream results from the sum of the corresponding multiplicities in both input streams.

$$\begin{aligned} \cup(S_1, S_2) := & \{(e, t, n_1 + n_2) \mid \\ & ((e, t, n_1) \in S_1 \vee n_1 = 0) \wedge ((e, t, n_2) \in S_2 \vee n_2 = 0) \\ & \wedge n_1 + n_2 > 0\} \end{aligned} \quad (8.3)$$

**Example 8.4.** Table 8.4 illustrates the union of logical streams  $S_1$  and  $S_2$  (see page 41). The multisets of tuples from both streams are unified for every time instant.

### 8.1.4 Cartesian Product

The *Cartesian product*  $\times : \mathbb{S}_{\mathcal{T}_1}^l \times \mathbb{S}_{\mathcal{T}_2}^l \rightarrow \mathbb{S}_{\mathcal{T}_3}^l$  of two logical streams combines elements of both input streams whose tuples are valid at the same time instant. Let  $\mathcal{T}_3$  denote the output type. The auxiliary function  $\circ : \Omega_{\mathcal{T}_1} \times \Omega_{\mathcal{T}_2} \rightarrow \Omega_{\mathcal{T}_3}$  creates an output tuple by

$S_1 \cup S_2$	
T	Multiset
1	$\langle c \rangle$
2	$\langle a, a, a, b, b \rangle$
3	$\langle a, a, a, b, b, b \rangle$
4	$\langle a, a, a, a, b, b, c, c \rangle$
5	$\langle a, a, b, b, b \rangle$
6	$\langle a, b, b, c, c \rangle$

Table 8.4: Union of logical streams  $S_1$  and  $S_2$

concatenating the contributing tuples. The product of their multiplicities determines the multiplicity of the output tuple.

$$\times(S_1, S_2) := \{(\circ(e_1, e_2), t, n_1 \cdot n_2) \mid (e_1, t, n_1) \in S_1 \wedge (e_2, t, n_2) \in S_2\} \quad (8.4)$$

$S_1 \times S_2$	
T	Multiset
1	$\langle \rangle$
2	$\langle a \circ b, a \circ b \rangle$
3	$\langle a \circ b, b \circ b, b \circ b, b \circ b \rangle$
4	$\langle a \circ a, a \circ b, a \circ c, a \circ a, a \circ b, a \circ c, a \circ a, a \circ b, a \circ c, b \circ a, b \circ b, b \circ c, c \circ a, c \circ b, c \circ c \rangle$
5	$\langle b \circ a, b \circ a, b \circ b, b \circ a, b \circ a, b \circ b \rangle$
6	$\langle b \circ a, b \circ c, b \circ c, b \circ a, b \circ c, b \circ c \rangle$

Table 8.5: Cartesian product of logical streams  $S_1$  and  $S_2$

**Example 8.5.** Table 8.5 demonstrates the output stream of the Cartesian product over input streams  $S_1$  and  $S_2$  (see page 41) which, in turn, is the Cartesian product on the corresponding multisets at each time instant. For better readability, we used the infix notation for the concatenation function  $\circ$ .

### 8.1.5 Duplicate Elimination

The *duplicate elimination*  $\delta : \mathbb{S}_{\mathcal{T}}^l \rightarrow \mathbb{S}_{\mathcal{T}}^l$  eliminates duplicate tuples for every time instant. The multiplicities of all tuples are hence set to 1.

$$\delta(S) := \{(e, t, 1) \mid \exists n \in \mathbb{N}. (e, t, n) \in S\} \quad (8.5)$$

**Example 8.6.** Table 8.6 shows the output of the duplicate elimination for stream  $S_1$  (see

$\delta(S_1)$	
T	Multiset
1	$\langle c \rangle$
2	$\langle a \rangle$
3	$\langle a, b \rangle$
4	$\langle a, b, c \rangle$
5	$\langle b \rangle$
6	$\langle b \rangle$

 Table 8.6: Duplicate elimination on logical stream  $S_1$ 

page 41). The example nicely points out that duplicates are removed at time instant granularity.

### 8.1.6 Difference

The *difference*  $- : \mathbb{S}_{\mathcal{T}}^l \times \mathbb{S}_{\mathcal{T}}^l \rightarrow \mathbb{S}_{\mathcal{T}}^l$  subtracts elements of the second argument stream from the first argument stream as follows. Value-equivalent elements valid at the same time instant are subtracted in terms of multiplicities. The types of both streams need to be compatible.

$$\begin{aligned} -(S_1, S_2) := & \{(e, t, n) \mid (\exists n_1, n_2 \in \mathbb{N}. n_1 > n_2 \wedge \\ & (e, t, n_1) \in S_1 \wedge (e, t, n_2) \in S_2 \wedge n = n_1 - n_2) \\ & \vee ((e, t, n) \in S_1 \wedge \nexists n_2 \in \mathbb{N}. (e, t, n_2) \in S_2)\} \end{aligned} \quad (8.6)$$

**Remark 8.1.** At this point we want to highlight one major advantage of our descriptive, logical operator algebra, namely that the semantics can be expressed elegantly, compactly, and intuitively. The reason for this is that the logical algebra is totally independent of any implementation. While algorithmic descriptions can also be used to define the semantics of an operator, they turn out to be much more complicated. See, for example, the definition of the temporal difference using the  $\lambda$ -calculus in [SJS01], or the implementation of the difference in our physical operator algebra in Chapter 11.

$S_1 - S_2$	
T	Multiset
1	$\langle c \rangle$
2	$\langle a, a, a \rangle$
3	$\langle a, a, a \rangle$
4	$\langle a, a \rangle$
5	$\langle b \rangle$
6	$\langle b, b \rangle$

 Table 8.7: Difference of logical streams  $S_1$  and  $S_2$

**Example 8.7.** Table 8.7 illustrates the result of subtracting stream  $S_2$  from stream  $S_1$  (see page 41). For each time instant, the duplicate-sensitive difference is computed on the multisets.

### 8.1.7 Grouping

Let  $\mathbb{F}_{group}$  be the set of all grouping functions over type  $\mathcal{T}$ . Let  $k \in \mathbb{N}, k > 0$ , be the number of possible groups for a given input stream. A *grouping function*  $f_{group} \in \mathbb{F}_{group}, f_{group} : \Omega_{\mathcal{T}} \rightarrow \{1, \dots, k\}$  determines a *group identifier* for every tuple. The *grouping*

$$\gamma : \mathbb{S}_{\mathcal{T}}^l \times \mathbb{F}_{group} \rightarrow (\underbrace{\mathbb{S}_{\mathcal{T}}^l \times \dots \times \mathbb{S}_{\mathcal{T}}^l}_{k \text{ times}})$$

partitions the input stream into  $k$  disjoint substreams according to the given grouping function (expressed as subscript).

$$\begin{aligned} \gamma_{f_{group}}(S) &:= (S_1, \dots, S_k) \\ \text{where } S_j &:= \{(e, t, n) \in S \mid f_{group}(e) = j\} \end{aligned} \tag{8.7}$$

**Remark 8.2.** For the extended relational algebra, the grouping function would be based on equality of the grouping attribute values. Furthermore, grouping and aggregation are combined in a single operator, whereas we split the functionality into separate operators. This is helpful because we can reuse grouping to express *partitioned windows* (see Section 8.2.3).

We use the following *projection* operator together with the grouping operator to access a single substream (group). For a given group index  $j \in \{1, \dots, k\}$ ,  $\pi$  returns the  $j$ -th group. The group index is passed as subscript.

$$\pi_j(\gamma_{f_{group}}(S)) = S_j. \tag{8.8}$$

$\gamma_{f_{group}}(S_1)$						
$S_{1_1}$		$S_{1_2}$		$S_{1_3}$		
T	Multiset	T	Multiset	T	Multiset	
1	$\langle \rangle$	1	$\langle \rangle$	1	$\langle c \rangle$	
2	$\langle a, a, a \rangle$	2	$\langle \rangle$	2	$\langle \rangle$	
3	$\langle a, a, a \rangle$	3	$\langle b \rangle$	3	$\langle \rangle$	
4	$\langle a, a, a \rangle$	4	$\langle b \rangle$	4	$\langle c \rangle$	
5	$\langle \rangle$	5	$\langle b, b \rangle$	5	$\langle \rangle$	
6	$\langle \rangle$	6	$\langle b, b \rangle$	6	$\langle \rangle$	

Table 8.8: Grouping on logical stream  $S_1$

**Example 8.8.** Table 8.8 demonstrates the grouping. Input stream  $S_1$  (see page 41) is split into three substreams  $S_{1_1}, S_{1_2}$ , and  $S_{1_3}$  using the grouping function  $f : \Omega_{\mathcal{T}} \rightarrow \{1, 2, 3\}$

where

$$f(e) := \begin{cases} 1 & \text{if } e = a, \\ 2 & \text{if } e = b, \\ 3 & \text{otherwise.} \end{cases}$$

### 8.1.8 Scalar Aggregation

Let  $\mathbb{F}_{agg}$  be the set of all aggregate functions over type  $\mathcal{T}_1$ . An *aggregate function*  $f_{agg} \in \mathbb{F}_{agg}$  with  $f_{agg} : \wp(\Omega_{\mathcal{T}_1} \times \mathbb{N}) \rightarrow \Omega_{\mathcal{T}_2}$  computes an *aggregate* of type  $\mathcal{T}_2$  from a set of elements of the form *(tuple, multiplicity)*. The aggregate function is specified as subscript.  $\wp$  denotes the power set. The *aggregation*  $\alpha : \mathbb{S}_{\mathcal{T}_1}^l \times \mathbb{F}_{agg} \rightarrow \mathbb{S}_{\mathcal{T}_2}^l$  evaluates the given aggregate function for every time instant on the non-temporal multiset of all tuples from the input stream being valid at this instant.

$$\begin{aligned} \alpha_{f_{agg}}(S) := \{ & (agg, t, 1) \mid \exists X \subseteq S. X \neq \emptyset \wedge \\ & X = \{(e, n) \mid (e, t, n) \in S\} \wedge agg = f_{agg}(X) \} \end{aligned} \quad (8.9)$$

The aggregation implicitly eliminates duplicates for every time instant as it computes an aggregate value for all tuples valid at the same time instant weighted by the corresponding multiplicities. Note that the aggregate function can be a higher-order function. As a result, it is also possible to evaluate multiple aggregate functions over the input stream in parallel. The output type  $\mathcal{T}_2$  describes the aggregates returned by the aggregate function. An aggregate consists of (i) the aggregate value(s), and (ii) grouping information. The latter is important if a grouping is performed prior to the aggregation. An aggregate function should retain the portion of the tuples relevant to identify their group. For the relational case, this portion would correspond to the grouping attributes. Recall that the scalar aggregation treats its input stream as a single group. Section 8.1.9 presents grouping with aggregation.

$\alpha_{\text{SUM}}(S_1)$	
T	Multiset
1	$\langle c \rangle$
2	$\langle a + a + a \rangle$
3	$\langle a + a + a + b \rangle$
4	$\langle a + a + a + b + c \rangle$
5	$\langle b + b \rangle$
6	$\langle b + b \rangle$

Table 8.9: Scalar aggregation over logical stream  $S_1$

**Example 8.9.** Table 8.9 illustrates the scalar aggregation for the sum. The tuples in the output stream are actually aggregate values, here denoted as sums. The aggregate value belonging to a time instant  $t$  is the sum of all tuples from  $S_1$  being valid at  $t$  (see page 41).

### 8.1.9 Derived Operators

In this section, we briefly describe the adaptation of some common but more complex relational operators to logical streams. Let  $S_1, S_2$  be two logical streams with types  $\mathcal{T}_1$  and  $\mathcal{T}_2$  respectively.

#### Theta-Join

The theta-join can be expressed as

$$\bowtie_{\theta,f} (S_1, S_2) := \mu_f(\sigma_\theta(S_1 \times S_2)), \quad (8.10)$$

where  $\theta$  denotes the join predicate and  $f$  the function that creates the join results.

#### Semi-Join

The semi-join is a specific theta-join that outputs only tuples from the first input stream. The function  $\pi_{\mathcal{T}_1}$  is a projection to the type of the first input stream.

$$\ltimes_\theta (S_1, S_2) := S_1 \bowtie_{\theta, \pi_{\mathcal{T}_1}} \delta(S_2) \quad (8.11)$$

#### Intersection

The intersection of two streams with compatible types can be formulated as

$$\cap (S_1, S_2) := S_1 - (S_1 - S_2). \quad (8.12)$$

#### Max-Union

The max-union merges two streams with compatible types by setting the multiplicity of a tuple in the output to the maximum multiplicity of this tuple in both input streams.

$$\cup_{max}(S_1, S_2) := (S_1 - S_2) \cup (S_2 - S_1) \cup (S_1 \cap S_2) \quad (8.13)$$

#### Strict Difference

Due to multiset semantics we also want to introduce the strict difference which differs from the difference presented in Section 8.1.6 by eliminating duplicates in the result.

$$-_{strict} (S_1, S_2) := S_1 - (S_1 \ltimes_=_ S_2) \quad (8.14)$$

The semi-join  $\ltimes_=_$  determines all elements from  $S_1$  whose tuple is equal to that of an element from  $S_2$ .

### Grouping with Aggregation

In our logical operator algebra, we consider grouping with aggregation as a derived operator. First, the *grouping* partitions the input stream according to the given grouping function  $f_{group}$  into  $k$  disjoint groups,  $k \in \mathbb{N}, k > 0$ . Next, a *scalar aggregation* with aggregate function  $f_{agg}$  is computed independently for each group. Finally, the output streams of the aggregation operators are merged.

$$\gamma_{f_{group}, f_{agg}}(S) := \bigcup_{i=1}^k \alpha_{f_{agg}}(\pi_i(\gamma_{f_{group}}(S))) \quad (8.15)$$

The union  $\bigcup_{i=1}^k$  corresponds to the natural extension of the binary union  $\cup$  for multiple inputs  $S_1, \dots, S_k$ . In Section 8.1.8, we already mentioned that all aggregates provide grouping information, which makes them distinguishable in the output stream, because the aggregate function preserves the portion of the tuples relevant to identify their group.

### Anti-Join

The anti-join can be expressed as

$$\triangleright_\theta(S_1, S_2) := S_1 - (S_1 \ltimes_\theta S_2), \quad (8.16)$$

where  $\theta$  denotes the join predicate. For every time instant, the anti-join returns an element from  $S_1$  only if no element exists in  $S_2$  that qualifies the join predicate.

**Remark 8.3.** Contrary to this logical point of view, composing derived operators from the basic ones may not be efficient from the implementation perspective. For instance, the theta-join can be implemented more efficiently as a single operator, because it is extremely expensive to compute the Cartesian product first and select the qualifying elements afterwards. For that reason, PIPES also provides optimized implementations for derived operators.

## 8.2 Window Operators

We extended our previous work [KS05], which focused on time-based sliding window queries, by count-based and partitioned windows for compliance with window specifications in SQL:2003 and CQL.

### 8.2.1 Time-based Sliding Window

The *time-based sliding window*  $\omega^{\text{time}} : \mathbb{S}_{\mathcal{T}}^l \times T \rightarrow \mathbb{S}_{\mathcal{T}}^l$  takes a logical stream  $S$  and the window size as arguments. The *window size*  $w \in T, w > 0$ , represents a period of (application) time and is expressed as subscript.<sup>1</sup> The operator shifts a time interval of size  $w$  time units

---

<sup>1</sup>Our notation  $w \in T$  indicates the number of time units captured by the window. The window size is given by the duration of the time interval  $[0, w]$ .

over its input stream to define the output stream.

$$\begin{aligned}\omega_w^{\text{time}}(S) := & \{(e, \hat{t}, \hat{n}) \mid \exists X \subseteq S. X \neq \emptyset \wedge \\ & X = \{(e, t, n) \in S \mid \max\{\hat{t} - w + 1, 0\} \leq t \leq \hat{t}\} \\ & \wedge \hat{n} = \sum_{(e, t, n) \in X} n\}\end{aligned}\tag{8.17}$$

At a time instant  $\hat{t}$ , the output stream contains all tuples of  $S$  whose timestamp intersects with the time interval  $[\max\{\hat{t} - w + 1, 0\}, \hat{t}]$ . In other words, a tuple appears in the output stream at  $\hat{t}$  if it occurred in the input stream within the last  $w$  time instants  $\leq \hat{t}$ .

There are two special cases of windows supported by CQL: the *now-window* and the *unbounded window*. The now-window captures only the current time instant of the input stream, hence,  $w = 1$ . We cover the semantics of the unbounded window,  $w = \infty$ , by defining that  $\forall \hat{t} \in T : \max\{\hat{t} - \infty, 0\} = 0$ . This means that all elements of  $S$  with timestamps  $\leq \hat{t}$  belong to the window.

$\omega_2^{\text{time}}(S_1)$	
T	Multiset
1	$\langle c \rangle$
2	$\langle a, a, a, c \rangle$
3	$\langle a, a, a, a, a, a, b \rangle$
4	$\langle a, a, a, a, a, a, b, b, c \rangle$
5	$\langle a, a, a, b, b, b, c \rangle$
6	$\langle b, b, b, b \rangle$
7	$\langle b, b \rangle$

Table 8.10: Time-based window over logical stream  $S_1$

**Example 8.10.** Table 8.10 specifies the logical output stream of the time-based sliding window operator applied to stream  $S_1$  (see page 41) with a window size of 2 time units. A tuple occurs in the output stream at time instant  $t$  if it occurs in the input stream at time instants  $t$  or  $t - 1$  for  $t - 1 \geq 0$ .

### 8.2.2 Count-based Sliding Window

The *count-based sliding window*  $\omega^{\text{count}} : \mathbb{S}_{\mathcal{T}}^l \times \mathbb{N} \rightarrow \mathbb{S}_{\mathcal{T}}^l$  defines its output stream over time by sliding a window of the last  $N$  elements over its input stream. The *window size*  $N \in \mathbb{N}, N > 0$ , is specified as subscript.

**Remark 8.4.** Our operator algebra presented in [KS05] did not provide count-based windows as they may cause ambiguous semantics. In order to be as expressive as CQL, this windowing construct is however required. The definition above complies with the informal description of *tuple-based windows* given in [ABW06]. Arasu et al. also found out that these windows may produce a nondeterministic output because the window size has

T	Multiset
1	$\langle c \rangle$
2	$\langle a, c \rangle$
3	$\langle a, b, c \rangle$
4	$\langle a, b, b \rangle$
5	$\langle a, b, b \rangle$
6	$\langle a, b, c \rangle$
...	...
$\infty$	$\langle a, b, c \rangle$

 Table 8.11: Logical stream  $S_3$ 

$\omega_3^{\text{count}}(S_3)$	
T	Multiset
1	$\langle c \rangle$
2	$\langle a, c \rangle$
3	$\langle a, b, c \rangle$
4	$\langle a, b, b \rangle$
5	$\langle a, b, b \rangle$
6	$\langle a, b, c \rangle$
...	...
$\infty$	$\langle a, b, c \rangle$

 Table 8.12: Count-based window over stream  $S_3$ 

to be exactly  $N$  elements, but multiple elements in an input stream may have the same timestamps. Thus, they are indistinguishable from each other and ties have to be broken in some manner. Count-based windows may therefore not be meaningful and appropriate whenever multiple tuples in the input stream are valid at the same time instant. We refer the interested reader to [Ara06] for the general definition of tuple-based windows. In this thesis, we restrict the usage of count-based windows to the unambiguous cases where the input stream must not contain multiple tuples being valid at the same time instant.

Let  $S$  be a logical stream in which at most a single tuple with multiplicity 1 is valid at any time instant.

$$\begin{aligned} \omega_N^{\text{count}}(S) := & \{(e, \hat{t}, \hat{n}) \mid \exists X \subseteq S. X \neq \emptyset \wedge \\ & X = \{(e, t, n) \in S \mid \max\{n(\hat{t}) - N + 1, 1\} \leq n(t) \leq n(\hat{t})\} \\ & \wedge \hat{n} = \sum_{(e, t, n) \in X} n \end{aligned} \quad (8.18)$$

where  $n(t)$  is the stream size at time instant  $t$  defined by

$$n(t) = |\{(e, \tilde{t}, n) \in S \mid \tilde{t} \leq t\}|.$$

At a time instant  $\hat{t}$ , the window captures the  $N$  tuples of  $S$  with the largest timestamps  $\leq \hat{t}$ . Note that the window may contain less than  $N$  tuples if  $S$  does not provide  $N$  tuples with timestamps  $\leq \hat{t}$ . A tuple  $e$  appears in the output stream at time instant  $\hat{t}$  if there exist one or more tuples  $e$  in  $S$  whose timestamp is among the  $N$  most recent timestamps  $\leq \hat{t}$ . The corresponding multiplicity  $\hat{n}$  is the sum of the multiplicities of the value-equivalent elements captured by the window. In our notation, the argument window size is expressed as subscript. Note that the definition above also covers the special case  $N = \infty$ , an unbounded window. In that case, we replace  $\max\{n(\hat{t}) - N + 1, 1\}$  with 1, i.e., the window will cover all elements of  $S$  with timestamps  $\leq \hat{t}$ .

**Example 8.11.** Table 8.11 defines the logical stream  $S_3$ , where only a single tuple is valid at every time instant. For this type of stream, the count-based window has a meaningful

semantics because there is no ambiguity in tuples being valid in the output stream at a certain time instant. Table 8.12 shows the results of a count-based window with size 3 over stream  $S_3$ . The tuples valid at time instant  $t$  in the output are those tuples of  $S_3$  valid at time instants  $t, t - 1, t - 2$ .<sup>2</sup> As only a single tuple is valid at a time instant  $t$  in  $S_3$ , the tuples with timestamps  $t, t - 1, t - 2$  correspond to the three most recent elements.

### 8.2.3 Partitioned Window

A *partitioned window*  $\omega^{\text{partition}} : \mathbb{S}_{\mathcal{T}}^l \times \mathbb{F}_{\text{group}} \times \mathbb{N} \rightarrow \mathbb{S}_{\mathcal{T}}^l$  logically splits its input stream into substreams according to the grouping function and applies a count-based window independently on each substream. We express a partitioned window as a derived operator. Let  $k \in \mathbb{N}, k > 0$ , be the number of substreams (groups) generated by the grouping function  $f_{\text{group}} \in \mathbb{F}_{\text{group}}$ , and let  $N \in \mathbb{N}, N > 0$ , be the size of the count-based windows. The grouping function and window size are expressed as subscript.

$$\omega_{f_{\text{group}}, N}^{\text{partition}}(S) := \bigcup_{i=1}^k \omega_N^{\text{count}}(\pi_i(\gamma_{f_{\text{group}}}(S))) \quad (8.19)$$

Note that time-based partitioned windows would provide no additional expressiveness as

$$\bigcup_{i=1}^k \omega_w^{\text{time}}(\pi_i(\gamma_{f_{\text{group}}}(S))) = \omega_w^{\text{time}}(S).$$

$\omega_{f_{\text{group}}, 1}^{\text{partition}}(S_3)$	
T	Multiset
1	$\langle c \rangle$
2	$\langle a \rangle$
3	$\langle a, b \rangle$
4	$\langle a, b \rangle$
5	$\langle a, b \rangle$
6	$\langle c, b \rangle$
...	...
$\infty$	$\langle c, b \rangle$

Table 8.13: Partitioned window over logical stream  $S_3$

**Example 8.12.** Table 8.13 displays the results of a partitioned window with size 1 applied to stream  $S_3$  (see page 51). The grouping function  $f : \Omega_{\mathcal{T}} \rightarrow \{1, 2\}$  is defined by

$$f(e) := \begin{cases} 1 & \text{if } e = a \vee e = c, \\ 2 & \text{otherwise.} \end{cases}$$

<sup>2</sup>The time instants  $t - 1$  and  $t - 2$  might not exist because 0 is the earliest possible time instant.

The implicit grouping generates two substreams to which a count-based window of size 1 is applied. A count-based window of size 1 entails a kind of *update semantics* because the next stream element (in ascending order by timestamps) replaces the previous one. In our case,  $(c, 1, 1)$  is replaced by  $(a, 2, 1)$  as both elements belong to the same substream (group). As stream  $S_3$  contains no elements with a timestamp greater than 6, the tuples in the output stream valid at this time instant continue to be valid up to infinity.



# 9 Logical Plan Generation

While query formulation with a graphical interface directly produces a query plan, queries expressed in a query language need to be compiled into a logical query plan. Parsing and translating a query from its textual representation into a logical plan closely resembles query plan construction in conventional database systems. In comparison with native SQL, we only extended the `FROM` clause with window specifications. Therefore, we need to clarify how the plan generator positions the window operators inside the plan. Section 9.1 discusses window placement in general, whereas Section 9.2 gives reasons for the omission of now-windows. Section 9.3 demonstrates plan generation for the example queries formulated in Section 7.2.

## 9.1 Window Placement

Whenever the parser identifies a window expression following a stream reference, the corresponding window operator is installed in the query plan downstream of the node standing for the stream reference. There are two cases:

1. If the stream reference represents a *source stream*, then the plan contains a leaf node as proxy for this stream reference. In this case, the plan generator installs the corresponding window operator downstream of the leaf node.
2. If the stream reference represents a *derived stream* (subquery), then the query plan has been built partially and consists of the sources and operators computing the derived stream. In this case, the plan generator installs the corresponding window operator as topmost operator of the partial plan.

The rest of the plan generation process is equal to that in conventional DBMSs, i. e., the plan generator constructs the plan downstream of the window operators as if these were not present.

## 9.2 Default Windows

Window placement is only performed for windows explicitly specified in the query language statement. The user may have omitted the specification of default windows for the sake of simplicity. The corresponding now-windows do thus not occur in the logical query plan. Here, we have already applied an optimization, namely that now-windows do not have any impact on a logical stream (see also Section 10.3.2). Plan generation can hence ignore any now-windows that are explicitly defined in the query statement. Omitting default windows in logical plans eventually saves computational resources

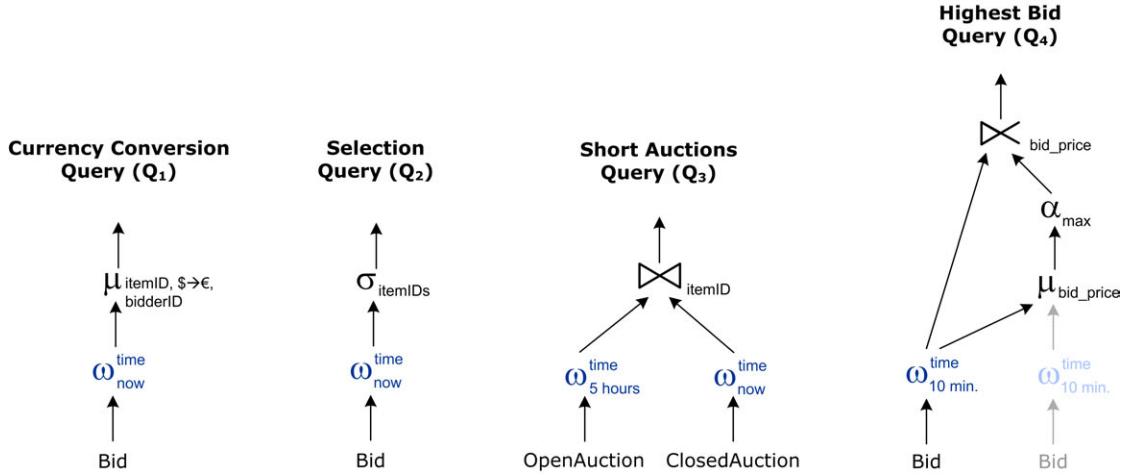


Figure 9.1: Logical Plans for NEXMark queries

because the corresponding operators also disappear in the physical plans generated afterwards.

### 9.3 Examples

After the definition of our logical operators and the explanation of window operator placement, we want to show the logical plans for the example queries given in Section 7.2 (see Figures 9.1 and 9.2). The logical plans display the now-windows for the sake of better comprehension.<sup>1</sup>

The plans for the currency conversion ( $Q_1$ ) and selection query ( $Q_2$ ) are straightforward. A now-window is simply placed upstream of the map and filter respectively. The short auctions query ( $Q_3$ ) uses time-based windows of different sizes. The window defined over the `OpenAuction` stream has a size of 5 hours, whereas the one over the `ClosedAuction` stream is a now-window. The subscript of the join symbol indicates that the join is defined on item identifiers. The projection to the attributes of the `OpenAuction` stream is performed by the function creating the join results. We omitted an extra symbol for that function. The plan for the highest bid query ( $Q_4$ ) is more complicated since it contains a subquery. Originally, each query is compiled into an operator plan that is a tree of operators. Because of subquery sharing, the operator plan diverges to a directed, acyclic operator graph. Subplans falling away due to subquery sharing appear in a lighter tone in our figures. In query  $Q_4$ , the time-based window over the `Bid` stream can be shared since it is a commonality of the outer query and the nested query. The nested query performs a projection to the bid price attribute and computes the maximum afterwards. The outer query returns those elements in the window that match the maximum bid price computed by the nested query, i. e., a semi-equijoin is placed as topmost operator.

<sup>1</sup>Other approaches like the positive-negative approach require the now-windows [ABW06, GHM<sup>+</sup>07] (see also Chapter 14).

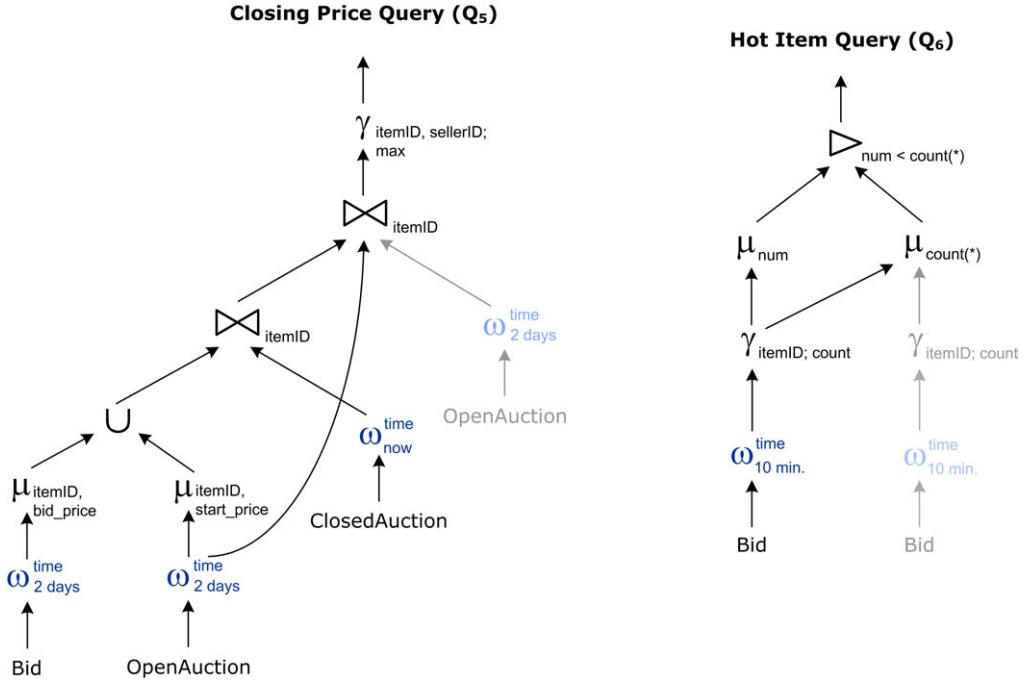


Figure 9.2: Logical Plans for NEXMark queries continued

Hence, the subquery with aggregation is resolved using a semi-join [Day87].

Figure 9.2 illustrates the logical plans for the more complex queries of the NEXMark scenario. In the closing price query ( $Q_5$ ), we first compute the derived `CurrentPrice` stream. To obtain the current price of an auction, we merge the `Bid` and `OpenAuction` streams. Next, we compute an equi-join on item identifiers over the merged streams with the `ClosedAuction` stream. Since the time-based window of size two days captures all bids relevant for an item, we obtain a join result for each bid plus a join result for the initiation of the auction. The following equi-join with the `OpenAuction` stream associates the `sellerID` to each resultant tuple of the first join, which is a pair of item identifier and bid price. Recall that the `sellerID` identifier was projected out before the union to make the schema compatible. Finally, a grouping with aggregation applied to the `CurrentPrice` stream computes the maximum price for each item identifier, which corresponds to the closing price.

The logical plan for the hot item query ( $Q_6$ ) resolves the nested query with universal quantification by means of an anti-join ( $\triangleright$ ) [Day87]. First, the number of bids issued for the individual items within the recent ten minutes is computed by grouping on item identifiers combined with a count aggregate. The count attribute is renamed to *num* for the first input of the join. The `itemID` attribute is projected out for the second input. The predicate of the anti-join verifies whether the *num* attribute of the first input is less than the count attribute of the second input. At every time instant, the anti-join outputs only those elements from the first input stream that do not satisfy the join predicate for any element in the second input stream (see Section 8.1.9). Consequently, at any time instant  $t$ ,

a result is only produced if a tuple exists in the first stream at  $t$  whose  $num$  value is equal to or greater than the  $count$  value of *all* tuples in the second stream at instant  $t$ .

# 10 Algebraic Query Optimization

The foundation for any query optimization is a well-defined semantics. Our logical algebra not only precisely specifies the output of our stream operators but is also expressive enough to support state-of-the-art continuous queries. This chapter explains why our approach permits query optimization and clarifies to which extent. Section 10.1 introduces an important property fulfilled by our standard operators, called *snapshot-reducibility*. Section 10.2 defines equivalences for logical streams and operator plans. Section 10.3 reports on transformation rules holding in our logical algebra, while Section 10.4 discusses their applicability.

## 10.1 Snapshot-Reducibility

In order to define snapshot-reducibility, we first introduce the *timeslice operation* that allows us to extract snapshots from a logical stream.

**Definition 10.1** (Timeslice). The *timeslice* operator  $\tau : (\mathbb{S}_T^l \times T) \rightarrow \wp(\Omega_T \times \mathbb{N})$  takes a logical stream  $S^l$  and a time instant  $t$  as arguments, where  $t$  is expressed as subscript. It computes the snapshot of  $S^l$  at  $t$ , i. e., the non-temporal multiset of all tuples being valid at time instant  $t$ .

$$\tau_t(S^l) := \{(e, n) \in \Omega_T \times \mathbb{N} \mid (e, t, n) \in S^l\} \quad (10.1)$$

For the relational case, a snapshot can be considered as an *instantaneous relation* since it represents the bag of all tuples valid at a certain time instant (compare to [ABW06]). Hence, the timeslice operator is a tool for obtaining an instantaneous relation from a logical stream.

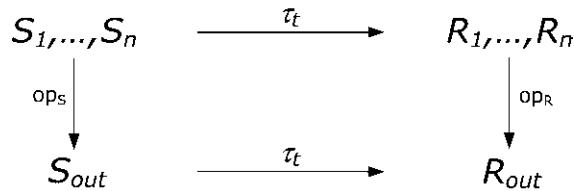


Figure 10.1: Snapshot-reducibility

**Definition 10.2** (Snapshot-Reducibility). We denote a stream-to-stream operator  $op_S$  with inputs  $S_1, \dots, S_n$  as *snapshot-reducible* iff for any time instant  $t \in T$ , the snapshot at  $t$  of the results of  $op_S$  is equal to the results of applying its relational counterpart  $op_R$  to the snapshots of  $S_1, \dots, S_n$  at time instant  $t$ .

Figure 10.1 shows a commuting diagram that illustrates the above definition. Snapshot-reducibility is a well-known concept in the temporal database community [SJS00, BBJ98, JCE<sup>+</sup>94]. It guarantees that the semantics of a relational, non-temporal operator is preserved in its more complex, temporal counterpart. As time plays an important role in stream processing as well, we decided to adopt the snapshot-reducibility property for our stream operators to the extent possible, with the aim to profit from the great deal of work done in temporal databases.

**Lemma 10.1.** *Let the tuples be relational tuples and operator functions and predicates adhere to the semantics of the extended relational algebra. It follows that the standard operators of our stream algebra are snapshot-reducible.*

*Proof.* There exists a non-temporal analog in the extended relational algebra for every standard operator in our logical algebra. Moreover, we defined the semantics of standard operators such that it satisfies the snapshot-reducibility property.  $\square$

For example, the duplicate elimination over logical streams is snapshot-reducible to the duplicate elimination over relations. However, the class of standard operators represents only a subset of our logical algebra. This class considered alone would not be expressive enough for our purpose because windowing constructs are missing. Our window operators are not snapshot-reducible.

## 10.2 Equivalences

Let us first derive an equivalence relation for logical streams, and thereafter for logical query plans.

**Definition 10.3** (Logical Stream Equivalence). We define two logical streams  $S_1^l, S_2^l \in \mathbb{S}_T^l$  to be *equivalent iff* their snapshots are equal.

$$S_1^l \doteq S_2^l : \Leftrightarrow \forall t \in T. \tau_t(S_1^l) = \tau_t(S_2^l) \quad (10.2)$$

**Definition 10.4** (Logical Plan Equivalence). Given two query plans  $Q_1$  and  $Q_2$  over the same set of logical input streams. Let each query plan represent a tree composed of operators from our logical operator algebra. Let  $S_1^l$  and  $S_2^l$  be the output streams of the plans respectively. We denote both plans as *equivalent iff* their logical output streams are equivalent, i. e., if for any time instant  $t$  the snapshots obtained from their output streams at  $t$  are equal.

$$Q_1 \doteq Q_2 : \Leftrightarrow S_1^l \doteq S_2^l \quad (10.3)$$

## 10.3 Transformation Rules

Due to defining our standard operations in semantic compliance with [SJS01], the plethora of transformation rules listed in [SJS01] carries over from temporal databases to stream processing. Here, semantic compliance means that our stream operators generate *snapshot-multiset-equivalent results* to the temporal operators presented in [SJS01]. The transfer

of this rich foundation for conventional and temporal query optimization enables us to algebraically optimize logical query plans over data streams.

### 10.3.1 Conventional Rules

Logical stream equivalence, which represents snapshot-equivalence over temporal multisets, causes the relational transformation rules to be applicable to our stream algebra.

**Definition 10.5** (Snapshot-Reducible Plans). We define a *snapshot-reducible plan* as a query plan that involves exclusively snapshot-reducible operators.

**Lemma 10.2.** *Given a snapshot-reducible plan, any optimizations with conventional transformation rules lead to an equivalent plan.*

*Proof.* Let  $S_1^l$  and  $S_2^l$  be the output streams of the original and optimized plan respectively. We have to show that for any time instant  $t$ ,  $\tau_t(S_1^l) = \tau_t(S_2^l)$ . This condition follows directly from the snapshot-reducibility property of all operators involved and the correctness of the conventional transformation rules for the relational algebra.  $\square$

Lemma 10.2 states that all conventional transformation rules apply equally to plans built with standard operations of our stream algebra. This includes for example the well-known rules for join reordering, predicate pushdown, and subquery flattening [GUW00, Day87]. Transformed into our notation, examples for transformation rules are:

$$(S_1^l \times S_2^l) \times S_3^l \doteq S_1^l \times (S_2^l \times S_3^l) \quad (10.4)$$

$$\sigma_p(S_1^l \cup S_2^l) \doteq \sigma_p(S_1^l) \cup \sigma_p(S_2^l) \quad (10.5)$$

$$\delta(S_1^l \times S_2^l) \doteq \delta(S_1^l) \times \delta(S_2^l) \quad (10.6)$$

The interested reader is referred to [SJS01] for a complete list of applicable transformation rules.

### 10.3.2 Window Rules

We can add some novel rules for window operators to the previous transformation rules. Our first rule says that default windows can be omitted. Recall that a default window is a now-window, i. e., a time-based window of size 1.

$$\omega_1^{\text{time}}(S^l) \doteq S^l \quad (10.7)$$

The correctness of this transformation rules directly derives from the definition of the time-based sliding window (see Section 8.2.1).

Furthermore, the time-based sliding window commutes with all stateless operators: filter, map, and union. For logical streams  $S^l, S_1^l$ , and  $S_2^l$ , we thus define the following

transformation rules:

$$\sigma_p(\omega_w^{\text{time}}(S^l)) \doteq \omega_w^{\text{time}}(\sigma_p(S^l)) \quad (10.8)$$

$$\mu_f(\omega_w^{\text{time}}(S^l)) \doteq \omega_w^{\text{time}}(\mu_f(S^l)) \quad (10.9)$$

$$\cup(\omega_w^{\text{time}}(S_1^l), \omega_w^{\text{time}}(S_2^l)) \doteq \omega_w^{\text{time}}(\cup(S_1^l, S_2^l)) \quad (10.10)$$

We actually would have to prove the correctness of each individual rule, but the proofs are quite similar and straightforward. The reason for the commutativity is that filter, map, and union do not manipulate the validity of tuples. A look at the physical algebra might improve the understanding of this argumentation (see Chapter 11). The filter predicates and mapping functions are only invoked on the tuple component of a stream element. From the perspective of the physical algebra, filter, map, and union satisfy the interval-preserving property formulated in [SJS01].

A system can profit from the above window rules for the following reasons. If a queue is placed between the filter and the window operator, pushing the filter below the window (Rule (10.8)) will reduce the queue size. In addition, this transformation saves processing costs for the window operator because the filtering is likely to decrease the input stream rate of the window operator. Altogether, the transformations can be useful for subquery sharing. For example, if a time-based window is applied after a union, and a new query is posed which could share the window functionality over one input stream, it is preferable to apply Rule (10.10) and push the window operator down the union.

Unfortunately, count-based and partitioned windows do not commute with filter and union in general because their semantics depends on the stream size (see Section 8.2.2). Nevertheless, we can formulate rules for the map operator.

$$\mu_f(\omega_N^{\text{count}}(S^l)) \doteq \omega_N^{\text{count}}(\mu_f(S^l)) \quad (10.11)$$

$$\mu_f(\omega_{f_{\text{group}}, N}^{\text{partition}}(S^l)) \doteq \omega_{f_{\text{group}}, N}^{\text{partition}}(\mu_f(S^l)) \quad (10.12)$$

if  $f$  does not conflict with  $f_{\text{group}}$

Rule (10.11) permits to commute the count-based window with the map operator. This transformation is possible since the map operator neither affects timestamps nor stream size. The same rule applies to the partitioned window if the following precondition is satisfied. The mapping function  $f$  must retain all information required to evaluate the grouping function  $f_{\text{group}}$  properly. This means for the relational case that function  $f$  must preserve all grouping attributes.

## 10.4 Applicability of Transformation Rules

Assume an arbitrary logical query plan to be given in the form of an operator tree. We first divide the plan into *snapshot-reducible* and *non-snapshot-reducible* subplans. The use of these properties enables us to decide where and which transformation rules can be applied properly.

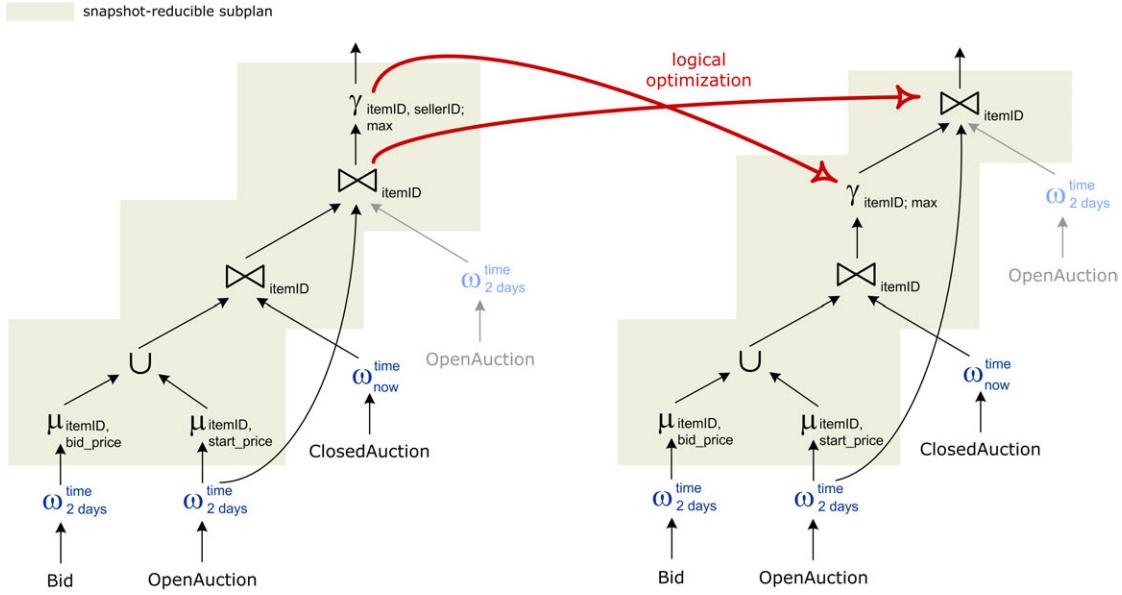


Figure 10.2: Algebraic optimization of a logical query plan

#### 10.4.1 Snapshot-Reducible Subplans

Conventional transformation rules can only be applied to snapshot-reducible subplans. Hence, those have to be identified first. This can be achieved by a bottom-up traversal. A new subplan starts (i) at the first snapshot-reducible operator downstream of a source or (ii) at the first snapshot-reducible operator downstream of a non-snapshot-reducible operator, e. g., a window operator. As long as snapshot-reducible operators are passed, these are added to the current subplan. Maximizing the subplans increases the number of transformation rules applicable.

A query optimizer should consider the property of whether a subplan is snapshot-reducible or not in its plan enumeration process to apply the transformation rules correctly. Such a property can be handled similarly to the property whether duplicates are relevant or not, applied in [SJS01] for optimization purposes.

**Example 10.1.** Figure 10.2 shows a possible algebraic optimization for the closing price query. The grey highlighted areas denote snapshot-reducible subplans. In this example, there exists only a single snapshot-reducible subplan that comprises all operators downstream of the window operators. We choose an optimization that is not obvious, namely, we push the grouping with aggregation below the join. This is possible because the attribute `itemID` is used for joining and grouping. The secondary grouping attribute `sellerID` specified for the grouping in the left plan just retains the seller information but does not affect the query answer in terms of additional results. The reason is an invariant of the auction scenario, namely, that every item can be sold only once.

Join reordering would be a further transformation rule applicable to the left plan. In that case, the output of the union would be joined with the windowed `OpenAuction` stream first, followed by the join with the windowed `ClosedAuction` stream.

During plan enumeration the optimizer will have to decide based on a cost model whether one of the proposed optimizations actually leads to an improved query execution plan or not.

Note that for all considered query plans of the NEXMark benchmark the snapshot-reducible portion is defined by the subplan downstream of the window operators. Nonetheless, our approach is more general and allows plans to have multiple window operators on a path. When this occurs, a snapshot-reducible subplan ends at the last operator upstream of the window.

#### 10.4.2 Non-Snapshot-Reducible Subplans

Non-snapshot-reducible subplans consist of window operators and standard operators whose functionality is beyond that of their relational analogs. For instance, our map operator is more powerful than its relational counterpart as it permits arbitrary functions to be invoked on tuples. Furthermore, it is possible to enhance our algebra with non-snapshot-reducible operators like temporal joins [GJSS05] to increase expressiveness (see also Section 15.2). The downside of this approach is that query optimization becomes quite limited. In general, query optimization is not feasible in non-snapshot-reducible subplans, apart from our window transformation rules. The optimizer should consider the latter in addition to the conventional transformation rules during plan enumeration. For this purpose, the optimizer has to check each window operator along with adjacent operators in the given query plan.

**Example 10.2.** By applying window transformation rule (10.9), it would be possible to push the map operators down the window operators for the plans shown in Figure 10.2. A further optimization would be to pull the window operator up the union because the window is identical for both inputs (Rule (10.10)). However, these optimizations would conflict with the subquery sharing performed on the window over the OpenAuction stream.

#### 10.4.3 Extensions to Multiple Queries

Due to the long-running queries and the large number of queries executed concurrently by a DSMS, the stand-alone optimization of individual query plans is not appropriate. The optimizer should rather take other running queries into account to identify common subexpressions. To profit from subquery sharing, DSMSs unify all query plans into a global, directed, acyclic query graph. Therefore, one objective during plan enumeration is whether transformation rules can be applied to enable sharing of common subqueries. DSMSs should consequently exploit multi-query optimization techniques [Sel88, RSSB00], but this is not enough. As queries are long-running, their efficiency may degrade over time because of changing stream characteristics such as value distributions and stream rates. Overcoming the resultant inefficiencies necessitates re-optimizations of continuous queries at runtime, called *dynamic query optimization* [ZRH04, YKPS07, KYC<sup>+</sup>06], in addition to the static optimization prior to execution. The optimization process itself is not the focus of this part, but we elaborate on this topic in Part IV.

# 11 Physical Operator Algebra

This chapter presents our physical operator algebra. Section 11.1 shows the motivation behind our time-interval approach, while Section 11.2 characterizes important properties of our physical operators. Our data structure for state maintenance is explained in Section 11.3. Prior to the presentation of the standard operators in Section 11.5, Section 11.4 spends some words on the notation used in our algorithms. Section 11.6 reveals our implementation for the window operators. Finally, Section 11.7 introduces two novel physical operators.

## 11.1 Basic Idea

From the implementation point of view, it is not satisfactory to process logical streams directly because evaluating the operator output separately for every time instant would cause a tremendous computational overhead. Hence, we developed operators that process physical streams. Recall that a physical stream is a more compact representation of its logical counterpart that describes the validity of tuples by time intervals. Any physical stream can be transformed into a logical stream with the function  $\varphi^{p \rightarrow l}$  defined in Section 6.5.3. The transformation splits the time intervals into chronons and summarizes the multiplicity of value-equivalent tuples being valid at identical chronons. This semantic equivalence is the sole requirement between a logical stream and its physical counterpart for algebraic query optimization. The inverse transformation, namely from a logical stream to a semantically equivalent physical stream, is not needed in practice because query execution does not make use of the logical algebra. Moreover, multiple physical streams may represent the same logical stream. The reason is that multiple value-equivalent elements with consecutive timestamps from a logical stream can be merged into elements of a physical stream in different ways because coalescing timestamps to time intervals is non-deterministic. The example in Figure 6.2 already demonstrated that a logical stream can have several physical counterparts.

The basic idea of our physical algebra is to use novel window operators for adjusting the validity of tuples, i. e. the time intervals, according to the window specifications, along with appropriately defined standard operators, so that the results of a physical query plan are snapshot-equivalent to its logical counterpart. While stateless operators do not consider the associated time intervals and thus can deal with potentially infinite windows well, time intervals affect stateful operators as follows. Elements are relevant to a stateful operator as long as their time interval may overlap with the time interval of any future stream element. This also means that a stateful operator can purge those elements from its state whose time interval cannot intersect with the time interval of any incoming stream element in the future. The latter explains how window specifications restrict the

resource usage of stateful operators in our physical algebra.

To the best of our knowledge, our time-interval approach is unique in the stream community. Related work embracing similar semantics substantially differs from our approach in terms of implementation [ABW06, GHM<sup>+</sup>07, AAB<sup>+</sup>05]. Those systems implement the positive-negative approach which propagates different types of elements, namely positive and negative ones tagged with a timestamp, through an operator plan to control expiration. As a consequence, stream rates are doubled which means that twice as many elements as in the time-interval approach have to be processed by operators in general. A more detailed comparison with this approach is given in Chapter 14.

## 11.2 Operator Properties

Our physical algebra provides at least a single implementation for each operation of the logical algebra. A physical operator takes one or multiple physical streams as input and produces a physical stream as output. These physical stream-to-stream operators are implemented in a data-driven manner assuming that stream elements are pushed through the query plan. Physical operators are connected directly when composing query plans. This implies that a physical operator has to process the incoming elements instantly on arrival without the ability to select the input from which the next element should be consumed. The pushed-based processing methodology is another important distinction between our implementation and related ones using pull-based techniques and inter-operator queues for communication [MWA<sup>+</sup>03, ABW06, ACC<sup>+</sup>03, HMA<sup>+</sup>04].

**Remark 11.1.** Our multi-threaded query execution framework [KS04, CHK<sup>+</sup>07] permits the concurrent arrival of stream elements at an operator with multiple input streams. Since we do not want to discuss synchronization and scheduling issues here, we assume the processing of a single stream element to be atomic and refer the interested reader to [CHK<sup>+</sup>07].

### 11.2.1 Operator State

The operators of our physical algebra can be classified into two categories:

- *Stateless operators* produce a result based on the evaluation of a single incoming element. As no other stream elements need to be considered, stateless operators do not need to maintain internal data structures keeping an extract of their input streams. In our physical algebra, the following operators are stateless: filter, map, union, and time-based window. Except for the union, these operators produce the result for an incoming element instantly on arrival. Because the union merges elements from two input streams, it has to regard the ordering requirement for the physical output stream. For this reason, an element of the first input stream cannot be emitted until all elements of the second input stream with a smaller start timestamp have been appended to the output stream and vice versa.
- *Stateful operators* need to access an internal data structure to generate a result. The internal data structure maintaining the operator state has to hold all elements that

may contribute to any future query results. Such a data structure has to support operations for efficient insertion, retrieval, and eviction. We categorize the following operators of our physical algebra as stateful: Cartesian product / join, duplicate elimination, difference, scalar aggregation, grouping with aggregation, count-based and partitioned window, coalesce and split.

### 11.2.2 Nonblocking Behavior

According to [BBD<sup>+</sup>02], “A blocking operator is a query operator that is unable to produce the first tuple of the output until it has seen the entire input.” Blocking operators are not suitable for stream processing due to the potentially unbounded input streams. Therefore, all stream-to-stream operators of our physical algebra have to be nonblocking. However, it is generally known that some relational operators are blocking, e. g., the aggregation or difference. The reason why we can provide stream variants of these operators is that we exploit windowing and incremental evaluation in our algorithms [GÖ03b]. If a query plan contains any stateful operators downstream of the window operators, we require the window sizes to be finite. As a result, the window operators restrict the scope of stateful operators to finite, sliding windows. As a consequence, the stateful operators downstream of the window operators do not block. With regard to time intervals, the following happens. Due to the finite window sizes, the length of time intervals in any physical stream downstream of the window operators is finite. As only a finite number of stream elements are allowed to have identical start timestamps (see Chapter 6), only a finite number of elements may overlap at an arbitrary time instant. Due to the fact that a stateful operator needs to keep only those elements in the state whose time interval may overlap with that of incoming elements in the future in order to generate the correct results, the total size of the operator state is limited. Any relational operator, except for sorting which is inherently blocking, can be unblocked by windowing techniques.

Stateless operators can even cope with unbounded streams as they do not require to maintain a state. We thus allow the user to specify unbounded windows in those queries that do not contain stateful operations.

### 11.2.3 Ordering Requirement

Each physical operator assumes that every input stream is correctly ordered. Moreover, each physical operator has to ensure that its output stream is properly ordered, i. e., elements are emitted in non-decreasing order by start timestamps (see Section 6.3.3). The ordering requirement is crucial to correctness because the decision which elements an operator can purge from its state depends on the progression of (application) time obtained from the start timestamps of the operator’s input streams. Recall that for the majority of streaming applications, the elements provided by raw streams already arrive in ascending order by start timestamps [BBD<sup>+</sup>02, GÖ03b]. Consequently, the ordering requirement is implicitly satisfied for the physical base streams of a query plan (see Section 6.5.1). If raw streams may receive out-of-order elements, e. g., due to network latencies, mechanisms like *heartbeats* and *slack* parameters can be applied in our approach

as well for correcting stream order [SW04, ACC<sup>+</sup>03]. Section 13.5 covers this aspect in more detail.

### 11.2.4 Temporal Expiration

Windowing constructs (i) restrict resource usage and (ii) unblock otherwise blocking operators over infinite streams. In stateful operators, elements in the state expire due to the validity set by the window operators. A stateful operator considers an element  $(e, [t_S, t_E])$  in its state as expired if it is guaranteed that no element in one of its input streams will arrive in the future whose time interval will overlap with  $[t_S, t_E]$ . According to the total order claimed for streams, this condition holds if the minimum of all start timestamps of the latest incoming element from each input stream is greater than  $t_E$ . A stateful operator can delete all expired elements from its state. We will show that some operators such as the aggregation emit these expired elements as results prior to their removal.

## 11.3 SweepAreas

Since our algorithms for stateful operators utilize specific data structures for state maintenance, we first want to introduce the *abstract data type SweepArea* (SA) and outline possible implementations. Besides SweepAreas, our algorithms also employ common data structures like queues, lists, and maps with their standard operations and semantics. Furthermore, we utilize iterators [Gra93] for traversing data structures.

### 11.3.1 Abstract Data Type SweepArea

The abstract data type (ADT) *SweepArea* models a dynamic data structure to manage a collection of elements having the same type  $\mathcal{T}$  (see page 69). Besides functionality to insert and replace elements and to determine the size, a SweepArea provides in particular generic methods for probing and eviction. Our development of this data structure was originally inspired by the sweepline paradigm [NP82].

The behavior of a SweepArea is controlled by three parameters passed to the constructor:

1. The *total order relation*  $\leq$  determines the order in which elements are returned by the methods *iterator* and *extractElements*.
2. The *binary query predicate*  $p_{query}$  is used in the method *query* to check whether the elements in the SweepArea qualify or not.
3. The *binary remove predicate*  $p_{remove}$  is used in the methods *extractElements* and *purgeElements* to identify elements that can be removed from the SweepArea.

### Probing

The function *query* probes the SweepArea with a *reference element*  $s$  given as parameter. It delivers an iterator over all elements  $\hat{s}$  of the SweepArea that satisfy  $p_{query}(s, \hat{s})$  for  $j = 1, \dots, n$ .

---

**ADT SweepArea**

---

total order relation  $\leq$ ;  
 binary predicate  $p_{query}$ ;  
 binary predicate  $p_{remove}$ ;

**SweepArea**(*total order relation*  $\leq$ , *binary predicate*  $p_{query}$ , *binary predicate*  $p_{remove}$ ) {  
 Creates a new instance with the given parameters;  
}

Procedure **insert**(*element*  $s$ ) {  
 Adds  $s$  to the SweepArea;  
}

Procedure **replace**(*element*  $\hat{s}$ , *element*  $s$ ) {  
 Replaces the first occurrence of element  $\hat{s}$  in the SweepArea with  $s$ ;  
}

Iterator **iterator**() {  
 Returns an iterator over all elements in the SweepArea in ascending order by  $\leq$ ;  
}

Iterator **query**(*element*  $s$ ,  $j \in \{1, 2\}$ ) {  
 Returns an iterator over all elements  $\hat{s}$  in the SweepArea for which  $p_{query}$  applied to  
 elements  $s$  and  $\hat{s}$  is satisfied. Parameter  $j$  determines whether element  $s$  becomes the first  
 and  $\hat{s}$  the second argument of  $p_{query}$ , or vice versa. The order in which the iterator provides  
 qualifying elements is not specified and depends on the actual implementation;  
}

Iterator **extractElements**(*element*  $s$ ,  $j \in \{1, 2\}$ ) {  
 Returns an iterator over all elements  $\hat{s}$  in the SweepArea for which  $p_{remove}$  applied to  
 elements  $s$  and  $\hat{s}$  is satisfied. Parameter  $j$  determines whether element  $s$  becomes the first and  $\hat{s}$  the second  
 argument of  $p_{remove}$ , or vice versa. The returned iterator provides the  
 qualifying elements in ascending order by  $\leq$ . Whenever the iterator returns an element, this  
 element is removed from the SweepArea;  
}

Procedure **purgeElements**(*element*  $s$ ,  $j \in \{1, 2\}$ ) {  
 Removes all elements  $\hat{s}$  from the SweepArea for which  $p_{remove}$  applied to elements  $s$  and  $\hat{s}$  is  
 satisfied. Parameter  $j$  determines whether element  $s$  becomes the first and  $\hat{s}$  the second  
 argument of  $p_{remove}$ , or vice versa;  
}

int **size0** {  
 Returns the size of the SweepArea;  
}

---

or  $p_{query}(\hat{s}, s)$  for  $j = 2$ . Parameter  $j$  determines whether  $s$  becomes the first or second argument of the query predicate. This argument swapping is required to cope with (i) asymmetric query predicates and (ii) query predicates over different argument types.

In contrast to the function *query*, where the order in which qualifying elements are delivered depends on the actual implementation of the SweepArea, the iterator returned by a call to the function *iterator* delivers all elements of the SweepArea in ascending order by  $\leq$ .

### Eviction

The methods *extractElements* and *purgeElements* share a common attribute in that they purge *unnecessary* elements from the SweepArea. While the function *extractElements* permits access to the unnecessary elements via the returned iterator prior to their removal, the method *purgeElements* instantly deletes them. We have defined two separate methods because the method *purgeElements* can often be implemented more efficiently than consuming the entire iterator returned by a call to the method *extractElements*. For example, the method *purgeElements* is superior to *extractElements* whenever the SweepArea is based on data structures that support bulk deletions. Both methods make use of the remove predicate  $p_{remove}$  and have a *reference element*  $s$  as parameter. An element  $\hat{s}$  from the SweepArea is considered as *unnecessary* if  $p_{remove}$  applied to  $s$  and  $\hat{s}$  evaluates to true. Analogously to the method *query*, parameter  $j$  is used to deal with asymmetric remove predicates and remove predicates over different argument types.

Note that it is also possible to remove elements from the SweepArea via the iterator returned by the method *query*. Whenever the method *remove* is called on this iterator, the last element returned by the iterator is removed from the underlying SweepArea.

#### 11.3.2 Use of SweepAreas

In a previous work of our research group [DSTW02], a similar form of the ADT SweepArea has been successfully used as the foundation for a powerful join framework. [DSTW02] nicely shows how to define the order relation and query and remove predicates to implement a plethora of join algorithms, e. g., band-, temporal-, spatial-, and similarity-joins. Due to the generic design, SweepAreas can be adapted easily to multi-way joins [VNB03, CHKS05]. The ADT SweepArea presented here is an extension to meet the requirements of our physical stream operator algebra. In the corresponding algorithms, SweepAreas are used to efficiently manage the state of operators. A SweepArea is maintained for each input of a stateful operator. From a top-level point of view, SweepAreas can be compared with synopses [ABW06] or state machine modules [RDH03] used in other DSMSs for state maintenance.

We decided to use SweepAreas inside our operators because a SweepArea gracefully combines functionality for efficient probing and purging on different criteria in a single generic data structure.

1. *Probing* is primarily based on the tuple component of stream elements. Hence, a data structure for state maintenance should arrange stream elements in an appropriate manner to ensure efficient retrieval.

2. *Purging* depends on the time interval component of stream elements. Due to temporal expiration, elements in the state become unnecessary over time. Keeping the state information small is important to save system resources. The smaller the state of an operator, the lower is its memory usage and processing costs. Thus, it is essential to get rid of unnecessary elements, namely the expired ones. Therefore, a data structure for state maintenance should not only provide efficient access to the tuples, but also to the time intervals.

### 11.3.3 Implementation Considerations

Up to this point we defined the semantics of the methods for the ADT SweepArea without discussing any particular implementation issues. Logically, a SweepArea can be viewed as a single data structure storing a collection of elements. In our case, however, a single data structure is often not adequate to ensure both efficient probing and purging. Because of this, we usually construct a SweepArea by combining two data structures. The *primary* data structure contains the elements and arranges them for probing, whereas the *secondary* data structure keeps references to these elements and arranges the references to facilitate purging. The downside of this dual approach is the increased cost of operations that have to be performed on both data structures to ensure consistency.

#### Example

Our algorithms utilize a hash table or list as primary data structure and a list or priority queue as secondary data structure in the majority of cases (see Sections 11.5, 11.6, and 11.7). Let us consider a concrete example. For a binary equi-join over physical streams, the SweepAreas are implemented as hash tables (primary data structure) whose elements are linked in ascending order by end timestamps using a priority queue (secondary data structure). While the hash table ensures efficient probing, we exploit the priority queue to detect and remove expired elements. Without the priority queue, purging would either be costly, if performed *eagerly*, as every bucket would need to be scanned for expired elements or, if performed *lazily*, purging would lead to an increased SweepArea size which, in turn, would raise memory consumption and processing costs. While this hash-based implementation is suitable for equi-joins, it is not appropriate for arbitrary theta-, similarity-, and spatial-joins. This shows that already the physical join necessitates various SweepArea implementations.

#### Expiration Patterns

Whether a list or priority queue is used as secondary data structure depends on the expiration pattern of the input stream providing the elements for the SweepArea. Golab et al. [GÖ05] categorize streams according to their expiration patterns. In our case, the expiration pattern of a physical stream is determined by the order of end timestamps. Recall that any physical stream has to satisfy the start timestamp ordering requirement. If stream elements also occur in non-decreasing order by their end timestamps, a list is sufficient as secondary data structure. Otherwise, when no assertions on the end

timestamp order can be given, a priority queue is used instead. The latter case applies particularly to derived streams, e.g., the output stream of a join. For our hash-join example, this means, using a list as secondary data structure instead of the priority queue would reduce operator cost because insertion and eviction of an element could be performed with constant costs. Due to the expiration pattern, elements would be appended at the end of the list and removed from its start. On the contrary, a priority queue implemented with a binary heap would cause the costs for insertion and eviction to be logarithmic in the size of the SweepArea in the worst case.

### Specializations

It is advisable to substitute the default SweepArea for an optimized one whenever applications allow the use of specialized and thus more efficient implementations, e.g., in the case spatial or similarity joins [DSTW02, DS00, APR<sup>+</sup>98], or due to specific application constraints like one-to-one joins. A SweepArea could be extended to exploit stream constraints [BSW04] and punctuations [TMSF03, DMRH04] to purge further unnecessary elements from the state. We have already developed SweepAreas permitting access to external memory to implement stream operators that spill parts of the state to disk, for instance, [UF01, VNB03, DSTW02, CHKS05]. Moreover, it might be advisable in some applications to leverage an index structure inside a SweepArea, e.g., an R-tree [BKSS90] indexing moving objects. The implementation of a SweepArea consequently has to be customized to the individual requirements of an operator and the particular application. For this reason, the complexity of an operator inevitably depends on the actual implementation of the SweepAreas employed. Independent of the actual implementation, any SweepArea has to adhere to the operator semantics. We will therefore specify the relevant SweepArea parameters such as order relation, query and remove predicates along with the description of our physical operators.

## 11.4 Notation

### 11.4.1 Algorithm Structure

Our algorithms obey the following basic structure which consists of three successive parts. The different parts in the algorithms are separated by blank lines.

1. *Initialization.* The required variables and data structures are initialized.
2. *Processing.* A *foreach*-loop processes the elements that arrive from the input streams of an operator.
3. *Termination.* For the case of finite input streams, additional steps might be necessary to produce the correct result although all stream elements have already been processed. After executing the termination part, an operator can delete all internal data structures and variables to release the allocated memory resources.

The termination case results from our input-triggered state maintenance concept which updates the operator state only at arrival of new stream elements.

(a) Stream $S_1$			(b) Stream $S_2$		
Tuple	$t_S$	$t_E$	Tuple	$t_S$	$t_E$
$c$	1	8	$b$	1	7
$a$	5	11	$d$	3	9
$d$	6	14	$a$	4	5
$a$	9	10	$b$	7	15
$b$	12	17	$e$	10	18

Table 11.1: Example input streams

### 11.4.2 Specific Syntax and Symbols

$s \hookleftarrow S_{in}$  denotes element  $s$  being delivered from input stream  $S_{in}$ .  $s \hookrightarrow S_{out}$  indicates that element  $s$  is appended to the output stream  $S_{out}$ .  $\emptyset$  denotes the empty stream. The syntax  $s := (e, [t_S, t_E])$  means that  $s$  is a shortcut for the physical stream element  $(e, [t_S, t_E])$ . A  $\leftarrow$  stands for variable assignments. We distinguish elements retrieved from a SweepArea from those being provided by the input stream by attaching a hat symbol, i. e.,  $\hat{s} := (\hat{e}, [\hat{t}_S, \hat{t}_E])$  signals that  $\hat{s}$  belongs to a SweepArea.

## 11.5 Standard Operators

Despite the different representations, the stream types for a logical operator and its physical counterpart are identical since those depend on the tuple components. Unless explicitly specified, operator predicates and functions comply with the definitions given in Chapter 8.

### 11.5.1 Filter

The filter (see Algorithm 2) evaluates the unary predicate  $p$  for each tuple  $e$  of an incoming stream element  $s := (e, [t_S, t_E])$ . If the predicate holds,  $s$  is appended to the output stream  $S_{out}$ . Otherwise,  $s$  is discarded. It is obvious that the output stream follows the same order as the input stream as the filter simply drops elements.

---

#### Algorithm 2: Filter ( $\sigma_p$ )

---

**Input** : physical stream  $S_{in}$ ; filter predicate  $p$   
**Output** : physical stream  $S_{out}$

```

1  $S_{out} \leftarrow \emptyset;$ 
2 foreach  $s := (e, [t_S, t_E]) \hookleftarrow S_{in}$  do
3   if  $p(e)$  then
4      $s \hookrightarrow S_{out};$ 

```

---

$\sigma_p(S_1)$		
Tuple	$t_S$	$t_E$
$a$	5	11
$a$	9	10

 Table 11.2: Filter over physical stream  $S_1$ 

**Example 11.1.** Table 11.2 shows the output stream of a filter with predicate

$$p(e) := \begin{cases} \text{true} & \text{if } e = a, \\ \text{false} & \text{otherwise.} \end{cases}$$

applied to input stream  $S_1$  (see page 73). The filter selects all elements from the input stream whose tuple equals  $a$ .

### 11.5.2 Map

The map operator (see Algorithm 3) transforms an element  $s := (e, [t_S, t_E])$  from the input stream into an element of the output stream by invoking the unary mapping function  $f$  on tuple  $e$ . Because the time intervals remain unchanged, the stream order of  $S_{out}$  is equal to that of  $S_{in}$ .

#### Algorithm 3: Map ( $\mu_f$ )

```

Input : physical stream  $S_{in}$ ; mapping function  $f$ 
Output : physical stream  $S_{out}$ 

1  $S_{out} \leftarrow \emptyset;$ 
2 foreach  $s := (e, [t_S, t_E]) \leftarrow S_{in}$  do
3    $\lfloor (f(e), [t_S, t_E]) \hookrightarrow S_{out};$ 

```

$\mu_f(S_1)$		
Tuple	$t_S$	$t_E$
$c$	1	8
$x$	5	11
$d$	6	14
$x$	9	10
$y$	12	17

 Table 11.3: Map over physical stream  $S_1$ 

**Example 11.2.** Table 11.3 shows the output stream of the map operator, which is applied

to input stream  $S_1$  (see page 73). The mapping function  $f : \Omega_{\mathcal{T}} \rightarrow \Omega_{\mathcal{T}}$  is defined by

$$f(e) := \begin{cases} x & \text{if } e = a, \\ y & \text{if } e = b, \\ e & \text{otherwise,} \end{cases}$$

where  $x, y \in \Omega_{\mathcal{T}}$ . In this case, the mapping function does not change the type and simply replaces all occurrences of tuples  $a$  and  $b$  by  $x$  and  $y$  respectively.

### 11.5.3 Union

The union (see Algorithm 4) merges two input streams  $S_{in_1}$  and  $S_{in_2}$  into a single output stream  $S_{out}$ . The min-priority queue  $Q$  ensures the proper order of  $S_{out}$  according to start timestamps. The *order relation*  $\leq_{ts}$  on physical stream elements is the less-than-or-equal-to relation on *start timestamps*, which is a total order. Incoming elements of both streams are inserted into min-priority queue  $Q$  with priority  $\leq_{ts}$ . In addition, the minimum start timestamp  $min_{ts}$  over the latest element of both input streams is updated for each incoming element.

---

**Algorithm 4:** Union ( $\cup$ )

---

```

Input : physical streams  $S_{in_1}, S_{in_2}$ 
Output : physical stream  $S_{out}$ 

1  $S_{out} \leftarrow \emptyset;$ 
2  $t_{S_1}, t_{S_2}, min_{ts} \in T \cup \{\perp\}; t_{S_1} \leftarrow \perp; t_{S_2} \leftarrow \perp; min_{ts} \leftarrow \perp;$ 
3 Let  $Q$  be an empty min-priority queue with priority  $\leq_{ts}$ ;
4  $j \in \{1, 2\};$ 
5 foreach  $s := (e, [t_s, t_E]) \leftarrow S_{in_j}$  do
6    $t_{S_j} \leftarrow t_s;$ 
7    $min_{ts} \leftarrow \min(t_{S_1}, t_{S_2});$ 
8    $Q.insert(s);$ 
9   if  $min_{ts} \neq \perp$  then
10     $\quad \text{TRANSFER}(Q, min_{ts}, S_{out});$ 
11 while  $\neg Q.isEmpty()$  do
12   $\quad Q.extractMin() \leftarrow S_{out};$ 

```

---

The procedure TRANSFER extracts the minimum of the given priority queue  $Q$  and appends it to the output stream  $S_{out}$  as long as the start timestamp of the minimum is less than or equal to the given parameter timestamp  $min_{ts}$ . Instead of integrating this code fragment directly into Algorithm 4, we defined an auxiliary procedure because we reuse this functionality in further algorithms.

---

**Procedure** TRANSFER(*min-priority queue Q, timestamp min<sub>t<sub>s</sub></sub>, stream S<sub>out</sub>*)

---

```

1 while  $\neg Q.\text{isEmpty}()$  do
2   Element  $(\tilde{e}, [\tilde{t}_S, \tilde{t}_E]) \leftarrow Q.\text{min}();$ 
3   if  $\tilde{t}_S \leq \text{min}_{t_S}$  then
4      $Q.\text{extractMin}() \hookrightarrow S_{out};$ 
5   else break;

```

---

### Output Stream Order

Let  $t_1, t_2 \in T \cup \{\perp\}$ . In our algorithms the function *min* is defined as follows:

$$\min(t_1, t_2) := \begin{cases} t_1 & \text{if } t_1, t_2 \in T \wedge t_1 \leq t_2, \\ t_2 & \text{if } t_1, t_2 \in T \wedge t_2 < t_1, \\ \perp & \text{if } t_1 = \perp \vee t_2 = \perp. \end{cases}$$

At the beginning  $\text{min}_{t_S}$  is undefined, i. e.,  $\text{min}_{t_S} = \perp$ . As soon as at least one element arrived from each input stream,  $\text{min}_{t_S} \neq \perp$ . The union extracts the minimum of queue  $Q$  and appends it to the output stream if its start timestamp is less than or equal to  $\text{min}_{t_S}$ . This step is repeated as long as the queue contains elements with a start timestamp  $\leq \text{min}_{t_S}$ . From the property that each input stream is ordered non-decreasingly by start timestamps follows that no element will arrive from either input stream having a start timestamp less than  $\text{min}_{t_S}$ . This means, the order of the output stream is fixed up to time instant  $\text{min}_{t_S}$ . The union can consequently output all elements with a start timestamp  $\leq \text{min}_{t_S}$ .

**Remark 11.2.** Recall that the operators in our physical operator algebra rely on a multi-threaded, push-based processing paradigm. Operators can be connected directly, i. e., without an inter-operator queue. Hence, it is not possible to consume elements across multiple operator input queues in a sorted manner as done in other DSMSs [ABW06, CCC<sup>+</sup>02, HMA<sup>+</sup>04]. Instead of performing a sort-merge step at the input of an *n*-ary operator, we employ a priority queue at the operator's output to satisfy the ordering requirement of the output stream.

**Example 11.3.** Table 11.4 displays a possible output stream of the union over streams  $S_1$  and  $S_2$  (see page 73). The output stream is in ascending order by start timestamps. Since we do not enforce an order for elements having identical start timestamps, it is possible to generate several correct output streams. Be aware that the order of elements with identical start timestamps in the output stream depends on element arrival and the operator implementation.

#### 11.5.4 Cartesian Product / Theta-Join

Algorithm 6 computes a theta-join over two physical streams. The join state consists of two SweepAreas, one for each input stream, and a min-priority queue at the output to

$S_1 \cup S_2$		
Tuple	$t_S$	$t_E$
c	1	8
b	1	7
d	3	9
a	4	5
a	5	11
d	6	14
b	7	15
a	9	10
e	10	18
b	12	17

Table 11.4: Union of physical streams  $S_1$  and  $S_2$ 

order the join results. We summarized the Cartesian product and theta-join in a single algorithm as the sole difference is the parameterization of the SweepAreas, in particular the query predicate.

### SweepArea Parameters

The initialization fragment of the algorithm defines the parameters for the SweepAreas (see also Table 11.5). The internal *order relation* of both SweepAreas is set to  $\leq_{t_E}$ . The order relation  $\leq_{t_E}$  on physical stream elements is the less-than-or-equal-to relation on *end timestamps*, which is a total order.

Let  $s := (e, [t_S, t_E])$  be the incoming stream element and  $\hat{s} := (\hat{e}, [\hat{t}_S, \hat{t}_E])$  denote an element from a SweepArea. The SweepAreas of the join algorithm are parameterized by the following two predicates.

$$p_{query}^\theta(s, \hat{s}) := \begin{cases} \text{true} & \text{if } \theta(e, \hat{e}) \wedge [t_S, t_E] \cap [\hat{t}_S, \hat{t}_E] \neq \emptyset, \\ \text{false} & \text{otherwise.} \end{cases}$$

The *query predicate* is evaluated in the method *query*. The choice of the SweepArea implementation and the definition of the query predicate have to ensure that the method *query* returns an iterator over all elements in the SweepArea qualifying for the join result. The query predicate takes the incoming element and an element from the opposite SweepArea as arguments. It checks that (i) their tuples qualify the join predicate  $\theta$ , and (ii) their time intervals overlap.

$$p_{remove}(s, \hat{s}) := \begin{cases} \text{true} & \text{if } t_S \geq \hat{t}_E, \\ \text{false} & \text{otherwise.} \end{cases}$$

The method *purgeElements* removes all elements from the SweepArea that fulfill the *remove predicate*  $p_{remove}$ . The remove predicate guarantees that only those elements are

Operations	Order Relation	Query Predicate	Remove Predicate
Theta-Join	$\leq_{t_E}$	$\theta(e, \hat{e}) \wedge [t_S, t_E) \cap [\hat{t}_S, \hat{t}_E) \neq \emptyset$	$t_S \geq \hat{t}_E$
Duplicate Elimination	$\leq_{t_E}$	$e = \hat{e} \wedge [t_S, t_E) \cap [\hat{t}_S, \hat{t}_E) \neq \emptyset$	$t_S \geq \hat{t}_E$
Difference	$\leq_{t_S}$	$e = \hat{e} \wedge [t_S, t_E) \cap [\hat{t}_S, \hat{t}_E) \neq \emptyset$	$t_S \geq \hat{t}_E$
Scalar Aggregation	$\leq_{t_S}$	$[t_S, t_E) \cap [\hat{t}_S, \hat{t}_E) \neq \emptyset$	$t_S \geq \hat{t}_E$
Grouping with Aggregation	$\leq_{t_S}$	$[t_S, t_E) \cap [\hat{t}_S, \hat{t}_E) \neq \emptyset$	$t_S \geq \hat{t}_E$
Count-based Window	$\leq_{t_S}$	—	$SA.size() = N$
Partitioned Window	$\leq_{t_S}$	$SA.size() = N$	—
Split	$\leq_{t_E}$	$true$	$t_S \geq \hat{t}_E$
Coalesce	$\leq_{t_S}$	$e = \hat{e} \wedge \hat{t}_E = t_S$	$t_S > \hat{t}_E \vee \hat{t}_E - \hat{t}_S \geq l_{max}$

Table 11.5: Overview of parameters used to tailor the ADT SweepArea to specific operators;  
 let  $(e, [t_S, t_E])$  and  $(\hat{e}, [\hat{t}_S, \hat{t}_E])$  be the first and second argument of the predicates respectively

---

**Algorithm 6:** Cartesian Product ( $\times$ ) / Theta-Join ( $\bowtie_\theta$ )

---

**Input** : physical streams  $S_{in_1}, S_{in_2}$ ; join predicate  $\theta$   
**Output** : physical stream  $S_{out}$

```

1  $S_{out} \leftarrow \emptyset;$ 
2 Let  $SA_1, SA_2$  be the empty SweepAreas( $\leq_{t_E}, p_{query}^\theta, p_{remove}$ );
3  $t_{S_1}, t_{S_2}, min_{t_S} \in T \cup \{\perp\}; t_{S_1} \leftarrow \perp; t_{S_2} \leftarrow \perp; min_{t_S} \leftarrow \perp;$ 
4 Let  $Q$  be an empty min-priority queue with priority  $\leq_{t_S}$ ;
5  $j, k \in \{1, 2\};$ 
6 foreach  $s := (e, [t_S, t_E]) \leftarrow S_{in_j}$  do
7    $k \leftarrow j \bmod 2 + 1;$ 
8    $SA_k.\text{purgeElements}(s, j);$ 
9    $SA_j.\text{insert}(s);$ 
10  Iterator  $qualifies \leftarrow SA_k.\text{query}(s, j);$ 
11  while  $qualifies.\text{hasNext}()$  do
12    Element  $(\hat{e}, [\hat{t}_S, \hat{t}_E]) \leftarrow qualifies.\text{next}();$ 
13    if  $j = 1$  then  $Q.\text{insert}((e \circ \hat{e}, [t_S, t_E] \cap [\hat{t}_S, \hat{t}_E]));$ 
14    else  $Q.\text{insert}((\hat{e} \circ e, [t_S, t_E] \cap [\hat{t}_S, \hat{t}_E]));$ 
15     $t_{S_j} \leftarrow t_S;$ 
16     $min_{t_S} \leftarrow \min(t_{S_1}, t_{S_2});$ 
17    if  $min_{t_S} \neq \perp$  then
18       $\text{TRANSFER}(Q, min_{t_S}, S_{out});$ 
19 while  $\neg Q.\text{isEmpty}()$  do
20   $Q.\text{extractMin()} \leftarrow S_{out};$ 

```

---

dropped that will not participate in any future join results. Here, the condition checks a time interval overlap between an element  $\hat{s}$  in the SweepArea and the incoming stream element  $s$ . If the start timestamp of  $s$  is equal to or greater than the end timestamp of  $\hat{s}$ , the respective time intervals of both elements cannot overlap. Hence, this pair of elements does not qualify as join result. For all other cases, an overlap in terms of time intervals is still possible. As a consequence, these elements need to be retained.

### Choice of SweepArea Implementation

The SweepArea implementation has to be chosen according to the join predicate. Using an inappropriate SweepArea implementation might produce incorrect join results. While list-based SweepArea implementations (nested-loops join) are suitable for arbitrary join predicates, hash-based variants are restricted to equi-joins. The Cartesian Product can be expressed as a special nested-loops join where join predicate  $\theta$  always returns true. Note that even for the Cartesian product the query predicate still has to verify time interval overlap.

- *Symmetric Hash Join (SHJ)*. Each SweepArea is implemented as a hash table (primary data structure). The elements in the hash table are additionally linked in ascending order by their end timestamps (priority queue as secondary data structure). Inserting a stream element into a SweepArea means (i) to put it into the hash table based on a hash function, and (ii) to adjust the linkage. The hash function must map value-equivalent elements into the same bucket. Whenever a SweepArea is probed, the hash function determines the bucket containing the qualifying elements. The linkage is used for purging elements effectively from the SweepArea. The method *purgeElements* follows the linkage and discards elements as long as their end timestamp is less than or equal to the start timestamp of the incoming element.
- *Symmetric Nested-Loops Join (SNJ)*. Each SweepArea is implemented as an ordered list that links elements in ascending order by end timestamps. Insertion has to preserve the order. Probing the SweepArea is a sequential scan of the linked list. Expired elements are removed from the SweepArea by a traversal of the list that stops at the first element with an end timestamp greater than the start timestamp of the incoming element.
- *Asymmetric Variants*. With our SweepArea framework it is also possible to construct and take advantage of the *asymmetric join variants* described in [KNV03]. In that case, the two previous implementation alternatives for SweepAreas are combined for state maintenance. This means, the join utilizes one SHJ and one SNJ SweepArea.

### Description of Algorithm

Algorithm 6 adapts the ripple join technique [HH99] to data-driven query processing [WA91]. For each element arriving from input stream  $S_{in_j}$ , the following steps are performed:

1. All elements satisfying predicate  $p_{remove}$  are purged from the opposite SweepArea  $SA_k$  (see line 8).
2. The incoming element is inserted into SweepArea  $SA_j$  (see line 9).
3. The opposite SweepArea  $SA_k$  is probed with predicate  $p_{query}$  (see line 10). The returned iterator contains all elements from  $SA_k$  that qualify as join result.
4. For each element returned by the iterator, a result is produced. The tuple component of a join result is the concatenation of the tuples obtained from the iterator's next element and the incoming element. Depending on which input stream the incoming elements comes from, the arguments of the concat-function are swapped to conform with the type-sensitive definition given in Section 8.1.4. The time interval attached to a join result is the intersection of the involved time intervals (see lines 13 and 14).
5. All join results are inserted into the min-priority queue in order to ensure the output stream ordering requirement. Elements are extracted from the queue and appended to the output stream if their start timestamp is less than or equal to  $\min_{t_S}$ , the minimum over the start timestamps of the most recent element from every input stream (see call to procedure TRANSFER in line 18).

The last part of the algorithm handles the finite case in which all elements of the input streams have been processed already. The while-loop (see line 19) transfers the remaining elements from queue  $Q$  to the output stream.

$S_1 \bowtie S_2$		
Tuple	$t_S$	$t_E$
$d$	6	9
$b$	12	15

Table 11.6: Equi-join of physical streams  $S_1$  and  $S_2$

**Example 11.4.** The results of an equi-join over streams  $S_1$  and  $S_2$  (see page 73) are listed in Table 11.6. Two stream elements qualify if their tuples are equal and their time intervals overlap. The join result is composed of the tuple and the intersection of the time intervals. Here, the concatenation function  $\circ$  is a projection to the first argument.

### 11.5.5 Duplicate Elimination

Algorithm 7 eliminates any time interval overlap for value-equivalent elements from the input stream. The operator state consists of a SweepArea and a min-priority queue. The SweepArea keeps all elements that already contributed to a result and whose time interval might overlap with that of future incoming elements. The min-priority queue is responsible for the order of the output stream.

**Algorithm 7:** Duplicate Elimination ( $\delta$ )

---

**Input** : physical stream  $S_{in}$   
**Output** : physical stream  $S_{out}$

```

1  $S_{out} \leftarrow \emptyset;$ 
2 Let  $SA$  be the empty SweepArea( $\leq_{t_E}$ ,  $p_{query}$ ,  $p_{remove}$ );
3 Let  $Q$  be an empty min-priority queue with priority  $\leq_{t_s}$ ;
4 foreach  $s := (e, [t_S, t_E]) \leftarrow S_{in}$  do
5    $SA.purgeElements(s, 1);$ 
6   TRANSFER( $Q$ ,  $t_S$ ,  $S_{out}$ );
7   Iterator  $qualifies \leftarrow SA.query(s, 1);$ 
8   if  $\neg qualifies.hasNext()$  then
9      $s \leftrightarrow S_{out};$ 
10     $SA.insert(s);$ 
11   else
12     Element  $(\hat{e}, [\hat{t}_S, \hat{t}_E]) \leftarrow qualifies.next();$ 
13     if  $\hat{t}_E < t_E$  then
14        $SA.replace((\hat{e}, [\hat{t}_S, \hat{t}_E]), (e, [t_E, t_E]));$ 
15        $Q.insert((e, [\hat{t}_E, t_E]));$ 
16 while  $\neg Q.isEmpty()$  do
17    $Q.extractMin() \leftrightarrow S_{out};$ 

```

---

**SweepArea Parameters and Implementation**

Table 11.5 shows that the parameters for the SweepArea are the same as specified for the join, except for the query predicate. Instead of an arbitrary predicate  $\theta$ , value-equivalence is checked.

$$p_{query}(s, \hat{s}) := \begin{cases} true & \text{if } e = \hat{e} \wedge [t_S, t_E] \cap [\hat{t}_S, \hat{t}_E] \neq \emptyset, \\ false & \text{otherwise.} \end{cases}$$

A hash-based SweepArea is suitable to efficiently detect value-equivalent elements. Thus, the SweepArea implementation is identical to the one proposed for the SHJ.

**Description of Algorithm**

For each element  $s := (e, [t_S, t_E])$  arriving from the input stream, the following steps are performed:

1. The SweepArea is cleaned by a call of the method *purgeElements* (line 5). Based on the remove predicate all elements in the SweepArea with an end timestamp less than or equal to  $t_S$  are discarded.
2. As long as the start timestamp of the top element in the min-priority queue is less than or equal to  $t_S$ , the top element is extracted from the queue and appended to the output stream (see line 6).

3. In line 7, the SweepArea is probed for elements satisfying predicate  $p_{query}$ , i. e., an element is returned when (i) it is value-equivalent to  $s$  and (ii) its time interval overlaps with  $[t_S, t_E]$ .
4. If the probing does not return any element,  $s$  is considered unique and thus a result, which is appended to  $S_{out}$  (see lines 8 and 9). In addition,  $s$  is inserted into the SweepArea with the aim to identify future incoming elements as duplicates (see line 10).
5. If there exists an element  $\hat{s} := (\hat{e}, [\hat{t}_S, \hat{t}_E])$  in the SweepArea that satisfies the query predicate, i. e., it represents a duplicate for  $s$ , the iterator *qualifies* will return this element. Due to the order of the input stream,  $\hat{t}_S \leq t_S$ . The following two cases can occur: (i) For  $\hat{t}_E \geq t_E$ , nothing has to be updated because the time interval  $[t_S, t_E]$  is fully covered by  $[\hat{t}_S, \hat{t}_E]$ . (ii) For  $\hat{t}_E < t_E$ , the time interval of the incoming element contains time instants not yet covered by the SweepArea. For this reason, we have to update the SweepArea. We substitute  $(\hat{e}, [\hat{t}_S, \hat{t}_E])$  for  $(e, [\hat{t}_E, t_E])$  (see line 14). The time interval  $[\hat{t}_E, t_E]$  corresponds to the new time interval fragment supplied by the incoming element  $s$ .
6. The new element  $(e, [\hat{t}_E, t_E])$  is a result but it cannot be appended to the output stream instantly as still further stream elements might arrive with a start timestamp less than  $\hat{t}_E$ . To preserve the start timestamp order of the output stream, the new element is inserted into the priority queue (see line 15).

In the case of a finite input stream, the final stage is that all elements of the queue are appended to the output stream (see lines 16 and 17).

$\delta(S_1)$		
Tuple	$t_S$	$t_E$
$c$	1	8
$a$	5	11
$d$	6	14
$b$	12	17

Table 11.7: Duplicate elimination over physical stream  $S_1$

**Example 11.5.** Table 11.7 shows the output stream generated by a duplicate elimination over input stream  $S_1$  (see page 73). Element  $(a, [9, 10])$  is dropped because prior to its arrival, element  $(a, [5, 11])$  was appended to the output stream. Element  $(a, [9, 10])$  represents a duplicate because it is value-equivalent to  $(a, [5, 11])$  and the interval  $[9, 10] \subset [5, 11]$ .

### 11.5.6 Difference

The duplicate-sensitive difference operator (see Algorithm 8) subtracts input stream  $S_{in_2}$  from  $S_{in_1}$ . A subtraction takes place whenever an element from  $S_{in_1}$  is value-equivalent to an element from  $S_{in_2}$  and the time intervals of both elements overlap. In order to have

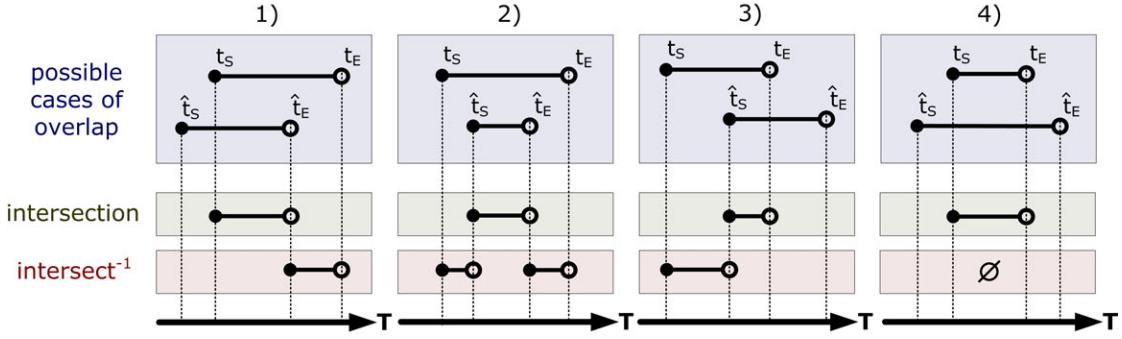


Figure 11.1: Inverse intersection

control over duplicates, the algorithm is designed in a symmetric manner. While the symmetric difference is computed internally ( $S_{in_1} - S_{in_2}$  and  $S_{in_2} - S_{in_1}$ ), only the results of  $S_{in_1} - S_{in_2}$  are appended to the output stream. The operator state consists of two SweepAreas. The first SweepArea manages elements involved in the difference  $S_{in_1} - S_{in_2}$ , whereas the second SweepArea manages those involved in the difference  $S_{in_2} - S_{in_1}$ .

### Inverse Intersection

The difference algorithm uses the auxiliary function  $\text{intersect}^{-1}$  to subtract two time intervals from each other. Let  $[t_S, t_E)$  and  $[\hat{t}_S, \hat{t}_E)$  be two time intervals. The function  $\text{intersect}^{-1} : (T \times T) \times (T \times T) \rightarrow \wp(T \times T)$  computes all subintervals of  $[t_S, t_E)$  with maximum size that do no overlap with  $[\hat{t}_S, \hat{t}_E)$ . Formally,

$$\text{intersect}^{-1}([t_S, t_E), [\hat{t}_S, \hat{t}_E)) := \begin{cases} \{[\hat{t}_E, t_E)\} & \text{if } \hat{t}_S \leq t_S \wedge \hat{t}_E < t_E, \\ \{[t_S, \hat{t}_S), [\hat{t}_E, t_E)\} & \text{if } \hat{t}_S > t_S \wedge t_E > \hat{t}_E, \\ \{[t_S, \hat{t}_S)\} & \text{if } \hat{t}_S > t_S \wedge t_E \leq \hat{t}_E, \\ \emptyset & \text{if } [t_S, t_E) \cap [\hat{t}_S, \hat{t}_E) = \emptyset \vee \\ & (\hat{t}_S \leq t_S \wedge \hat{t}_E \geq t_E). \end{cases} \quad (11.1)$$

Figure 11.1 illustrates the possible cases and results of function  $\text{intersect}^{-1}$ .

### SweepArea Parameters and Implementation

While the query and remove predicate is equal for both SweepAreas, their order relation and implementation differs. As shown in Table 11.5, the query and remove predicate are the same as for the duplicate elimination. Both SweepAreas rely on a hash-based implementation as described for the SHJ. However, the linkage of elements inside the hash tables differs. The first SweepArea links the elements inside the hash table in ascending order by start timestamps (order relation  $\leq_{t_S}$ ), whereas the second SweepArea links them in ascending order by end timestamps (order relation  $\leq_{t_E}$ ). Besides the global element linkage across buckets, the difference requires an additional local linkage according to

**Algorithm 8:** Difference (-)

---

**Input** : physical streams  $S_{in_1}, S_{in_2}$   
**Output** : physical stream  $S_{out}$

```

1  $S_{out} \leftarrow \emptyset;$ 
2 Let  $SA_1$  be an empty SweepArea( $\leq_{ts}, p_{query}, p_{remove}$ );
3 Let  $SA_2$  be an empty SweepArea( $\leq_{t_E}, p_{query}, p_{remove}$ );
4  $t_{S_1}, t_{S_2}, min_{ts} \in T \cup \{\perp\}; t_{S_1} \leftarrow \perp; t_{S_2} \leftarrow \perp; min_{ts} \leftarrow \perp;$ 
5  $j, k \in \{1, 2\};$ 
6 foreach  $s := (e, [t_S, t_E]) \leftarrow S_{in_j}$  do
7    $k \leftarrow j \bmod 2 + 1;$ 
8    $t_{S_j} \leftarrow t_S;$ 
9   Iterator  $qualifies \leftarrow SA_k.\text{query}(s, j);$ 
10  if  $\neg qualifies.\text{hasNext}()$  then  $SA_j.\text{insert}(s);$ 
11  else
12    List  $insertIntoSA_j \leftarrow \emptyset;$ 
13    List  $insertIntoSA_k \leftarrow \emptyset;$ 
14     $insertIntoSA_j.append(s);$ 
15    while  $insertIntoSA_j.\text{size}() > 0 \wedge qualifies.\text{hasNext}()$  do
16      Element  $\hat{s} := (\tilde{e}, [\tilde{t}_S, \tilde{t}_E]) \leftarrow qualifies.\text{next}();$ 
17      Iterator  $overlaps \leftarrow \text{all } (\tilde{e}, [\tilde{t}_S, \tilde{t}_E]) \in insertIntoSA_j \text{ where } [\tilde{t}_S, \tilde{t}_E] \cap [\tilde{t}_S, \tilde{t}_E] \neq \emptyset;$ 
18      if  $overlaps.\text{hasNext}()$  then
19        Element  $\tilde{s} := (\tilde{e}, [\tilde{t}_S, \tilde{t}_E]) \leftarrow overlaps.\text{next}();$ 
20         $overlaps.\text{remove}();$ 
21         $qualifies.\text{remove}();$ 
22        foreach  $\bar{s} := (\bar{e}, [\bar{t}_S, \bar{t}_E]) \in \{\tilde{e}\} \times \text{intersect}^{-1}([\tilde{t}_S, \tilde{t}_E], [\tilde{t}_S, \tilde{t}_E])$  do
23          if  $\bar{t}_E \leq \tilde{t}_S$  then  $SA_j.\text{insert}(\bar{s});$ 
24          else  $insertIntoSA_j.append(\bar{s});$ 
25        foreach  $\bar{s} \in \{\tilde{e}\} \times \text{intersect}^{-1}([\tilde{t}_S, \tilde{t}_E], [\tilde{t}_S, \tilde{t}_E])$  do  $insertIntoSA_k.append(\bar{s});$ 
26      foreach  $i \in \{1, 2\}$  do
27        foreach  $\tilde{s} \in insertIntoSA_i$  do
28           $SA_i.\text{insert}(\tilde{s});$ 
29       $min_{ts} \leftarrow \min(t_{S_1}, t_{S_2});$ 
30      if  $min_{ts} \neq \perp$  then
31        if  $j = 1$  then
32           $SA_2.\text{purgeElements}((e, [min_{ts}, t_E]), 1);$ 
33        else
34          Iterator  $results \leftarrow SA_1.\text{extractElements}((e, [min_{ts}, t_E]), 2);$ 
35          while  $results.\text{hasNext}()$  do
36             $results.\text{next}() \hookrightarrow S_{out};$ 
37  Iterator  $results \leftarrow SA_1.\text{iterator}();$ 
38  while  $results.\text{hasNext}()$  do
39     $results.\text{next}() \hookrightarrow S_{out};$ 

```

---

start timestamps inside the individual buckets. This causes the method *query* to return an iterator over qualifying elements in ascending order by start timestamps whenever the SweepArea is probed.

### Description of Algorithm

The basic processing concept resembles that of the join. Insertion takes place on the SweepArea associated with the input stream from which the incoming element arrives, whereas probing and purging is performed on the opposite SweepArea. Whenever an element arrives from an input stream  $S_{in_j}$ ,  $j \in 1, 2$ , the following instructions are carried out:

1. The opposite SweepArea  $SA_k$  is probed. The method *query* returns an iterator over all elements of the SweepArea that satisfy the query predicate, i. e., they are value-equivalent to the incoming element  $s$  and have a time interval overlap (see line 9). If no element satisfies these conditions, nothing has to be subtracted from the incoming element and it can be inserted into SweepArea  $SA_j$  (see line 10).
2. In cases where elements in  $SA_k$  satisfy the query predicate, these need to be subtracted from the incoming element. Lines 12 to 28 handle this case. First, two temporary lists are created, one for each SweepArea. List *insertIntoSAj* stores the remaining fragments of the incoming element to be eventually inserted into SweepArea  $SA_j$  after subtraction. Initially,  $s$  is inserted into the list *insertIntoSAj*. List *insertIntoSAk* contains the remaining fragments of elements delivered from iterator *qualifies* after subtracting them from the elements in *insertIntoSAj*.
3. As long as list *insertIntoSAj* contains elements and iterator *qualifies* has further elements, two elements can qualify for subtraction (see while-loop condition in line 15). Whenever an element  $\tilde{s}$  from list *insertIntoSAj* overlaps with the next element  $\hat{s}$  from iterator *qualifies*, both elements are removed from the underlying collections. This means,  $\tilde{s}$  is removed from list *insertIntoSAj* (line 20) and  $\hat{s}$  from SweepArea  $SA_k$  (line 21). As the difference only removes the overlapping interval fragments from both elements, we need to add the rest to the corresponding lists. The remaining fragments of both elements are computed by applying the auxiliary function *intersect*<sup>-1</sup> to their time intervals.  $\{\tilde{s}\} \times \text{intersect}^{-1}([\tilde{t}_S, \tilde{t}_E], [\hat{t}_S, \hat{t}_E])$  constructs the remaining fragments of  $\tilde{s}$  (see line 22). The remaining fragments of  $\hat{s}$  can be computed by swapping the arguments of function *intersect*<sup>-1</sup> (see line 25). The remaining fragments are inserted into the corresponding lists (see lines 24 and 25) apart from those for list *insertIntoSAj* that have an end timestamp  $\leq \hat{t}_S$ . These elements are directly inserted into SweepArea  $SA_j$  (see line 23) because the condition ensures that these will not participate in any further subtractions. Note that timestamp  $\hat{t}_S$  is the start timestamp of the latest element returned by iterator *qualifies*. According to the implementation, this iterator returns the qualifying elements of  $SA_k$  in ascending order by start timestamps.
4. If either list *insertIntoSAj* is empty or all elements of iterator *qualifies* have been processed, the elements in the temporary lists are inserted into the corresponding

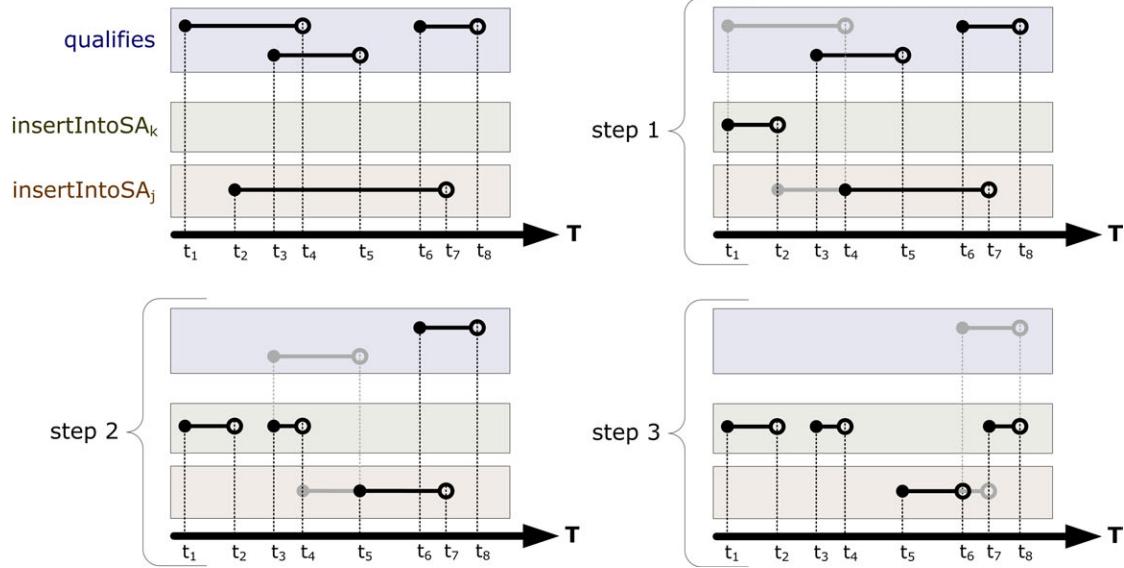


Figure 11.2: Computing the difference on time intervals

SweepAreas (see lines 26 – 28). At this point, it is guaranteed that any time interval overlap between qualifying elements from SweepArea  $SA_k$  and the incoming element has been eliminated.

5. The remove predicate ensures that only elements are deleted from the SweepArea that will not interact with any future incoming elements. Due to the start timestamp order of the input streams, an element  $\hat{s} := (\hat{e}, [\hat{t}_S, \hat{t}_E])$  in the SweepArea will not have any time interval overlap with a future incoming element if  $\hat{t}_E \leq \min_{t_S}$ , where  $\min_{t_S}$  is the minimum start timestamp over the most recent element delivered from each input stream. Element removal is triggered on the opposite SweepArea. If the incoming element belongs to the first input stream, SweepArea  $SA_2$  is purged of all elements for which predicate  $p_{remove}$  holds (see line 32). The linkage by end timestamps is utilized for this purpose. If the incoming element belongs to the second input stream, elements from SweepArea  $SA_1$  satisfying the remove predicate are extracted and appended to the output stream (see lines 34 – 36). The iterator  $results$  traverses the elements of SweepArea  $SA_1$  following the linkage by start timestamps and stops at the first element that violates the remove predicate. Recall that the method  $extractElements$  on SweepArea  $SA_1$  provides elements according  $\leq_{t_S}$ . Although it might be more efficient to purge expired elements from  $SA_1$  if elements in the SweepArea are linked by end timestamps (as done for  $SA_2$ ), the linkage by start timestamps is required here to guarantee the correct order of the output stream.

If all incoming stream elements have been processed, SweepArea  $SA_1$  may still contain some results, which have to be appended to the output stream to produce the complete result (see lines 37 to 39).

Figure 11.2 illustrates the stepwise progress of the *while*-loop by taking a closer look at the involved time intervals. It shows the time intervals of qualifying elements returned by the method *query*, and the two auxiliary lists *insertIntoSA<sub>k</sub>* and *insertIntoSA<sub>j</sub>*. Steps 1 to 3 demonstrate how the intervals of the qualifying elements are subtracted from the interval of the incoming element initially stored in list *insertIntoSA<sub>j</sub>*. Removed intervals are drawn in lighter grey.

**Property 11.1** (Iterator Ordering). Algorithm 8 is based on the invariant that the elements stored in the two temporary lists are disjoint in terms of time intervals. This invariant and thus correctness might be violated if the iterator returned by method *query* is not in ascending order by start timestamps.

$S_1 - S_2$		
Tuple	$t_s$	$t_e$
c	1	8
a	5	11
d	9	14
a	9	10
b	15	17

Table 11.8: Difference of physical streams  $S_1$  and  $S_2$

**Example 11.6.** The results of subtracting stream  $S_2$  from  $S_1$  (see page 73) are listed in Table 11.8. The difference subtracts value-equivalent stream elements in terms of time intervals, i. e., the interval belonging to an element of the second input stream is subtracted from elements of the first stream. In our example, the element  $(d, [9, 14])$  results from subtracting element  $(d, [3, 9])$  from  $(d, [6, 14])$ . Analogously,  $(b, [12, 17])$  minus  $(b, [7, 15])$  produces  $(b, [15, 17])$  in the output stream.

### 11.5.7 Scalar Aggregation

Algorithm 9 shows the implementation of the scalar aggregation. Stream processing demands a continuous output of aggregates [HHW97]. In our case, an *aggregate* is a physical stream element, i. e., it is composed of a tuple containing the aggregate value and a time interval. The state of the scalar aggregation consists of a SweepArea which maintains so-called *partial aggregates*. A partial aggregate is also a physical stream element, but the tuple component stores state information required to compute the final aggregate value. Partial aggregates are updated whenever their time interval intersects with that of an incoming stream element. The computation of a partial aggregate is finished if it is guaranteed that it will not be affected by any future stream elements. Then, the final aggregate value is computed from the tuple of the corresponding partial aggregate. The aggregate appended to the output stream is composed of the final aggregate value and the time interval of the corresponding partial aggregate.

## Aggregate Computation

Algorithm 9 takes three functions as input parameters to compute the results of an aggregate function ( $f_{agg}$ ). Types  $\mathcal{T}_1$  and  $\mathcal{T}_2$  match the ones of the corresponding logical operator and specify the input and output type respectively. The new type  $\mathcal{T}_3$  defines the type of partial aggregates, i. e., the type of tuples in the SweepArea.

- The *initializer*  $f_{init} : \Omega_{\mathcal{T}_1} \rightarrow \Omega_{\mathcal{T}_3}$  is applied to a tuple of an incoming stream element to instantiate the tuple of a partial aggregate.
- The *merger*  $f_{merge} : \Omega_{\mathcal{T}_3} \times \Omega_{\mathcal{T}_1} \rightarrow \Omega_{\mathcal{T}_3}$  updates the state information of a partial aggregate. This is accomplished by merging the tuple of an incoming element with the tuple of a partial aggregate in the SweepArea. The result is a new tuple which contains the updated state information.
- The *evaluator*  $f_{eval} : \Omega_{\mathcal{T}_3} \rightarrow \Omega_{\mathcal{T}_2}$  takes the tuple of a partial aggregate and computes the final aggregate value.

**Property 11.2** (Expressiveness). “These three functions can easily be derived for the basic SQL aggregates; in general, any operation that can be expressed as commutative applications of a binary function is expressible.” [MFHH02]

**Remark 11.3.** Although our approach to manage tuples of partial aggregates in the SweepArea resembles the basic aggregation concept proposed for sensor networks in [MFHH02], we extended it towards the temporal semantics addressed in this work. Bear in mind that our aggregates not only consist of an aggregate value but are also equipped with a time interval. User-defined aggregates [LWZ04] follow a similar pattern for aggregate computation. A user-defined aggregate is a procedure written in SQL, which is grouped into three blocks labeled: INITIALIZE, ITERATE, and TERMINATE. The state of the aggregate is stored in local tables. Like our three functions, these blocks are used to initialize and update the state of the aggregate, and to compute its final value.

## State of Partial Aggregates

Aggregate functions can be classified into three basic categories [GCB<sup>+</sup>97, MFHH02]: *distributive*, *algebraic*, and *holistic*. Depending on the category, the size requirements to store the internal state of partial aggregates varies. Recall that the tuple of a partial aggregate encapsulates the required state information. In the following, we discuss how to represent tuples of partial aggregates to meet the requirements of the different categories of aggregate functions.

- For *distributive aggregate functions* like MIN, MAX, SUM, and COUNT, a tuple of an element in the SweepArea is either a single tuple from the input stream or a natural number. The tuple corresponds to the aggregate value. Hence, type  $\Omega_{\mathcal{T}_3}$  is equal to  $\Omega_{\mathcal{T}_2}$  and function  $f_{eval}$  is the identity.
- For *algebraic aggregate functions* such as AVERAGE, STANDARD DEVIATION, VARIANCE, and TOP-K, a tuple in the SweepArea needs to contain more information than a

single value. However, the total state information stored in a tuple of a partial aggregate is of constant size. For instance, it is necessary to keep the sum and the count for the average. The final aggregate value results from dividing the sum by the count.

- For *holistic aggregate functions* like MEDIAN and COUNT DISTINCT, the state cannot be restricted to a constant size. As a consequence, the tuple of an element in the SweepArea is a data structure, e. g., a list, set, or even histogram, built over tuples from the input stream. For COUNT DISTINCT, the tuple could be a hash set, which implicitly eliminates duplicates. The size of this set would be the final aggregate value.

**Example 11.7** (Average). Let us consider the SQL aggregate function AVG. The tuples for partial aggregates are pairs  $(sum, count)$  with schema (DOUBLE, INTEGER). Let  $e$  be the tuple of an incoming stream element, and let  $(s, c)$  be the tuple of a partial aggregate. Then, the three functions used to compute the aggregate value would be defined as follows:

$$\begin{aligned} f_{init}(e) &:= (e, 1) \\ f_{merge}((s, c), e) &:= (s + e, c + 1) \\ f_{eval}((s, c)) &:= s/c \end{aligned}$$

## SweepArea Parameters and Implementation

While the remove predicate is equal to that of the previous operators, the query predicate merely checks time interval overlap.

$$p_{query}(s, \hat{s}) := \begin{cases} \text{true} & \text{if } [t_S, t_E] \cap [\hat{t}_S, \hat{t}_E] \neq \emptyset, \\ \text{false} & \text{otherwise.} \end{cases}$$

Algorithm 9 requires the iterators returned by the methods *query* and *extractElements* to be sorted in ascending order by start timestamps. Hence, the order relation of the SweepArea is  $\leq_{t_S}$ . The implementation of the SweepArea is an ordered list, e. g., a randomized skip list [Pug90]. Because the method *query* does not need to check value-equivalence but only interval overlap, further implementations supporting range queries over interval data might be preferable, e. g., a dynamic variant of the priority search tree [McC85].

## Construction of Partial Aggregates

Whenever the interval of a partial aggregate  $(\hat{e}, [\hat{t}_S, \hat{t}_E])$  in the SweepArea intersects with the time interval  $[t_S, t_E]$  of the incoming element, an update on the SweepArea has to be performed. Thereby, the interval  $[\hat{t}_S, \hat{t}_E]$  is partitioned into contiguous subintervals of maximum length that either do not overlap with  $[t_S, t_E]$  or entirely overlap with  $[t_S, t_E]$ . Each of these new time intervals is assigned with a tuple. Both together form a new partial aggregate. Figure 11.3 shows the different cases of overlap and the respective computation of partial aggregates. We distinguish between two top-level cases:

---

**Algorithm 9:** Scalar Aggregation ( $\alpha_{f_{agg}}$ )

---

**Input** : physical stream  $S_{in}$ ; functions  $f_{init}, f_{merge}, f_{eval}$   
**Output** : physical stream  $S_{out}$

```

1  $S_{out} \leftarrow \emptyset;$ 
2 Let  $SA$  be the empty SweepArea( $\leq_{ts}, p_{query}, p_{remove}$ );
3 foreach  $s := (e, [t_S, t_E]) \leftarrow S_{in}$  do
4   Iterator  $qualifies \leftarrow SA.\text{query}(s, 1);$ 
5   if  $\neg qualifies.\text{hasNext}()$  then
6      $| SA.\text{insert}((f_{init}(e), [t_S, t_E]));$ 
7   else
8      $T last_{t_E} \leftarrow t_S;$ 
9     while  $qualifies.\text{hasNext}()$  do
10    Element  $(\hat{e}, [\hat{t}_S, \hat{t}_E]) \leftarrow qualifies.\text{next}();$ 
11     $qualifies.\text{remove}();$ 
12    if  $\hat{t}_S < t_S$  then
13       $| SA.\text{insert}((\hat{e}, [\hat{t}_S, t_S]));$ 
14      if  $t_E < \hat{t}_E$  then
15         $| | SA.\text{insert}((f_{merge}(\hat{e}, e), [t_S, t_E] \cap [\hat{t}_S, \hat{t}_E]));$ 
16         $| | SA.\text{insert}((\hat{e}, [t_E, \hat{t}_E]));$ 
17      else  $SA.\text{insert}((f_{merge}(\hat{e}, e), [t_S, \hat{t}_E]));$ 
18    else
19      if  $[\hat{t}_S, \hat{t}_E] = [t_S, t_E] \cap [\hat{t}_S, \hat{t}_E]$  then
20         $| | SA.\text{insert}((f_{merge}(\hat{e}, e), [\hat{t}_S, \hat{t}_E]));$ 
21      else
22         $| | SA.\text{insert}((f_{merge}(\hat{e}, e), [\hat{t}_S, t_E]));$ 
23         $| | SA.\text{insert}((\hat{e}, [t_E, \hat{t}_E]));$ 
24       $last_{t_E} \leftarrow \hat{t}_E;$ 
25      if  $last_{t_E} < t_E$  then  $SA.\text{insert}((f_{init}(e), [last_{t_E}, t_E]));$ 
26   Iterator  $results \leftarrow SA.\text{extractElements}(s, 1);$ 
27   while  $results.\text{hasNext}()$  do
28     Element  $(\hat{e}, [\hat{t}_S, \hat{t}_E]) \leftarrow results.\text{next}();$ 
29      $(f_{eval}(\hat{e}), [\hat{t}_S, \hat{t}_E]) \leftarrow S_{out};$ 
30   Iterator  $results \leftarrow SA.\text{iterator}();$ 
31   while  $results.\text{hasNext}()$  do
32     Element  $(\hat{e}, [\hat{t}_S, \hat{t}_E]) \leftarrow results.\text{next}();$ 
33      $(f_{eval}(\hat{e}), [\hat{t}_S, \hat{t}_E]) \leftarrow S_{out};$ 

```

---

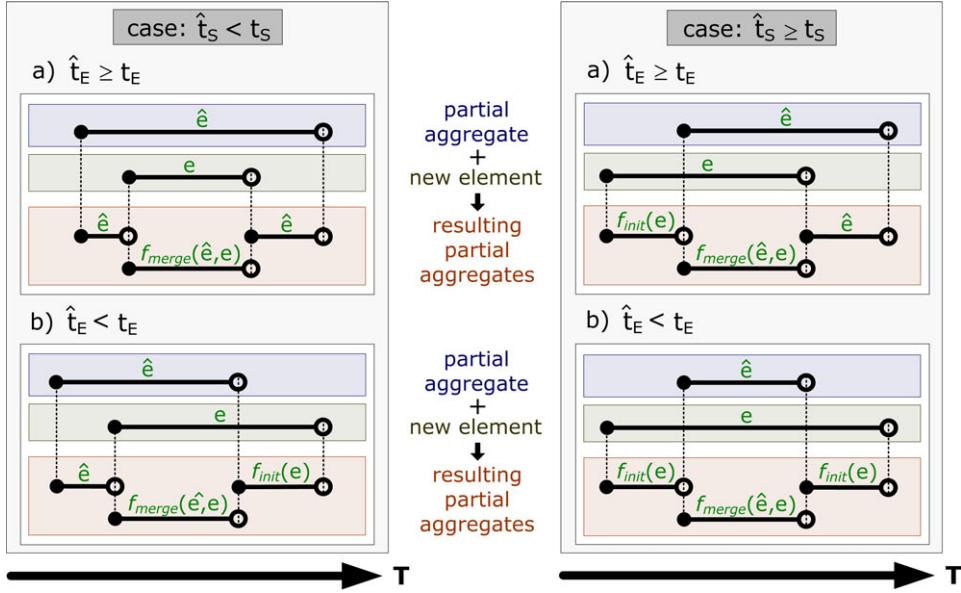


Figure 11.3: Incremental computation of aggregates from a partial aggregate  $(\hat{e}, [\hat{t}_S, \hat{t}_E])$  and an incoming element  $(e, [t_S, t_E])$

- For the case  $\hat{t}_S < t_S$ ,  $[t_S, t_E]$  is either contained in  $[\hat{t}_S, \hat{t}_E]$  or exceeds it at the right border.
- For the case  $\hat{t}_S \geq t_S$ ,  $[t_S, t_E]$  contains  $[\hat{t}_S, \hat{t}_E]$  or exceeds it at the left border.

The latter case has to be considered as our algorithm may generate time intervals with a start timestamp greater than the start timestamp  $t_S$  of the latest incoming element.

Depending on the above two cases, we create disjoint time intervals by separating the overlapping parts from the unique ones. Conceptually, tuples are built as follows:

1. The tuple  $\hat{e}$  is kept for the new partial aggregate if the generated time interval is a subinterval of  $[\hat{t}_S, \hat{t}_E]$  and does not overlap with  $[t_S, t_E]$ .
2. The tuple of the new partial aggregate is computed by  $f_{init}(e)$  if the generated time interval is a subinterval of  $[t_S, t_E]$  and does not overlap with  $[\hat{t}_S, \hat{t}_E]$ .
3. The tuple of the new partial aggregate is computed by invoking the aggregate function  $f_{merge}$  on the old tuple  $\hat{e}$  and the new tuple  $e$  if the generated time interval is the intersection of both time intervals.

For the special cases,  $\hat{t}_S = t_S$  and  $\hat{t}_E = t_E$  some intervals shown in Figure 11.3 may reduce to empty intervals. Although Figure 11.3 allows those cases for the ease of concise presentation, the design of Algorithm 9 prevents any (partial) aggregates with empty intervals entering the state. The interval construction for partial aggregates satisfies the following property:

**Property 11.3** (Interval Construction). The scalar aggregation eliminates any time interval overlap. As a consequence, the time intervals of aggregates in the output stream are disjoint. If a SweepArea contains multiple partial aggregates, their time intervals are consecutive, i. e., they do not overlap and there is no gap between them.

### Description of Algorithm

The scalar aggregation processes an incoming element as follows:

1. The SweepArea is probed. According to the query predicate, the method *query* returns an iterator over all elements in the SweepArea that have a time interval overlap with the incoming element (see line 4).
2. The state is updated. We distinguish between two cases: (i) If the iterator *qualifies* does not provide an element, a new partial aggregate is created by applying the aggregate function *f<sub>init</sub>* to tuple *e* (see line 6). (ii) For each partial aggregate  $(\hat{e}, [\hat{t}_S, \hat{t}_E])$  delivered by iterator *qualifies*, the state needs to be updated. This means we remove the old partial aggregate (see line 11) and insert new partial aggregates. The way new partial aggregates are built has been specified in the previous paragraph. Lines 12 – 23 correspond to the case differentiation (see also Figure 11.3).
3. Partial aggregates are extracted from the SweepArea by the method *extractElements* (see line 26). The iterator *results* traverses the SweepArea from its start in sequential order as long as the remove predicate holds. Recall that the SweepArea is a list of partial aggregates ordered by start timestamps. The remove predicate *p<sub>remove</sub>* verifies that only elements are discarded from the SweepArea and returned which cannot have a time interval overlap with future incoming elements. No updates will take effect on these partial aggregates any more, which means that their state computation is completed. The tuples of those partial aggregates are consequently ready for computing the final aggregate value.
4. Whenever the iterator *results* delivers a partial aggregate  $(\hat{e}, [\hat{t}_S, \hat{t}_E])$ , the final aggregate value is determined by *f<sub>eval</sub>(*ē*)*. The aggregate appended to the output stream (see line 29) consists of the final aggregate value and the original time interval  $[\hat{t}_S, \hat{t}_E]$ .<sup>1</sup> Due to the order of the iterator *results*, the output stream is ordered properly.

If  $S_{in}$  is finite and all elements have been processed, the SweepArea may still contain partial aggregates, which contain information about aggregates not yet generated. These partial aggregates are fetched by the method *iterator*. In analogy to processing the iterator returned by the method *extractElements* (see lines 26 – 29), the final aggregates are built and appended to the output stream (see lines 30 – 33). The output stream order is preserved since the method *iterator* follows the order relation of the SweepArea.

---

<sup>1</sup>For distributive aggregate functions, this final evaluation step can be omitted as the tuple of the partial aggregate represents the final aggregate value. However, algebraic and holistic aggregate functions necessitate this additional step.

**Variable**  $last_{t_E}$ 

Let us finally clarify the role of variable  $last_{t_E}$ . Before probing,  $last_{t_E}$  is initialized with  $t_S$  (see line 8). We consider two different cases where variable  $last_{t_E}$  becomes effective (see line 25): (i) If iterator *qualifies* does not return any elements, the initialization causes the creation of a new partial aggregate from the incoming element by setting the tuple to  $f_{init}(e)$  and the interval to the one of the incoming element. (ii) If iterator *qualifies* provides elements satisfying the query predicate,  $last_{t_E}$  stores the end timestamp of the most recent element delivered by the iterator (see line 24). Whenever the end timestamp of the last partial aggregate returned by iterator *qualifies* is less than the end timestamp of the incoming element, we generate a new partial aggregate by setting the tuple to  $f_{init}(e)$  and the interval to the non-overlapping interval fragment  $[last_{t_E}, t_E]$ .

$\alpha_{\text{SUM}}(S_1)$		
Tuple	$t_S$	$t_E$
$c$	1	5
$c + a$	5	6
$c + a + d$	6	8
$a + d$	8	9
$a + d + a$	9	10
$a + d$	10	11
$d$	11	12
$d + b$	12	14
$b$	14	17

Table 11.9: Scalar aggregation over physical stream  $S_1$

**Example 11.8.** Table 11.9 contains the output stream of the scalar aggregation applied to stream  $S_1$  (see page 73) computing the sum. Because the aggregation eliminates any time interval overlap, the time intervals of the results are the disjoint segments of all intervals from the input stream. The aggregate value (tuple) of a result is the sum over all tuples from the input stream whose time interval intersects with the interval assigned to the result.

### 11.5.8 Grouping with Aggregation

While Algorithm 9 focuses on scalar aggregation, Algorithm 10 displays grouping with aggregation. For this algorithm, we had to tackle the problem that the number of groups is not known in advance and varies over time. Our solution is a generic and dynamic framework to (i) assign the elements of the input stream to disjoint groups, (ii) update the individual groups, and (iii) merge the results of the individual groups and append them to the output stream. This skeleton not only serves as the basis for Algorithm 10, but also for the implementation of the partitioned window (see Section 11.6.3).

The state of the grouping with aggregation operator is a map data structure which associates a *group identifier* with a *SweepArea*. The group identifier indicates the group to

which an element belongs. The SweepArea associated with a group identifier serves as the underlying data structure to store the aggregates for this group.

### SweepArea Implementation and Parameters

Because a SweepArea is utilized to implement the aggregation for a single group, we can reuse the SweepArea implementation and parameterization of the scalar aggregation without any modifications. Consequently, a SweepArea stores partial aggregates for a specific group and aggregate computation relies on the three functions  $f_{init}$ ,  $f_{merge}$ , and  $f_{eval}$  to model an aggregate function.

### Description of Algorithm

Algorithm 10 can be divided into the three basic parts mentioned above.

1. The grouping function  $f_{group}$  takes the tuple of an incoming element to determine the  $groupID$ . The  $groupID$  identifies the group to which an incoming element belongs (see line 7). The map  $groups$ , which is typically implemented as a hash map, stores a reference to the SweepArea for each  $groupID$ . If the map  $groups$  does not contain the returned  $groupID$ , this group does not exist and a new SweepArea is created (see line 11). Afterwards the new entry composed of the  $groupID$  and the new SweepArea is added to the map  $groups$ .
2. After the right SweepArea has been determined, the partial aggregates in this SweepArea need to be updated (see line 16). This is achieved by the auxiliary procedure `UPDATE`, which is an excerpt of Algorithm 9. Therefore, aggregate computation is done in the same way as for the scalar aggregation.
3. Based on the min-priority queue  $G$  all SweepAreas are identified that contain partial aggregates with a start timestamp less than or equal to the start timestamp of the incoming element. Partial aggregates that satisfy the remove predicate are extracted from those SweepAreas and the final aggregates are built (see lines 27 – 30). This involves the computation of the final aggregate values by invoking the function  $f_{eval}$  on the tuples of the partial aggregates. The concatenation function  $\circ$  enriches the final aggregate value with required grouping information. Thereafter, the final aggregate is inserted into the min-priority queue  $Q$  which collects the final aggregates of all groups and aligns them in start timestamp order. The procedure `TRANSFER` appends all elements from queue  $Q$  having a start timestamp less than or equal to  $min_{t_s}$  to the output stream (see line 33), where  $min_{t_s}$  is the oldest (smallest) start timestamp among partial aggregates over all SweepAreas referenced by the map  $groups$  (see line 24).

Lines 34 to 41 generate the remaining aggregates in the case that the entire input stream has been processed. The group identifiers of all SweepArea still holding partial aggregates are contained in queue  $G$ . By iterating over these SweepAreas, the final aggregates are constructed as described above, and inserted into the queue  $Q$ . Eventually, all elements from queue  $Q$  are appended to the output stream.

---

**Algorithm 10:** Grouping with Aggregation ( $\gamma_{f_{group}, f_{agg}}$ )

---

**Input** : physical stream  $S_{in}$ ; grouping function  $f_{group}$ ; functions  $f_{init}$ ,  $f_{merge}$ ,  $f_{eval}$   
**Output** : physical stream  $S_{out}$

```

1  $S_{out} \leftarrow \emptyset;$ 
2 Let  $groups$  be an empty map with entries  $\langle groupID, sweepArea \rangle$ ;
3 Let  $G$  be an empty min-priority queue over group identifiers with order  $\leq_{ts}$  on the element
   with the smallest start timestamp of each associated SweepArea;
4 Let  $Q$  be an empty min-priority queue over stream elements with priority  $\leq_{ts}$ ;
5  $min_{ts} \in T \cup \{\perp\}$ ;
6 foreach  $s := (e, [t_S, t_E]) \leftarrow S_{in}$  do
7   GroupIdentifier  $groupID \leftarrow f_{group}(e)$ ;
8   SweepArea  $SA$ ;
9   if  $groups.\text{containsKey}(groupID)$  then  $SA \leftarrow groups.get(groupID)$ ;
10  else
11     $SA \leftarrow \text{new SweepArea}(\leq_{ts}, p_{query}, p_{remove})$ ;
12     $groups.put(groupID, SA)$ ;
13  boolean  $emptySA$ ;
14  if  $SA.\text{size}() = 0$  then  $emptySA \leftarrow true$ ;
15  else  $emptySA \leftarrow false$ ;
16  UPDATE( $SA, s, f_{init}, f_{merge}$ );
17  if  $emptySA$  then  $G.\text{insert}(groupID)$ ;
18   $min_{ts} \leftarrow \perp$ ;
19  while  $\neg G.\text{isEmpty}()$  do
20     $groupID \leftarrow G.\text{min}()$ ;
21     $SA \leftarrow groups.get(groupID)$ ;
22    Element  $(\hat{e}, [\hat{t}_S, \hat{t}_E]) \leftarrow SA.\text{iterator}().\text{next}()$ ;
23    if  $min_{ts} \neq \perp \wedge min_{ts} = \hat{t}_S$  then break;
24     $min_{ts} \leftarrow \hat{t}_S$ ;
25    if  $min_{ts} \leq t_S$  then
26       $G.\text{extractMin}()$ ;
27      Iterator  $results \leftarrow SA.\text{extractElements}(s, 1)$ ;
28      while  $results.\text{hasNext}()$  do
29        Element  $(\hat{e}, [\hat{t}_S, \hat{t}_E]) \leftarrow results.\text{next}()$ ;
30         $Q.\text{insert}((groupID \circ f_{eval}(\hat{e}), [\hat{t}_S, \hat{t}_E]))$ ;
31        if  $SA.\text{size}() > 0$  then  $G.\text{insert}(groupID)$ ;
32      else break;
33    if  $min_{ts} \neq \perp$  then TRANSFER( $Q, min_{ts}, S_{out}$ );
34  while  $\neg G.\text{isEmpty}()$  do
35     $groupID \leftarrow G.\text{extractMin}()$ ;
36     $SA \leftarrow groups.get(groupID)$ ;
37    Iterator  $results \leftarrow SA.\text{iterator}()$ ;
38    while  $results.\text{hasNext}()$  do
39      Element  $(\hat{e}, [\hat{t}_S, \hat{t}_E]) \leftarrow results.\text{next}()$ ;
40       $Q.\text{insert}((groupID \circ f_{eval}(\hat{e}), [\hat{t}_S, \hat{t}_E]))$ ;
41  while  $\neg Q.\text{isEmpty}()$  do  $Q.\text{extractMin}() \leftarrow S_{out}$ ;

```

---

---

**Procedure** UPDATE(*SweepArea*  $SA$ , *element*  $s$ , *function*  $f_{init}$ , *function*  $f_{merge}$ )

---

Lines 4 to 25 of Algorithm 9;

---

### Priority Queue $G$

The min-priority queue  $G$  holds a group identifier for each non-empty group (*SweepArea*). Queue  $G$  aligns group identifiers in ascending order by the smallest start timestamp obtained from the corresponding SweepArea. Hence, the minimum in  $G$  is the *groupID* which belongs to the SweepArea that holds the partial aggregate with the smallest (oldest) start timestamp (see line 20). Because the SweepAreas are implemented as sorted lists, the partial aggregate with the smallest start timestamp is the one which is returned first by the method *iterator* (see line 22). The variable  $min_{t_S}$  is assigned this start timestamp. If  $min_{t_S}$  is less than or equal to the start timestamp of the incoming element, the function *extractElements* is called on the SweepArea to release partial aggregates whose computation is completed (see line 27). As a consequence, the value of the smallest start timestamp in this SweepArea usually increases.

The purpose of the *while*-loop is to release those partial aggregates from the SweepAreas whose computation is completed (see lines 19 – 32). Instead of calling *extractElements* on each SweepArea, which would be expensive, the queue  $G$  is used to determine suitable candidates. Those SweepAreas hold at least a single partial aggregate with a start timestamp less than or equal to the start timestamp of the incoming element  $t_S$ . The goal of extracting elements from those SweepAreas is to increase the minimum start timestamp  $min_{t_S}$ , which belongs to the partial aggregate with the smallest start timestamp across all SweepAreas. The *while*-loop terminates if either  $min_{t_S}$  becomes greater than  $t_S$  (see line 32), or it is not possible to further raise  $min_{t_S}$  (see line 23). The timestamp  $min_{t_S}$  controls the transfer of elements from queue  $Q$  to the output stream. Only elements with a start timestamp  $\leq min_{t_S}$  are appended to  $S_{out}$  because otherwise the desired order of the output stream might be violated.

**Remark 11.4.** Although it would be possible to restrict the method *extractElements* to the SweepArea representing the group of the incoming element, this approach delays the transfer of results if the incoming elements are not distributed uniformly over all groups. The reason is that only those elements from queue  $Q$  can be appended to  $S_{out}$  which have start timestamps  $\leq min_{t_S}$ . However,  $min_{t_S}$  denotes the minimum start timestamp over all SweepAreas. If  $min_{t_S}$  belongs to a group that does not receive any elements for a certain time period, no output would be produced during this timespan.

**Example 11.9.** Table 11.10 shows the output stream of the grouping with aggregation over stream  $S_1$  (see page 73). The grouping function  $f_{group}$  assigns elements of  $S_1$  to groups according to the character in their tuple. This means, all elements with tuple  $a$  belong to the same group, and so on. The aggregate function is COUNT. The output stream type is a composite type consisting of the input stream type (character for the group identifier) and the type of the aggregate values (positive integer for the count).

$\gamma_{f_{group}, \text{COUNT}}(S_1)$		
Tuple	$t_s$	$t_e$
(c, 1)	1	8
(a, 1)	5	9
(d, 1)	6	14
(a, 2)	9	10
(a, 1)	10	11
(b, 1)	12	17

 Table 11.10: Grouping with aggregation over physical stream  $S_1$ 

## 11.6 Window Operators

The *window operators* in our physical operator algebra take a *chronon* stream as input. Hence, the time intervals of incoming elements have to be chronons. The *split* operator is able to convert an arbitrary physical stream into a chronon stream (see Section 11.7.1). Every window operator assigns a new time interval to the tuple of an incoming element. The window type determines how interval starts and endings are set.

### 11.6.1 Time-Based Sliding Window

The time-based sliding window is a stateless operator that sets the validity of incoming elements to the window size  $w$  (see Algorithm 12). Recall that the window size represents a period in (application) time,  $w \in T$ ,  $w > 0$ .

---

**Algorithm 12:** Time-Based Sliding Window ( $\omega_w^{\text{time}}$ )

---

**Input** : chronon stream  $S_{in}$ ; window size  $w$   
**Output** : physical stream  $S_{out}$

```

1  $S_{out} \leftarrow \emptyset;$ 
2 foreach  $s := (e, [t_s, t_s + 1)) \hookleftarrow S_{in}$  do
3    $\lfloor (e, [t_s, t_s + w)) \hookrightarrow S_{out};$ 

```

---

### Description of Algorithm

For each incoming element, Algorithm 12 appends a new element to the output stream which has the same tuple as the incoming element but an extended validity, namely a time interval of length  $w$  chronons relative to the start timestamp of the incoming element. Since we use half-open time intervals to represent the validity of stream elements, an incoming element  $(e, [t_s, t_s + 1))$  is mapped to  $(e, [t_s, t_s + w))$ . The time-based sliding window does not modify the start timestamps, thereby it preserves the order of the input stream in the output stream.

(a) Chronon stream $S_3$			(b) $\omega_{50}^{\text{time}}(S_3)$		
Tuple	$t_S$	$t_E$	Tuple	$t_S$	$t_E$
$b$	1	2	$b$	1	51
$a$	3	4	$a$	3	53
$c$	4	5	$c$	4	54
$a$	7	8	$a$	7	57
$b$	10	11	$b$	10	60

 Table 11.11: Chronon stream  $S_3$  and time-based window of size 50 time units over  $S_3$ 

## Impact

The implementation nicely illustrates the effect of the time-based sliding window operator on a query plan. To comply with the semantics specified in Section 8.2.1, the time-based sliding window operator should slide a temporal window of size  $w$  over its input stream. This means for a chronon stream  $S_{in}$ , at a time instant  $t \in T$  all elements of  $S_{in}$  with a start timestamp  $t_S$ , where  $t - w + 1 \leq t_S \leq t$ , should be in the window. The physical time-based window achieves this effect by adjusting the validity of each incoming element according to the window size. Whenever the time-based window is installed in a query plan, downstream operators implicitly consider the new validities, i. e., the new time intervals. As stateful operators have to keep elements in their state as long as these are not expired, there is an apparent correlation between the window size and the size of the state. Be aware that the time-based window operator does not affect stateless operators such as filter and map because these do not regard time intervals (see Algorithms 2 and 3).

**Property 11.4** (Now-Window). Applying a time-based window with size  $w = 1$  to a chronon stream has no effects because the output stream will be identical to the input stream.

**Example 11.10.** Table 11.11(a) defines chronon stream  $S_3$ . Table 11.11(b) shows the results of the time-based sliding window applied to  $S_3$  with window size 50 time units. While the tuples and start timestamps of the input stream are retained in the output stream, the time-based sliding window sets the end timestamps according to the window size. Each end timestamp has an offset of  $w$  chronons relative to its corresponding start timestamp.

## 11.6.2 Count-Based Sliding Window

Algorithm 13 specifies the implementation of the count-based window. This window operator sets the end timestamp of an incoming element to the start timestamp of the  $N$ -th future incoming element, where  $N \in \mathbb{N}, N > 0$ , denotes the window size. The state consists of a SweepArea that keeps the  $N$  most recent elements of the input stream.

---

**Algorithm 13:** Count-Based Sliding Window ( $\omega_N^{\text{count}}$ )

---

**Input** : chronon stream  $S_{in}$ ; window size  $N$   
**Output** : physical stream  $S_{out}$

```

1  $S_{out} \leftarrow \emptyset;$ 
2 Let  $SA$  be the empty SweepArea( $\leq_{t_S}, -, p_{remove}^N$ );
3 foreach  $s := (e, [t_S, t_S + 1]) \leftarrow S_{in}$  do
4   Iterator  $results \leftarrow SA.\text{extractElements}(s, 1);$ 
5   if  $results.\text{hasNext}()$  then
6     Element  $(\hat{e}, [\hat{t}_S, \hat{t}_S + 1]) \leftarrow results.\text{next}();$ 
7      $(\hat{e}, [\hat{t}_S, t_S)) \leftarrow S_{out};$ 
8    $SA.\text{insert}(s);$ 
9 Iterator  $results \leftarrow SA.\text{iterator}();$ 
10 while  $results.\text{hasNext}()$  do
11   Element  $(\hat{e}, [\hat{t}_S, \hat{t}_S + 1]) \leftarrow results.\text{next}();$ 
12    $(\hat{e}, [\hat{t}_S, \infty)) \leftarrow S_{out};$ 

```

---

**SweepArea Implementation and Parameters**

The SweepArea is a simple linked list. The linkage of elements corresponds to their arrival order ( $\leq_{t_S}$ ). This means, the method *insert* adds new elements at the end, while the method *extractElements* removes elements at the beginning of the list. The remove predicate

$$p_{remove}^N(s, \hat{s}) := \begin{cases} \text{true} & \text{if } SA.\text{size}() = N, \\ \text{false} & \text{otherwise.} \end{cases}$$

verifies whether the SweepArea reached a size of  $N$  elements or not. The query functionality is unused, hence, no query predicate is specified.

**Description of Algorithm**

When a new stream element arrives, a call of the method *extractElements* checks if the SweepArea is already filled with  $N$  elements (see line 4). In this case, the first element is extracted from the SweepArea. From this element, a new element is created by setting the end timestamp to the start timestamp of the incoming element (see line 7). Then, the new element is appended to the output stream. The output stream order is correct since the SweepArea preserves the order of the input stream and start timestamps are not modified. Finally, the incoming element is inserted into the SweepArea (see line 8). After all elements of a finite stream have been processed, a new element is created for each residue in the SweepArea and appended to  $S_{out}$  (see lines 9 – 12). The corresponding end timestamps are set to  $\infty$ .

**Property 11.5** (SweepArea Size). The size of the SweepArea does not exceed  $N$  elements. This is an apparent implication deduced from the fact that one element is extracted from

the SweepArea prior to the insertion of an incoming element as soon as the SweepArea has been filled with  $N$  elements.

### Assumption

Count-based windows may not be appropriate whenever multiple elements in a stream can have the same start timestamp since ties have to be broken in some manner to ensure that the window contains exactly  $N$  elements. This case consequently causes count-based windows to be nondeterministic [ABW06]. To ensure deterministic query results, we allow count-based windows solely for cases where start timestamps in a physical stream are unique. This property can be inferred from the premise specified in Section 8.2.2. Note that this property holds for many real-world applications because raw streams, e. g., obtained from sensors, typically provide stream elements with a unique timestamp.

### Impact

A count-based sliding window should keep the most recent  $N$  elements in the window over an ordered stream. This semantics is implemented by setting the end timestamp of each input stream element to the start timestamp of the  $N$ -th future element. If the  $N$ -th future stream element does not exist in the case of a finite input stream, the end timestamp is set to  $\infty$ . Setting the end timestamps in this way affects the next stateful operator downstream of the window to hold the most recent  $N$  elements in the state because an element in the state can only expire on arrival of a new element and not until the state reached a size of  $N$  elements.

$\omega_1^{\text{count}}(S_3)$		
Tuple	$t_S$	$t_E$
$b$	1	3
$a$	3	4
$c$	4	7
$a$	7	10
$b$	10	$\infty$

Table 11.12: Count-based window of size 1 over chronon stream  $S_3$

**Example 11.11.** Table 11.12 shows the output stream of a count-based window with size 1 over chronon stream  $S_3$  (see page 98). Because at each time instant solely a single tuple is valid, stream  $S_3$  satisfies our prerequisite for an unambiguous and meaningful semantics of the count-based window. For a count-based window of size 1, an incoming stream element invalidates the previous stream element. Hence, it models a kind of update semantics.

### 11.6.3 Partitioned Window

The partitioned window (i) partitions the input stream into different substreams (groups), (ii) applies a count-based window of size  $N$  to each substream, and (iii) merges the windowed substreams to a single output stream. To implement this functionality, we reuse the dynamic grouping framework proposed for the grouping with aggregation (see Algorithm 10). The operator state consists of the map  $groups$  which associates a group identifier with a SweepArea maintaining the elements for the corresponding group. While the distribution of incoming elements among the groups is identical to Algorithm 10, updating and merging the individual groups differs. Algorithm 14 shows the implementation of the partitioned window.

#### SweepArea Implementation and Parameters

Because a SweepArea serves as the underlying data structure for the count-based window over a substream, the straightforward implementation would be the one presented in Algorithm 13. Here we use a more sophisticated implementation instead because elements from multiple SweepAreas have to be merged efficiently. For this reason, each SweepArea is a combined data structure built of two linked lists. The first list stores the stream elements, whereas the second lists consists of references to elements in the first list. The first list preserves the order of the input stream, i. e., elements are arranged in a FIFO manner. This order is required to implement the functionality of the count-based window. The method *query* relies on this list. The second list links the elements according to  $\leq_{ts}$ . This additional linkage helps to efficiently determine the element with the smallest start timestamp. The method *iterator* exploits this linkage. Be aware that the method *replace* has to adjust the linkage of both lists correctly (see line 30 in Algorithm 14). Contrary to the count-based window, the query predicate  $p_{query}^N$  checks if the size of the SweepArea is  $N$  and is defined analogously to the remove predicate in Algorithm 13 (see Table 11.5). Element removal is performed via the iterator returned by the method *query*. Therefore, this algorithm does not need a remove predicate.

#### Description of Algorithm

Algorithm 14 is structured as follows:

1. The SweepArea (group) for an incoming element is determined by the grouping function  $f_{group}$  (see lines 6 – 15). If no SweepArea exists for the computed group identifier, a new SweepArea is created and added to the map  $groups$  with key  $groupID$ .
2. The SweepArea needs to be updated (see lines 16 to 21). The method *query* checks whether the SweepArea already contains  $N$  elements or not. If it contains  $N$  elements, the first element is retrieved from the SweepArea. Next, a new element is built by setting the end timestamp to the start timestamp of the incoming element and inserted into queue  $Q$ . Thereafter, the original element is removed from the SweepArea. Independent of the size of the SweepArea, the incoming element is

---

**Algorithm 14:** Partitioned Window ( $\omega_{f_{group}, N}^{\text{partition}}$ )

---

**Input** : chronon stream  $S_{in}$ ; grouping function  $f_{group}$ ; window size  $N$ ; maximum delay  $\Delta$

**Output** : physical stream  $S_{out}$

```

1  $S_{out} \leftarrow \emptyset;$ 
2 Let  $groups$  be an empty map with entries  $\langle groupID, sweepArea \rangle$ ;
3 Let  $G$  be an empty min-priority queue over SweepAreas with order  $\leq_{t_s}$  on the element with
   the smallest start timestamp;
4 Let  $Q$  be an empty min-priority queue over stream elements with priority  $\leq_{t_s}$ ;
5 foreach  $s := (e, [t_s, t_s + 1]) \leftarrow S_{in}$  do
6   GroupIdentifier  $groupID \leftarrow f_{group}(e)$ ;
7   SweepArea  $SA$ ;
8   boolean  $emptySA$ ;
9   if  $groups.\text{containsKey}(groupID)$  then
10    |  $SA \leftarrow groups.get(groupID)$ ;
11    |  $emptySA \leftarrow false$ ;
12   else
13    |  $SA \leftarrow \text{new SweepArea}(\leq_{t_s}, p_{query}^N, -)$ ;
14    |  $groups.put(groupID, SA)$ ;
15    |  $emptySA \leftarrow true$ ;
16   Iterator  $results \leftarrow SA.\text{query}(s, 1)$ ;
17   if  $results.\text{hasNext}()$  then
18    | Element  $(\hat{e}, [\hat{t}_s, \hat{t}_s + 1]) \leftarrow results.\text{next}()$ ;
19    |  $Q.\text{insert}((\hat{e}, [\hat{t}_s, t_s]))$ ;
20    |  $results.\text{remove}()$ ;
21    $SA.\text{insert}(s)$ ;
22   if  $emptySA$  then  $G.\text{insert}(SA)$ ;
23   else
24    | Element  $(\hat{e}, [\hat{t}_s, \hat{t}_s + 1]) \leftarrow SA.\text{iterator}().\text{next}()$ ;
25    |  $G.\text{decreaseKey}(SA, \hat{t}_s)$ ;
26    $SA \leftarrow G.\text{first}()$ ;
27   Element  $(\hat{e}, [\hat{t}_s, \hat{t}_s + 1]) \leftarrow SA.\text{iterator}().\text{next}()$ ;
28   if  $t_s - \hat{t}_s > \Delta$  then
29    |  $Q.\text{insert}((\hat{e}, [\hat{t}_s, t_s]))$ ;
30    |  $SA.\text{replace}((\hat{e}, [\hat{t}_s, \hat{t}_s + 1]), (\hat{e}, [t_s, t_s + 1]))$ ;
31    |  $G.\text{decreaseKey}(SA, t_s)$ ;
32    |  $(\hat{e}, [\hat{t}_s, \hat{t}_s + 1]) \leftarrow G.\text{first}().\text{iterator}().\text{next}()$ ;
33   TRANSFER( $Q, \hat{t}_s, S_{out}$ );
34 while  $\neg G.\text{isEmpty}()$  do
35    $SA \leftarrow G.\text{extractMin}()$ ;
36   Iterator  $results \leftarrow SA.\text{iterator}()$ ;
37   while  $results.\text{hasNext}()$  do
38    | Element  $(\hat{e}, [\hat{t}_s, \hat{t}_s + 1]) \leftarrow results.\text{next}()$ ;
39    |  $Q.\text{insert}((\hat{e}, [\hat{t}_s, \infty)))$ ;
40 while  $\neg Q.\text{isEmpty}()$  do  $Q.\text{extractMin}() \hookrightarrow S_{out}$ ;

```

---

inserted (see line 21). At steady state a SweepArea contains exactly  $N$  elements since one element is removed from the SweepArea prior to the next insertion.

3. The transfer of elements from queue  $Q$  to the output stream depends on timestamp  $\hat{t}_S$ , which is the minimum start timestamp over all SweepAreas (see line 33). This condition is important for the correct order of the output stream (compare with variable  $\min_{t_S}$  in Algorithm 10). The priority queue  $G$  is used to efficiently detect the SweepArea holding the element with the smallest start timestamp. Queue  $G$  holds a reference to each SweepArea in ascending order by the smallest start timestamp of their elements. Hence,  $G.\text{first}()$  returns the SweepArea that contains the element with the smallest start timestamp (see line 26). As each SweepArea provides a linkage by start timestamps, the element with the smallest start timestamp within a SweepArea is the first one delivered by the iterator of the method *iterator* (see line 27). Lines 28 to 32 specify an optimization which is explained below.

If all elements of the input stream have been processed, an iteration over the non-empty SweepAreas generates the final output (see lines 34 – 40). The end timestamp of the remaining elements is set to  $\infty$  (compare with Algorithm 13). Inserting the elements into queue  $Q$  brings them into the desired output stream order. Eventually, the queue is consumed and its elements are appended to the output stream.

**Remark 11.5.** An optimization for lines 34 – 40 would be to merge the iterators over the SweepAreas directly as those are implicitly ordered by start timestamps. This optimization would restrict the size of the queue to the number of non-empty SweepAreas. The same optimization can be applied to lines 34 to 41 in Algorithm 10.

### Priority Queue $G$

Queue  $G$  is used analogously to Algorithm 10. Its sole purpose is to efficiently identify the SweepArea holding the element with the minimum start timestamp. In contrast to grouping with aggregation where queue  $G$  stores references to group identifiers,  $G$  maintains references to SweepAreas here. The reason is that the partitioned window does not enrich output stream elements with their group identifier and thus this information is not required for generating results. Another difference to Algorithm 10 is the use of the method *decreaseKey*, which reorganizes the priority queue due to a change in the priority assigned to a SweepArea. As the priority is determined by the smallest start timestamp in a SweepArea, it needs to be adjusted whenever elements are removed from the SweepArea (see line 20) or elements in the SweepArea are replaced (see line 30).

### Parameter $\Delta$

During initialization it might happen that a SweepArea (group) contains less than  $N$  elements. It is also possible that a SweepArea is already filled with  $N$  elements, but no further elements arrive during a certain time period. In both cases, the smallest start timestamp of the corresponding SweepArea remains unchanged, while other groups continuously receive further elements during this period. For correctness, it is essential to

append only those elements to the output stream which have a start timestamp less than or equal to the minimum start timestamp over all SweepAreas. Therefore, the output generation is blocked as long as the minimum start timestamp belongs to an element in a SweepArea that does not proceed in (application) time because it does not receive further elements. The parameter  $\Delta$  restricts this kind of blocking by specifying an upper bound for the delay between the start timestamp of the incoming element  $t_S$  and the minimum start timestamp over all SweepAreas  $\hat{t}_S$  (see line 28).

Whenever the difference between  $t_S$  and  $\hat{t}_S$  exceeds  $\Delta$ , the element  $(\hat{e}, [\hat{t}_S, \hat{t}_S + 1))$ , which is the one with the minimum start timestamp, is split into two elements with consecutive time intervals at  $t_S$ . The first element  $(\hat{e}, [\hat{t}_S, t_S))$  is inserted into queue  $Q$  (see line 29), the second element  $(\hat{e}, [t_S, t_S + 1))$  replaces the original element in the SweepArea (see line 30). As the start timestamp of this new element is  $t_S$ , the split caused the minimum start timestamp in this SweepArea to increase from  $\hat{t}_S$  to  $t_S$ . Hence, splitting the element with the minimum start timestamp generally raises the minimum start timestamp over all SweepAreas. As a consequence, the timestamp used to transfer elements from queue  $Q$  continually increases. The splitting does not conflict with correctness as no chronon is lost when splitting a time interval into two consecutive intervals.

Parameter  $\Delta$  should not be set too small as splitting increases the output stream rate. However, setting parameter  $\Delta$  ensures a minimum of liveness whenever incoming stream elements are not distributed uniformly among groups. Thus, the choice of parameter  $\Delta$  highly depends on the specified grouping function  $f_{group}$ .

$\omega_{f_{group},1}^{\text{partition}}(S_1)$		
Tuple	$t_S$	$t_E$
$b$	1	10
$a$	3	7
$c$	4	$\infty$
$a$	7	$\infty$
$b$	10	$\infty$

Table 11.13: Partitioned window of size 1 over chronon stream  $S_3$

**Example 11.12.** Table 11.13 demonstrates the results of a partitioned window with size 1 over stream  $S_3$  (see page 98). We reuse the grouping function specified in the example for Algorithm 10, which groups elements according to characters. Hence, the first group (substream) receives all stream elements with tuple  $b$ , the second group those with  $a$ , and the third group those with  $c$ . Due to the count-based window of size 1 applied to every group, the original end timestamp of an element in that group is replaced with the start timestamp of the next element belonging to this group. For instance, at delivery of element  $(b, [10, 11])$  the element  $(b, [1, 2))$  stored in the corresponding SweepArea is removed. A new element  $(b, [1, 10))$  is created and inserted into queue  $Q$  and finally output, while element  $(b, [10, 11])$  is inserted into the SweepArea. If no further elements arrive, the current elements in the SweepArea remain valid. For this reason, the end timestamp of the last three elements in the table is  $\infty$ .

## 11.7 Split and Coalesce

The two following operators, *split* and *coalesce*, take a physical stream as input and produce a snapshot-equivalent physical stream as output. The split operator generates multiple value-equivalent stream elements for a single incoming element by splitting the time interval into consecutive intervals, whereas coalesce merges value-equivalent elements with consecutive time intervals. Hence, both operators are inverse to each other. We use both operators for physical optimization (see Chapter 12). Moreover, split is required for windowed subqueries as it allows the transformation of a physical stream into a chronon stream (see Section 12.3).

### 11.7.1 Split

Algorithm 15 defines the split operator denoted by  $\varsigma_{l_{out}}$ , where parameter  $l_{out}$  specifies the maximum length of time intervals in the output stream. This operator splits every incoming element with a time interval length greater than  $l_{out}$  into multiple value-equivalent elements by dividing the original time interval into consecutive subintervals of length  $l_{out}$ . The last subinterval may be shorter than  $l_{out}$  for the case that the length of the original time interval is no multiple of  $l_{out}$ . The state consists of a SweepArea that keeps elements from the input stream until their time intervals are completely split.

---

**Algorithm 15:** Split ( $\varsigma_{l_{out}}$ )

---

**Input** : physical stream  $S_{in}$ ; output interval length  $l_{out}$   
**Output** : physical stream  $S_{out}$

```

1  $S_{out} \leftarrow \emptyset;$ 
2 Let SA be the empty SweepArea( $\leq_{t_E}, p_{query}, p_{remove}$ );
3 Let Q be an empty min-priority queue with priority  $\leq_{t_S, t_E}$ ;
4  $latest_{t_S} \in T \cup \{\perp\}; latest_{t_S} \leftarrow \perp;$ 
5 foreach  $s := (e, [t_S, t_E]) \leftarrow S_{in}$  do
6   if  $latest_{t_S} \neq \perp$  then
7      $\quad \text{SPLITUP}(SA.\text{query}(s, 1), latest_{t_S}, t_S, l_{out}, Q, S_{out});$ 
8     if  $t_E - t_S > l_{out}$  then
9        $\quad (e, [t_S, t_S + l_{out}]) \leftarrow S_{out};$ 
10       $\quad SA.\text{insert}(s);$ 
11    else  $s \leftarrow S_{out};$ 
12     $latest_{t_S} \leftarrow t_S;$ 
13     $\quad SA.\text{purgeElements}(s, 1);$ 
14  $\text{SPLITUP}(SA.\text{iterator}(), latest_{t_S}, \infty, l_{out}, Q, S_{out});$ 

```

---

### SweepArea Implementation and Parameters

The implementation is based on two lists. The first list keeps the stream elements in ascending order by start timestamps, while the second list consists of references to the elements in the first list ordered by  $\leq_{t_E}$ . The query predicate  $p_{query}$  always returns *true*.

---

**Procedure** SPLITUP(*Iterator qualifies, timestamp latest<sub>t<sub>s</sub></sub>, timestamp t<sub>S</sub>, output interval length l<sub>out</sub>, priority queue Q, stream S<sub>out</sub>*)

---

```

1 while qualifies.hasNext() do
2   Element ( $\hat{e}, [\hat{t}_S, \hat{t}_E]$ )  $\leftarrow$  qualifies.next();
3   T lastSplit  $\leftarrow \hat{t}_S + ((\text{latest}_{t_S} - \hat{t}_S) \text{ div } l_{out}) \cdot l_{out}$ ;
4   T newSplit  $\leftarrow$  lastSplit + lout;
5   while newSplit + lout <  $\hat{t}_E$   $\wedge$  newSplit  $\leq t_S$  do
6     Q.insert (( $\hat{e}, [newSplit, newSplit + l_{out}]$ ));
7     newSplit  $\leftarrow$  newSplit + lout;
8   if newSplit <  $\hat{t}_E$   $\wedge$  newSplit  $\leq t_S$  then
9     Q.insert (( $\hat{e}, [newSplit, \hat{t}_E]$ ));
10  while  $\neg Q.isEmpty()$  do
11    Q.extractMin()  $\hookrightarrow S_{out}$ ;

```

---

Because of this, the method *query* returns an iterator over all elements present in the SweepArea. The remove predicate  $p_{remove}$  is identical to that used for the join. Accordingly, purging elements from the SweepArea follows the linkage of the second list and drops elements as long as their end timestamp is less than or equal to the start timestamp of the incoming element. Queue *Q* aligns elements in lexicographical order denoted by  $\leq_{t_S, t_E}$ , i.e., primarily by start timestamps and secondarily by end timestamps.

### Description of Algorithm

For the first element from the input stream, variable  $\text{latest}_{t_S} = \perp$ . Otherwise,  $\text{latest}_{t_S}$  is set to the start timestamp of the most recent element (see line 12). For all incoming stream elements, except the first one, the auxiliary procedure SPLITUP is called (see line 7). After this, each incoming element is checked to see if the length of its time interval exceeds  $l_{out}$  (see line 8). If this is the case, a value-equivalent element attached with the first subinterval of length  $l_{out}$ , namely  $(e, [t_S, t_S + l_{out}])$ , is appended to the output stream (see line 9). Afterwards the incoming element is inserted into the SweepArea because further elements have to be generated through splitting. Otherwise, the time interval length of the incoming element is less than or equal to  $l_{out}$  and, thus, the element can directly be appended to  $S_{out}$  (see line 11). Finally, the method *purgeElements* removes all elements from the SweepArea that are not needed any longer (see line 13). Line 14 addresses the final stage where the remaining elements in the SweepArea are split and appended to  $S_{out}$  after all elements of  $S_{in}$  have been processed.

### Procedure SPLITUP

The iterator passed as first argument delivers all elements in the SweepArea, either by a call to the method *query* (see line 7) or to the method *iterator* (see line 14). SPLITUP generates new results for the time period  $(\text{latest}_{t_S}, t_S]$ . For each element  $(\hat{e}, [\hat{t}_S, \hat{t}_E])$  provided by iterator *qualifies*, the start timestamp at which the element was split last, termed

$lastSplit$ , is computed first, where the time instant  $lastSplit \leq latest_{t_S}$ . According to the specified output length of time intervals  $l_{out}$ , the next start timestamp, called  $newSplit$ , is set to  $lastSplit + l_{out}$ , where the time instant  $newSplit > latest_{t_S}$ . The while-loop generates new results by dividing the interval  $[newSplit, t_S + l_{out}]$  into consecutive intervals of size  $l_{out}$  (see lines 5 – 7). The loop terminates if either (i) the end timestamp of the new result ( $newSplit + l_{out}$ ) will become equal to or greater than the end timestamp of the original interval or (ii) the start timestamp of the new result exceeds  $t_S$ . In case (i), the rest of the original time interval contributes to the result (see line 9). Hence, the interval length is less than or at least equal to  $l_{out}$ . Condition (ii) prevents the creation of results with a start timestamp greater than  $t_S$  because those would conflict with the output stream order. All results inherit the tuple of the element they are generated from. To guarantee the ordering requirement for the output stream, all results are inserted into the min-priority queue  $Q$ . At the end of procedure SPLITUP queue  $Q$  is emptied by appending all elements to  $S_{out}$  (see lines 10 – 11).

**Property 11.6** (Chronon Output Stream). For the special case  $l_{out} = 1$ , the output stream of the split operator is a *chronon* stream.

$\zeta_1(S_1)$		
Tuple	$t_S$	$t_E$
c	1	2
c	2	3
c	3	4
c	4	5
c	5	6
a	5	6
c	6	7
a	6	7
d	6	7
...		
b	16	17

Table 11.14: Split with output interval length  $l_{out} = 1$  over physical stream  $S_1$

**Example 11.13.** Table 11.14 illustrates an excerpt of the output stream generated by the split operator applied to stream  $S_1$  (see page 73). Parameter  $l_{out}$  is 1. Therefore, the time intervals in the output stream are chronons. Every element in  $S_1$  is split into multiple elements as follows. The attached time interval is split into a sequence of time intervals, where each interval covers a single time instant, while the tuple remains the same.

### 11.7.2 Coalesce

Algorithm 17 shows the implementation of the coalesce operator. The coalesce, denoted by  $\zeta_{l_{max}}$ , merges value-equivalent elements with consecutive time intervals. Parameter

$l_{max}$  expressed as subscript defines an upper bound on the length of time intervals in the output,  $l_{max} < \infty$ . The state consists of a SweepArea that organizes elements in order to merge them with future incoming elements.

---

**Algorithm 17:** Coalesce ( $\zeta_{l_{max}}$ )

---

**Input** : physical stream  $S_{in}$ ; maximum output interval length  $l_{max}$   
**Output** : physical stream  $S_{out}$

```

1  $S_{out} \leftarrow \emptyset;$ 
2 Let  $SA$  be the empty SweepArea( $\leq_{ts}, p_{query}, p_{remove}^{l_{max}}$ );
3 foreach  $s := (e, [t_S, t_E]) \leftarrow S_{in}$  do
4   Iterator  $results \leftarrow SA.extractElements(s, 1);$ 
5   while  $results.hasNext()$  do
6      $results.next() \hookrightarrow S_{out};$ 
7   Iterator  $qualifies \leftarrow SA.query(s, 1);$ 
8   if  $qualifies.hasNext()$  then
9     Element  $(\hat{e}, [\hat{t}_S, \hat{t}_E]) \leftarrow qualifies.next();$ 
10     $SA.replace((\hat{e}, [\hat{t}_S, \hat{t}_E]), (\hat{e}, [\hat{t}_S, t_E]));$ 
11  else  $SA.insert(s);$ 
12 Iterator  $results \leftarrow SA.iterator();$ 
13 while  $results.hasNext()$  do
14    $results.next() \hookrightarrow S_{out};$ 

```

---

### SweepArea Implementation and Parameters

The SweepArea is a hash table where the elements are also linked according to  $\leq_{ts}$ , i.e., their arrival order is preserved. The query predicate is defined by

$$p_{query}(s, \hat{s}) := \begin{cases} \text{true} & \text{if } e = \hat{e} \wedge t_S = \hat{t}_E, \\ \text{false} & \text{otherwise.} \end{cases}$$

It checks whether an element in the SweepArea is value-equivalent to the incoming element and their time intervals are consecutive. The remove predicate,

$$p_{remove}^{l_{max}}(s, \hat{s}) := \begin{cases} \text{true} & \text{if } \hat{t}_E < t_S \vee \hat{t}_E - \hat{t}_S \geq l_{max}, \\ \text{false} & \text{otherwise,} \end{cases}$$

verifies if (i) the end timestamp of an element in the SweepArea is less than the start timestamp of the incoming element or (ii) the time interval length is equal to or greater than  $l_{max}$ . In the first case, the element does not need to be kept in the state any longer because a consecutive time interval will not be found. The second case addresses the liveness of the coalesce operator. It ensures that an element is released from the SweepArea if its interval length exceeds the upper bound  $l_{max}$ , although it might be possible to coalesce this element with future elements. Hence, the second condition restricts the blocking

behavior of the coalesce on the output whenever a large amount of value-equivalent elements with consecutive time intervals occur in the input stream.

### Description of Algorithm

Each incoming element is processed as follows:

1. The iteration carried out by method *extractElements* traverses the SweepArea in ascending start timestamp order. As long as the remove predicate holds, elements are extracted from the SweepArea and appended to the output stream (see lines 4 – 6).
2. The SweepArea is probed for value-equivalent elements which have an end timestamp equal to the start timestamp of the incoming element (see line 7). The first element satisfying this condition is merged with the incoming element. This means, a new element is created by keeping the tuple and coalescing the time intervals. This new element replaces the original element in the SweepArea (see line 10).
3. If the method *query* does not detect a qualifying element, the incoming element is inserted into the SweepArea (see line 11).

After all elements of the input stream have been processed, the remaining elements in the SweepArea are appended to the output stream (see lines 12 –14).

$\zeta_{20}(S_2)$

Tuple	$t_s$	$t_E$
<i>b</i>	1	15
<i>d</i>	3	9
<i>a</i>	4	5
<i>e</i>	10	18

Table 11.15: Coalesce with maximum interval length  $l_{max} = 20$  over physical stream  $S_2$

**Example 11.14.** Table 11.15 shows the output stream of the coalesce operator with  $l_{max} = 20$  over stream  $S_2$  (see page 73). The element  $(b, [7, 15])$  merges with  $(b, [1, 7])$ . Thus, the output stream contains a single element  $(b, [1, 15])$ . In this example,  $l_{max}$  does not have any effect because the interval length is less than 20.



# 12 Physical Query Optimization

While Chapter 10 presented the fundamentals of logical query optimization, this chapter discusses the basics of physical query optimization. First, Section 12.1 defines equivalences for physical streams and physical plans. Then, Section 12.2 addresses the impact of split and coalesce, two operators specific to our physical algebra. Finally, Section 12.3 describes how to generate a physical plan from its logical counterpart.

## 12.1 Equivalences

Physical stream and plan equivalence derive from the corresponding logical equivalences.

**Definition 12.1** (Physical Stream Equivalence). We define two physical streams  $S_1^p, S_2^p \in \mathbb{S}_{\mathcal{T}}^p$  to be *equivalent* iff their corresponding logical streams are equivalent.

$$S_1^p \doteq S_2^p : \Leftrightarrow \varphi^{p \rightarrow l}(S_1^p) \doteq \varphi^{p \rightarrow l}(S_2^p) \quad (12.1)$$

The function  $\varphi^{p \rightarrow l}$  transforms the given physical stream  $S^p$  into its logical counterpart (see Section 6.5.3). Be aware that physical stream equivalence neither implies that elements in both streams occur in the same order nor that the time intervals associated to tuples are built in the same way. We already consider two physical streams as equivalent if their snapshots are equal at every time instant.

**Definition 12.2** (Physical Plan Equivalence). Two physical query plans over the same set of physical input streams are denoted *equivalent* if their results are snapshot-equivalent, i. e., iff for every time instant  $t$  the snapshots of their output streams at  $t$  are equal.

The definition reduces physical plan equivalence to a pure semantic equivalence. If  $S_1^p$  and  $S_2^p$  are the physical output streams of two query plans respectively, physical plan equivalence enforces that  $S_1^p \doteq S_2^p$ . Physical optimizations do not conflict with the correctness of query results as long as plan equivalence is preserved.

## 12.2 Coalesce and Split

Our physical stream algebra contains two novel operators, *coalesce* and *split*, which do not have a counterpart in the logical algebra. The reason is that both operators solely affect the physical stream representation but not the logical one. Coalesce merges value-equivalent elements with consecutive time intervals, whereas split performs the inverse operation.

### 12.2.1 Semantic Impact

**Lemma 12.1.** *Split and coalesce commute with any snapshot-reducible operator.*

*Proof.* It is sufficient to prove that physical stream equivalence is preserved whenever split or coalesce are applied to a physical stream. Let  $S^p$  be a physical stream. It follows directly from the definitions that

$$\zeta_{l_{max}}(S^p) \doteq S^p \iff \varphi^{p \mapsto l}(\zeta_{l_{max}}(S^p)) \doteq \varphi^{p \mapsto l}(S^p). \quad (12.2)$$

From the snapshot perspective, coalesce does not have any effect [SJS01]. Hence, the two logical streams are equal. As split is exactly the inverse operator, the same arguments can be applied to verify

$$\varsigma_{l_{out}}(S^p) \doteq S^p. \quad (12.3)$$

To summarize, both operators preserve snapshot-equivalence. For this reason, it does not matter whether they are applied to the input streams of a snapshot-reducible (standard) operator or to its output stream, or at all. The logical output stream of the operator will remain unchanged.  $\square$

Lemma 12.1 offers new potential for physical query optimization by defining a set of physical transformation rules.

### 12.2.2 Runtime Effects

Although split and coalesce do not enhance the expressive power of our algebra, both operators have impact on stream rates and on the validity of stream elements, i. e., the length of time intervals. In general, coalesce extends the interval length and reduces stream rates, whereas split shortens the interval length and increases stream rates. Validity and stream rates are correlated with operator resource requirements. Stream rates directly affect processing costs. For a simple filter and a non-saturated system, a doubled stream rate doubles the processing costs. The validity mainly influences expiration. Some operators such as aggregation and difference cannot output results until the corresponding elements in the state are expired. For these operators, a shorter validity would lead to earlier results and thus would help to reduce latency. Another argument to be considered is that higher stream rates not only adversely affect operator costs but also scheduling costs.

Altogether, coalesce and split can be employed for physical query optimization. One objective for their use could be to control the tradeoff between latency of query results and operator resource allocation. However, any optimization decisions should be based on a sound cost model which is beyond the scope of Part II. In Part III, we investigate the impact of the average interval length and stream rates on our physical operators in detail. Furthermore, we propose an appropriate cost model to estimate operator resource allocation in terms of memory usage and processing costs. This cost model enables the optimizer to assess the benefit from installing split and coalesce operators in a physical query plan.

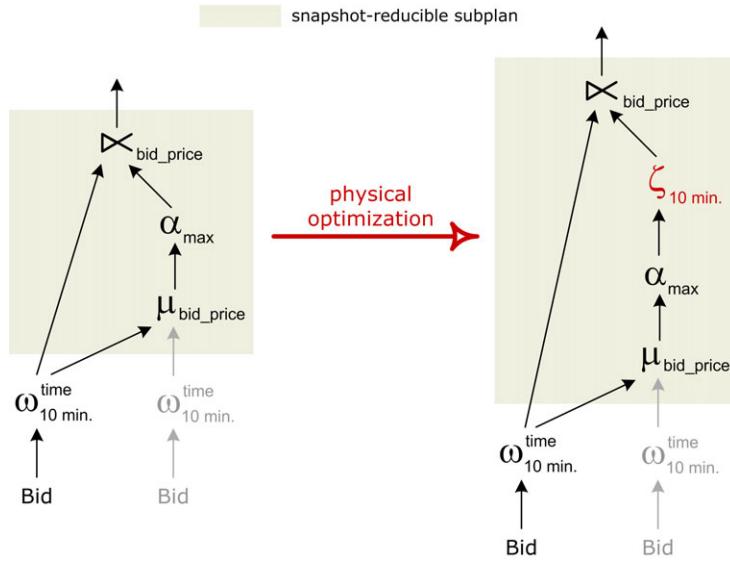


Figure 12.1: Physical plan optimization using coalesce

**Example 12.1.** Figure 12.1 shows a physical optimization for the highest bid query (see page 38 for query language statement). A coalesce operator is placed downstream of the scalar aggregation. The coalesce is likely to reduce the input stream rate of the semi-join because the maximum bid price computed by the aggregate is likely to be constant over a series of incoming bids. Recall that the aggregation receives all bids, not only those for a particular item. We set the maximum tolerated interval length to 10 minutes. This setting guarantees a certain degree of liveness by enforcing the coalesce operator to emit a result on arrival of an incoming element even in those cases where the maximum bid price remains constant for more than 10 minutes. Be aware that the coalesce operator does not violate the snapshot-reducibility property of subplans, a feature visualized by the grey highlighted boxes in Figure 12.1.

## 12.3 Physical Plan Generation

During physical plan generation the operators in the logical plan are replaced with their physical counterparts.

### 12.3.1 Standard Operators

In order to choose a suitable implementation for a standard operator, the common techniques known from conventional database systems can be applied. For example, which type of join implementation is used highly depends on the join predicate. In the case of an equi-join, a hash-based implementation is adequate, whereas a similarity join requires a nested-loops implementation in general.

Chapter 11 presented an abstract implementation for every standard operator. Various

implementations of the same physical operator can be derived from exchanging the SweepAreas. With regard to our join example, Algorithm 6 can be used with hash-based or a list-based SweepAreas, or a mixture [KNV03]. Hence, the concrete implementation of a physical operator depends on the choice and parameterization of its SweepAreas. In addition, a DSMS may provide further specializations of these operators optimized for a particular purpose, e. g., one-to-one or one-to-many joins. Based on the available metadata, the optimizer determines the best operator implementation and installs the corresponding physical operator in the query plan.

### 12.3.2 Window Operators

With regard to window operators, we distinguish between two cases:

1. If the window operator in the logical plan refers to a *source stream* being a chronon stream, it can be replaced directly with its physical counterpart.
2. If the window operator in the logical plan refers to a *derived stream* (subquery) or a source stream that already provides time intervals, then the logical window operator is replaced with a *split operator* having output interval length 1 followed by the corresponding physical window operator.

The first case occurs whenever a raw stream is converted into a physical stream (see Section 6.5.1). The time intervals in such a stream are chronons because they cover only a single time instant. According to our semantics, the logical time-based sliding window operator expands the validity of every tuple at every time instant. A tuple being valid at time instant  $t$  in the input becomes valid during all instants in  $[t, t + w)$  in the output. If the physical input stream is a chronon stream, the physical time-based sliding window preserves the desired semantics. In the second case, the time interval length exceeds 1 and the time-based sliding window would not produce correct results because it would simply adjust the time intervals to a new length. As a consequence, the physical plan would not generate results snapshot-equivalent to those of the corresponding logical plan. The same argumentation applies to count-based and partitioned windows. To ensure consistent results with our logical algebra, we solely permit chronon streams as inputs for window operators. Any physical stream which is not a chronon stream has to be transformed into a chronon stream first. This can be achieved by placing a split operator with output interval length 1 upstream of the window operator.

The split operator is therefore not only an instrument for physical optimization but actually required in our physical algebra to deal with windows over (i) subqueries and also (ii) source streams whose tuples are already tagged with a time interval instead of a timestamp.

**Example 12.2.** With regard to our running example, all source streams in a physical plan are chronon streams. For the continuous queries investigated so far, the logical plans shown in Figures 9.1 and 9.2 can be translated directly into their physical counterparts by replacing the individual operators. The following statement is an example for a windowed continuous query over a derived stream, namely, the `ClosingPriceStream` defined in Example 7.3 on page 37.

**Average Selling Price By Seller Query** – For each seller, maintain the average selling price over the 10 last items sold.

```
SELECT sellerID, AVG(price)
FROM ClosingPriceStream WINDOW(PARTITION BY sellerID ROWS 10)
GROUP BY sellerID;
```

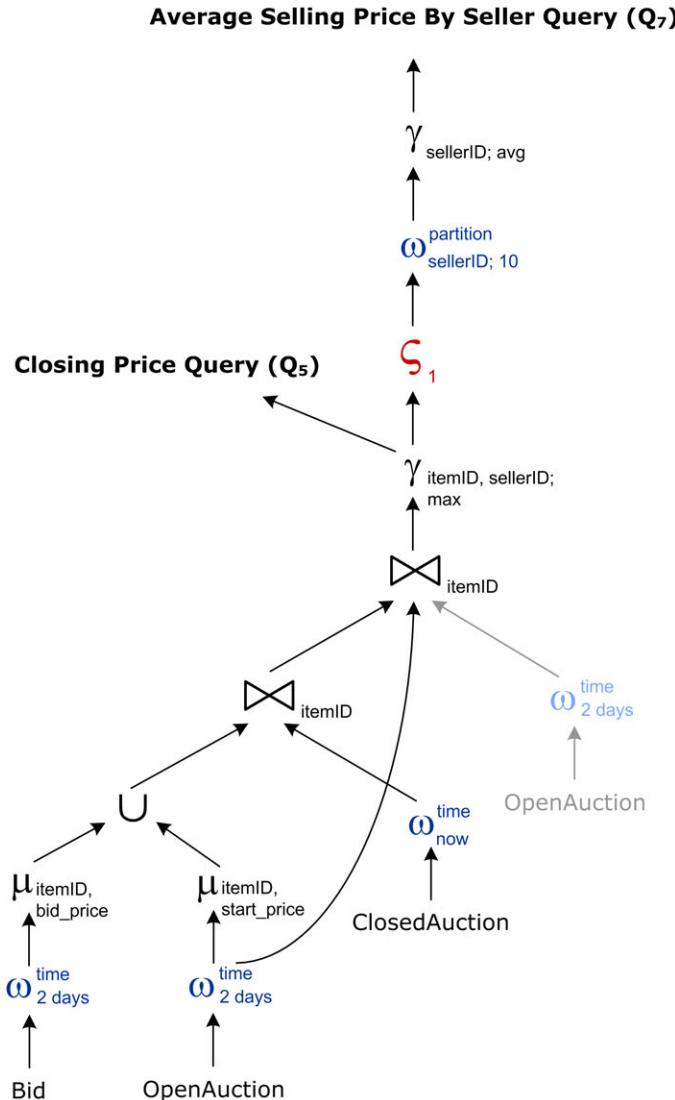


Figure 12.2: Physical plan with windowed subquery

Figure 12.2 demonstrates the translation from the logical into the physical plan. Here, the optimizer would replace the logical partitioned window with a split operator followed by the corresponding physical partitioned window. The alert reader will have noticed that the split operator is actually not required in this physical plan because the time

Operation	Expiration Pattern	Condition for Chronon Stream
$\sigma_p(S^p)$	retains end timestamp order	$S^p$ is a chronon stream
$\mu_f(S^p)$	retains end timestamp order	$S^p$ is a chronon stream
$\zeta_{l_{out}}(S^p)$	retains end timestamp order	$l_{out} = 1 \vee S^p$ is a chronon stream
$\cup(S_1^p, S_2^p)$	destroys end timestamp order	$S_1^p$ and $S_2^p$ are chronon streams
$\bowtie_\theta(S_1^p, S_2^p)$	destroys end timestamp order	$S_1^p$ or $S_2^p$ is a chronon stream
$\delta(S^p)$	destroys end timestamp order	$S^p$ is a chronon stream
$-(S_1^p, S_2^p)$	destroys end timestamp order	$S_1^p$ is a chronon stream
$\gamma_{f_{group}, f_{agg}}(S^p)$	destroys end timestamp order	$S^p$ is a chronon stream
$\omega_{f_{group}, N}^{\text{partition}}(S^p)$	destroys end timestamp order	–
$\zeta_{l_{max}}(S^p)$	destroys end timestamp order	$S^p$ is a chronon stream and $l_{max} = 1$
$\omega_w^{\text{time}}(S^p)$	enforces end timestamp order	$w = 1$ (now-window)
$\omega_N^{\text{count}}(S^p)$	enforces end timestamp order	–
$\alpha_{f_{agg}}(S^p)$	enforces end timestamp order	$S^p$ is a chronon stream

Table 12.1: Impact of physical operators on order of end timestamps in their output stream and conditions under which the output stream of a physical operator is a chronon stream

intervals in the `ClosingPriceStream` already have length 1 due to the now-window over the `ClosedAuctionStream`.

### 12.3.3 Split Omission

In order to detect optimization opportunities like the one mentioned in the previous example, we suggest annotating subplans that satisfy certain properties. The query optimizer incorporates the annotated properties into the translation process to select the final query evaluation plan.

The split operator can be omitted upstream of the window operator if the next operator upstream of the window already generates a chronon stream. The optimizer can verify this property by annotating a query plan bottom-up. Table 12.1 demonstrates under which conditions the chronon stream property propagates downstream.

**Remark 12.1.** During logical and physical plan generation, now-windows are removed from the final plan. From the physical point of view, this optimization is correct because (i) if  $S^p$  is a chronon stream, then  $\omega_1^{\text{time}}(S^p) \doteq S^p$ , otherwise (ii)  $\omega_1^{\text{time}}(\zeta_1(S^p)) \doteq S^p$  (see Lemma 12.1). In the latter case,  $S^p$  can either be a derived stream (subquery) or a source stream obtained from a raw stream that already provides time intervals. Placing the now-window would hence compute the identity and thus waste system resources.

### 12.3.4 Expiration Patterns

As explained in Section 11.3.3, the knowledge of expiration patterns permits the determination of suitable operator implementations to improve the efficiency of query

execution plans [GÖ05]. Annotating query plans with stream expiration properties helps the optimizer to identify appropriate SweepArea implementations for an operator during physical plan generation.

Recall that besides the required start timestamp order, a physical stream can also possess a non-decreasing order on end timestamps. This property implicitly holds for all chronon streams, including the class of physical source streams obtained from raw streams. Therefore, plan annotation can be performed bottom-up.

Table 12.1 reflects the behavior of physical operators with respect to the end timestamp order of their output stream. It lists operators that (i) retain the end timestamp order of their input stream in their output stream, (ii) destroy the end timestamp order, or (iii) enforce the end timestamp order of their output stream. Depending on the operator behavior, the optimizer can determine whether the input streams of a physical operator satisfy the optional end timestamp ordering property or not. This knowledge enables the optimizer to decide whether the expiration pattern optimization is applicable to the SweepAreas of that operator or not.

**Example 12.3.** Having a look at the example queries  $Q_3$  and  $Q_4$  shown in Figure 9.1 (see page 56) reveals that the expiration pattern optimization is applicable to the join in  $Q_3$  and the semi-join in  $Q_4$  because the source streams are chronon streams and all operators upstream either enforce or retain the end timestamp order. In contrast, the joins in  $Q_5$  and  $Q_6$  (see Figure 9.2 on page 57) cannot be optimized because the upstream union and grouping, respectively, destroy the end timestamp ordering property.



# 13 Query Execution

This chapter briefly discusses further implementation related issues to understand how continuous queries are implemented and executed in our comprehensive Java software infrastructure for stream processing called PIPES [KS04, CHK<sup>+</sup>06]. Section 13.1 outlines our basic design goals. Section 13.2 explains how continuous queries are managed, while Section 13.3 discusses their processing methodology. The functionality of our runtime components is described in Section 13.4. Section 13.5 summarizes the methods applied in PIPES to deal with stream imperfections and liveness issues. Section 13.6 reports on auxiliary functionality. Finally, Section 13.7 presents some background information about the development status of PIPES.

## 13.1 Design Goals

Contrary to existing implementations, PIPES is not a monolithic DSMS since, inspired by the work of [CW00], we believe that it is almost impossible to develop a general, efficient as well as flexible and extensible system that can cope with the manifold requirements of the various data stream applications. Instead, we developed a highly generic and modular infrastructure for stream processing that provides all the fundamental building blocks to construct a fully functional DSMS prototype tailored to the specific needs of an application. Figure 13.1 gives an architectural overview of PIPES.

PIPES is an integral part of the XXL library for advanced query processing [BBD<sup>+</sup>01, CHK<sup>+</sup>03] and extends XXL towards continuous, data-driven query processing over active data sources. Throughout our development, we have striven to ensure the high-quality programming style of XXL, using design patterns and well-established programming techniques whenever possible [GHJV95]. Despite the overhead induced by this library approach, it has several advantages compared to rapid prototyping, including the following: (i) Due to its generic and deliberate design, PIPES is more flexible and can be extended easily with new functionality. (ii) The high-quality source code, along with the documentation, improves reusability. (iii) PIPES serves as an ideal testbed for algorithmic comparisons because the modular structure and component frameworks facilitate the exchange of functionality.

## 13.2 Query Graph

PIPES manages all continuous queries in a directed, acyclic graph. *Sources* can be raw data streams, relations, or other data feeds. In this thesis we primarily consider data streams as sources. The results of continuous queries are transferred to *sinks* which are, for instance, applications, GUIs, database systems, or other DSMSs. The intermediate nodes

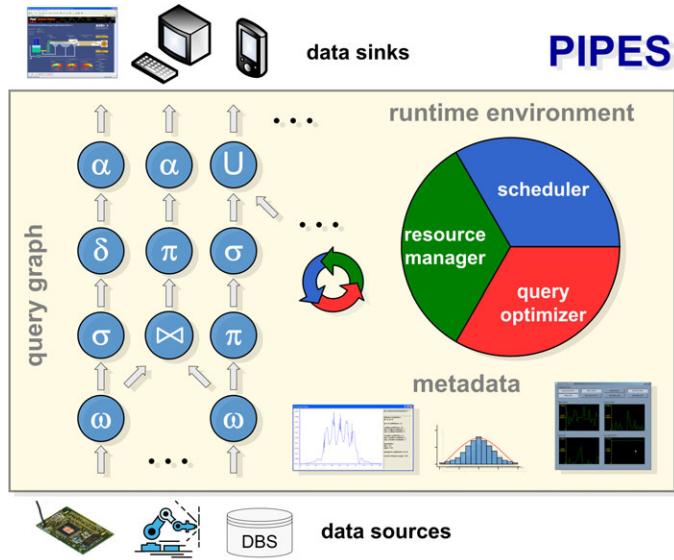


Figure 13.1: An architectural overview of PIPES

in the graph are operators of our physical algebra. Due to the long-running nature of queries, it is profitable not to run each query stand-alone but to exploit subquery sharing. A publish-subscribe mechanism integrated into the graph nodes enables the sharing of common subqueries without the need to place an inter-operator queue between each pair of adjacent nodes. We however recommend the placement of a queue downstream of each active data source to decouple the activity of the source from internal query processing.

### 13.3 Query Processing Fundamentals

Queries are basically executed in a data-driven manner, which is contrary to the demand-driven processing usually employed in database systems [Gra93]. Data-driven processing means, whenever new elements arrive at the sources, that those are pushed through the operator network. When an operator processes an incoming element, its results are forwarded directly to all downstream operators. Our algorithms presented in Chapter 11 implement the data-driven processing methodology, which produces query results in a natural and continuous manner with low latency.

### 13.4 Runtime Environment

Several runtime components control the execution of continuous queries, among them, scheduler, resource manager, and query optimizer. PIPES provides a separate framework for each component.

### 13.4.1 Metadata

Due to the long-running queries and fluctuating stream characteristics, DSMSs have to be highly adaptive. An essential prerequisite for adaptive runtime components is the presence of suitable metadata capturing statistics on the runtime state. For this reason, graph nodes provide a publish-subscribe interface that allows gathering the required metadata at runtime. Besides *static* metadata like schema information, *dynamic* metadata such as average stream rates, selectivities, memory usage, or processing costs are supplied on demand. As most of the metadata in such a system varies over time, PIPES supports functionality for the dynamic provision and continuous maintenance of the various types of available metadata (see [CKS07] for details).

### 13.4.2 Scheduling

The multi-threaded scheduling framework integrated in PIPES implements a 3-layer architecture [CHK<sup>+</sup>07]. The first layer partitions the query graph into subgraphs. The subgraphs are separated from each other through inter-operator queues. The second layer assigns subgraphs to threads, and determines a scheduling strategy for each thread. The scheduling strategy defines the order in which a thread accesses the inter-operator queues in its subgraphs to extract an element and to push it into the next downstream operator. The third layer controls thread scheduling. It allows the specification of how many threads are executed in parallel and how threads behave with regard to their states. This unique framework comes with the advantage of control over the tradeoff between the gain of concurrency and the overhead of context switches. Due to the hybrid architecture, scheduling is neither restricted to a single thread for the entire query graph [MF02, BBDM03, CCZ<sup>+</sup>03] nor to a separate thread per operator [AH00, CCC<sup>+</sup>02, MWA<sup>+</sup>03, HMA<sup>+</sup>04]. Because PIPES supports multi-threaded query execution, it can take advantage of modern multi-core computer architectures, which will gain importance in the future. However, even for a single-processor machine, multi-threaded query execution can be exploited to improve response times whenever multiple queries are evaluated concurrently [CHK<sup>+</sup>07].

### 13.4.3 Resource Management

Stateful operators such as the join need to allocate memory for storing their state. Furthermore, the size of their state mostly correlates with the processing costs, e.g., consider a nested-loops join. The resource management framework provides functionality for distributing system resources among the various operators. To adapt to changing stream characteristics and query workload, PIPES supports techniques to keep resource usage within pre-defined bounds, in particular in times of high system load. Available techniques are: (i) load shedding [CCC<sup>+</sup>02, TCZ<sup>+</sup>03, Vit85], (ii) spilling the state partially to external memory via special SweepAreas [UF01, VNB03, MLA04, CHKS05], and (iii) adapting the window size and time granularity [CKSV06]. The latter approach has the advantage that the query result is an exact subresult of the original result rather than an approximation and thus query optimization is still feasible, whereas sampling-based

techniques like load shedding conflict with optimizations in general [CMN99].

#### 13.4.4 Query Optimization

PIPS additionally contains a framework for rule-based query optimization whose formal foundation, namely the query semantics and the resulting transformation rules, is established in this part of the thesis. Our optimization framework provides the components of the planning subsystem including the parser, semantic interpreter, and logical and physical plan generator. To overcome the deficiency of optimizing only a stand-alone continuous query, we extended approaches from multi-query optimization [RSSB00] towards stream processing. The optimizer takes a new query as input and produces, in a heuristic manner, a set of snapshot-equivalent query plans as output. Each query plan is probed against the currently running query graph and, according to a cost function, the best matching plan is determined. Then, the new plan or the new parts of it are integrated into the operative query graph via the publish-subscribe architecture. Currently, our optimizer uses a preliminary plan enumeration algorithm similar to the one proposed in [SJS01]. In Part III, we present a suitable cost model for estimating the memory consumption and processing costs of query plans. To cope with the scalability and adaptivity requirements of DSMSs, we developed efficient techniques solving the dynamic plan migration problem for arbitrary query plans composed with our algebra (see Part IV). As a result, we are able to optimize already operative continuous queries at runtime without the need to stall their execution during migration [KYC<sup>+</sup>06, YKPS07].

### 13.5 Dealing with Liveness and Disorder

#### 13.5.1 Time Management

Our semantics for continuous queries assumes stream elements to be equipped with a timestamp. An important criterion for the correctness of our physical operators is the proper ordering of their input streams as well as their output stream. This temporal ordering requirement of an input stream implies that an operator should never receive a tuple with start timestamp less than that of any tuple previously received from this input stream. If the DSMS assigns timestamps to tuples when those arrive at the system, the input streams are implicitly ordered by system time. However, our approach mainly considers application time instead of system time because (i) the majority of streaming applications provide an explicit timestamp attribute [SQR03] and (ii) the semantics of continuous queries is more meaningful. For instance, consider a stream with temperature sensor readings. An application timestamp would denote the time at which the sensor reading was taken, whereas a system timestamp would indicate the time at which the corresponding stream element reached the system. In general application and system time are independent from each other [SW04].

### 13.5.2 Problem Definition

Application-defined time poses new challenges to query execution since stream elements arriving at the system from one or more, possibly distributed sources, may be out of order and uncoordinated with each other due to the following reasons: (i) unsynchronized application time clocks at the sources, (ii) different network latencies, and (iii) data transmission over a non-order-preserving channel [SW04]. *Heartbeats* are a mechanism to cope with these problems. A heartbeat consists of a simple timestamp  $t \in T$ . Whenever a source generates a heartbeat, it indicates that the system will receive no future stream elements from this source with a timestamp less than or equal to that specified by the heartbeat. [SW04] proposes techniques to deduce heartbeats from sources, which do not provide heartbeats themselves, based on bounds on application time skew and network latency.

### 13.5.3 Heartbeats in Query Plans

The heartbeat concept is equally applicable to our approach. PIPES fully implements operator-level heartbeats. With regard to liveness, heartbeats can be used to explicitly trigger additional expiration and processing steps to reduce latency. Source heartbeats are propagated downstream through the query graph. Let  $S_1, \dots, S_n$  be the input streams of a physical operator. Suppose  $t_1, \dots, t_n$  are the start timestamps of the latest element delivered from every input stream. When a heartbeat  $t'_i, i \in \{1, \dots, n\}$  reaches the operator, the heartbeat indicates a time progression in stream  $S_i$  from  $t_i$  to  $t'_i$ . Due to the temporal ordering property of the input streams, the operator can advance its state and output results up to timestamp  $\min\{t_1, \dots, t'_i, \dots, t_n\}$ . If the processing of the heartbeat causes the operator to produce new results, then the time progression is propagated downstream and there is no need to send a heartbeat as well. Otherwise, a heartbeat with the value of the minimum timestamp in the operator state is created and passed downstream. Here, the term *operator state* refers to any internal data structures, including the priority queues. The interested reader is referred to [SW04] for a more detailed explanation on heartbeat generation and propagation.

### 13.5.4 Stream Ordering

To handle out-of-order arrival, PIPES can install an input manager which buffers incoming elements and forwards them in increasing timestamp order to the corresponding input queues of the scheduler. The input manager can use heartbeats to release elements from its buffer [SW04]. Another approach to deal with disorder is presented in [ACC<sup>+</sup>03]. Order-sensitive operators require an order specification which permits bounded disorder over their input streams. Besides the order attribute, the order specification includes a *slack* parameter. The slack parameter, which is a non-negative integer, specifies the extent of disorder tolerated. Usually, the operators buffer a number of elements up to this slack value to restore order. To guarantee finite buffer space and finite time, elements are discarded from the input stream if the number of elements violating the assumed ordering exceeds the slack constraint. Hence, a greater slack specification allows to

handle a larger number of out-of-order elements at the expense of an increased latency of query results. PIPES only has to cope with the disorder problem at the sources since all physical operators satisfy the ordering requirement for the output stream. Although it would be feasible to implement a sort operator with a slack parameter in PIPES, which would be similar to a unary union with a limited priority queue size, such an operator would drop any elements being out of order by more than the *slack* size elements. This would consequently lead to approximate query results and make query optimization inapplicable, because the output stream would not even be a sample of the input stream in general. While the slack-approach seems to be practical, it conflicts with main objective of this work, namely to establish a sound query semantics. Therefore, we solve the disorder problem by buffering and heartbeats.

## 13.6 Auxiliary Functionality

Since PIPES seamlessly extends XXL, it has full access to the advanced query processing functionality of XXL. As a consequence, developers can profit from demand-driven relational and spatial query processing frameworks, e.g., [DSTW02, DS00, APR<sup>+</sup>98], XML support, sophisticated aggregate functions [BHS05, HS05, HS06a], and distributed query processing functionality. PIPES inherently offers the possibility of running queries over streams and relations. In addition, it provides the data flow translation operators proposed in [Gra93]. Moreover, index structures on external memory including [SG97, BKSS90, BGO<sup>+</sup>96, GPÖ06] can be employed for state maintenance in those cases where the available main memory is likely to become insufficient. For approximate query answers, XXL provides techniques such as reservoir sampling [Vit85] to construct synopses, i.e., representative summaries of elements.

## 13.7 Development Status of PIPES

The objective of this section is to convey a feeling for the complexity of PIPES. PIPES is a joint project with several members of the database research group at the University of Marburg. Major parts of PIPES, including the core, have been developed for this thesis. In the current version, PIPES consists of more than 300 Java source files, clearly arranged in more than 40 packages, and more than 1000 class files. The large latter number results from the extensive use of anonymous classes in our use cases. Overall, PIPES is about 72,000 lines of Java code. Without blank lines and comments, PIPES is about 40,000 lines of code. This number includes more than 12,000 lines of code for use cases and connectivity functionality to external applications. Hence, the core consists of about 28,000 pure lines of code.

The latest version of PIPES requires Java 5 and supports generics allowing for type-safe implementations and early error detection at compile time. Despite its multi-threaded architecture, PIPES runs fairly stable. In our scalability experiments, we have run thousands of continuous queries, even complex ones, concurrently for multiple hours.

To give users, administrators, and developers the opportunity to inspect the state of the system while it is running, we have developed a graphical user interface, the *PIPES*

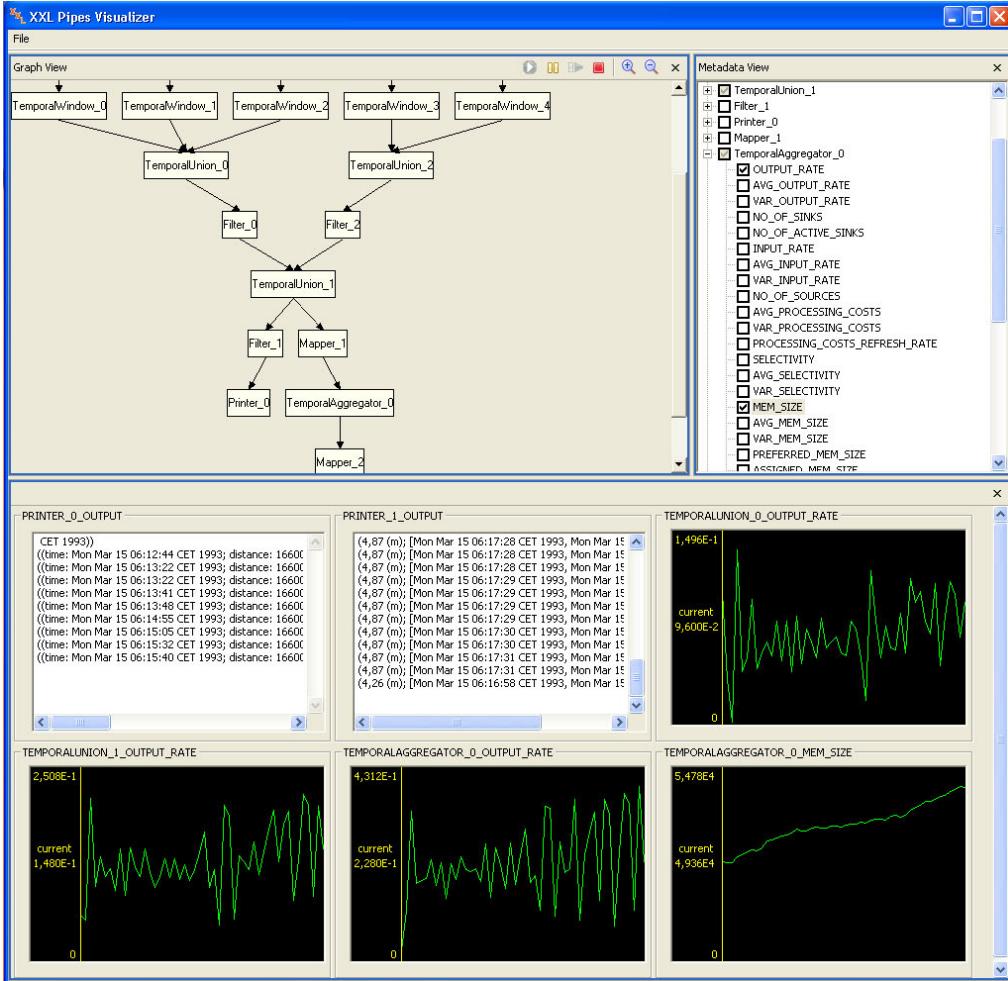


Figure 13.2: PIPES Visualizer

*Visualizer.* A screenshot is shown in Figure 13.2. The visualizer allows a user to (i) start and stop queries, (ii) view the structure of the query graph, (iii) view query results, and (iv) monitor the dynamic runtime metadata such as stream rates, operator memory usage, and operator selectivities. In addition to the visualizer, PIPES provides a frontend to pose queries in our declarative query language and, alternatively, a GUI for the drag-and-drop composition of physical query plans.

The first major milestone in the development of PIPES was its demonstration at the ACM International Conference on Management of Data (SIGMOD) in June 2004. This demonstration focused on the architecture of PIPES and our continuous query semantics. We illustrated the broad applicability of PIPES in the context of traffic management and online auctions. At this time we released PIPES as public, open-source software under the GNU LGPL license. Two years later, in April 2006, at the IEEE International Conference on Data Engineering (ICDE), we demonstrated how industrial software can reap the benefits of PIPES. In cooperation with Langner Communications AG, we successfully

coupled PIPES with the commercial Production-to-Business software *i-Plant* geared to monitor highly automated manufacturing processes. PIPES enables developers and users to benefit from our declarative query language when specifying complex application logic while avoiding custom coding. Moreover, applications profit from higher performance through query optimizations and subquery sharing.

Because we do not want to digress from the research aspects in this thesis, we refer the interested reader for further details to the PIPES user's manual and developer's guide linked at our project website [XXL07].

# 14 Implementation Comparison

This chapter compares our implementation, the *time-interval approach* (TIA), with a semantically equivalent implementation, the *positive-negative approach* (PNA), used for instance in STREAM [ABW06] and Nile [GHM<sup>+</sup>07]. Section 14.1 succinctly introduces PNA. Implementation-specific features of both approaches are discussed in Section 14.2. Section 14.3 shows the results of our thorough experimental studies. The most important insights of our comparison are briefly summarized in Section 14.4.

## 14.1 The Positive-Negative Approach

Another common implementation technique for continuous queries is the positive-negative approach. The implementation is based on streams with elements of the following format: a tuple  $e$ , a timestamp  $t \in T$ , and a sign,  $+$  or  $-$ . A stream is ordered by timestamps. The signs are used to define the validity of elements. The standard operators are modified to handle positive and negative elements, in particular with regard to temporal expiration. The window operators generate the positive and negative elements according to their semantics. For a time-based sliding window and an incoming stream element with application timestamp  $t$ , the window operator sends a positive element with that timestamp. After  $w$  (window size) time units the window operator sends a negative element with timestamp  $t + w$  to signal expiration. For further details, we refer the reader to [GHM<sup>+</sup>07, ABW06]. Unfortunately, these papers outline the ideas but do not delve into algorithmic details. We observed through our experimental studies that an efficient operator implementation, e. g., for grouping with aggregation or difference, is non-trivial and not self-explanatory.

A pair of elements consisting of a positive and its corresponding negative element can be used to express a stream element in the time-interval approach. Namely,  $(e, [t_S, t_E])$  can be implemented by sending a positive element  $(e, t_S, +)$  and a negative element  $(e, t_E, -)$ . Hence, positive elements refer to start timestamps, whereas negative elements refer to end timestamps. Even at this physical level, the semantic equivalence of both approaches becomes obvious. However, the interval approach does not have the drawback of doubling stream rates due to sending positive and negative elements.

## 14.2 Features

We are able to translate any logical query plan either into a physical TIA plan or a physical PNA plan because we implemented both types of operators for conducting our experiments. In the following, we discuss the effects of both approaches on query execution.

### 14.2.1 Window Operators

The fundamental difference between PNA and TIA is the way windowing constructs are implemented. While TIA uses time intervals to model validity, PNA sends positive and negative elements, which are generated by the window operators. Depending on the window type, either approach has its advantages.

#### Time-based Window

For TIA, the time-based window is a stateless operator that, for an incoming element, simply sets the validity according to the window size by modifying the attached time interval. In contrast, the time-based window for PNA is stateful because it has to store the negative elements until those can be emitted to the output stream without violating the order requirement. While TIA only needs to allocate memory for a single stream element, the memory consumption of PNA depends on the input stream rate and the window size. This constitutes a significant drawback. Let us consider an example. For simplicity, assume a constant stream rate of 100 elements per second and a window size of 30 minutes. Then, PNA would keep  $100 \cdot 60 \cdot 30 = 180000$  elements in the data structure of the time-based window. In addition to the memory overhead of PNA, the processing costs are non-negligible as well, covering the creation of negative elements plus their insertion and removal from the state.

#### Count-based and Partitioned Windows

PNA has an apparent advantage for these types of windows as it treats the validity start and end separately. Let us consider the count-based window. According to our semantics, the validity of an element begins at its associated (start) timestamp and ends at the (start) timestamp of the  $N$ -th future stream element, if  $N$  is the window size. PNA can emit the positive element instantly on arrival, whereas TIA has to wait until  $N$  further elements arrive to build the correct time interval. As a result, TIA sends the element at the same time as PNA sends the negative element. Hence, PNA is superior to TIA with regard to latency if a query contains a count-based window. The processing costs of PNA are increased compared to TIA because the negative elements have to be generated. The memory usage is the same. Since a partitioned window applies a count-based window to each substream, the latency benefit of PNA carries over to partitioned windows.

**Remark 14.1.** One might argue that PNA neither needs to produce nor store negative elements in the case of unbounded windows. Recall that this optimization is limited to query plans that contain only stateless operators downstream of the windows. Otherwise, resource usage will exceed any bounds due to the infinite input streams in general.

### 14.2.2 Stateless Operators

As PNA doubles stream rates, twice as many elements have to be processed by stateless operators. Filter predicates and mapping functions have to be evaluated for positive and negative elements. Furthermore, the buffers of the union, which merges streams in

non-decreasing timestamp order, will be greater than for TIA, in particular, for increasing application-time skew between the input streams.

### 14.2.3 Join and Duplicate Elimination

#### Latency

Although these operators are stateful, TIA produces the results for an incoming element immediately. Hence, TIA emits a result at the point in time when PNA sends the positive element for that result.

#### Memory Usage

Both approaches have to keep the same number of elements in the state. For time-based windows, PNA usually consumes more memory than TIA due to the overhead of the stateful window operator, which controls element expiration. A system using PNA can diminish this deficiency by exploiting state sharing. While it is mostly possible for the first operator downstream of a window to share the state with the window, the general case prevents this optimization.

#### Processing Costs

TIA profits from lower processing costs because it does not have to handle negative elements. Be aware that PNA needs to probe the state for negative elements as well in order to compute the correct negative results. For duplicate elimination and hash-joins, this may be an acceptable overhead because probing is cheap, but for a nested-loops join, which is for instance applied to evaluate inequality predicates, doubled probing costs are definitely significant. The processing costs also include invalidation costs. TIA identifies expired elements based on overlap conditions and purges the state of expired elements by following an internal temporal linkage. On the contrary, PNA detects the expiration of a positive element on arrival of the corresponding negative element. The expired element can be removed from the state efficiently using hashing techniques. Whether TIA or PNA removal is more efficient depends on the query expiration pattern [GÖ05]. If the stateful operator directly follows a time-based window, then elements will be removed from the state in the order of their arrival. If the stateful operator is downstream of other stateful operators, then the order in which elements expire is not predictable in general. The SweepAreas of TIA can be optimized for the FIFO expiration pattern using a list instead of the priority queue to maintain the temporal linkage (see Section 11.3.3). The general expiration pattern, however, necessitates the use of a priority queue causing additional logarithmic costs in the worst case. Contrary to that, PNA always invalidates positive elements in the state by hashing on arrival of negative elements.

#### 14.2.4 Grouping, Aggregation, and Difference

##### Latency

In contrast to join and duplicate elimination, TIA cannot produce results for an incoming element immediately. At least one further element is required to generate output. The reason is that grouping with aggregation, scalar aggregation, and difference extract expired elements from the SweepArea and append them to the output stream. PNA can generate the results for an incoming element immediately for grouping with aggregation and scalar aggregation, but only if the input stream does not contain duplicate timestamps. Otherwise, incomplete aggregates may propagate through the query plan and may eventually trigger false alarms or other fatal actions.

**Example 14.1.** Given a chain of three operators: a time-based window, a scalar aggregation computing the sum, and a filter. The tuples of the input stream are non-negative integers. The filter verifies whether the integer value is greater than 6. An application triggers an alarm whenever the filter reports a result. Let us assume the state of the aggregation to be empty. Let  $(7, t, +)$  be the first element that arrives at the aggregation. If this element is followed by  $(2, t, -)$  and no further elements with timestamp  $t$  will arrive, then the correct aggregate value at  $t$  would be 5. However, a false alarm would have been triggered if the aggregation did not wait for the next element and appended  $(7, t, +)$  instantly to the output stream.

The example shows clearly why the PNA implementation also has to wait for a progression in application time to generate the correct aggregates if multiple elements in a stream can have the same timestamp. Although it might be realistic for raw streams not to contain duplicate timestamps, this stream property is destroyed by stateful operators in general. Hence, this advantage of PNA applies only to small class of queries. The output delay is inevitable for the PNA difference operator.

##### Memory Usage and Processing Costs

With regard to memory usage and processing costs, the arguments given for the join and duplicate elimination apply equally to the difference. With regard to aggregation, things get more complex. TIA uses three functions to compute aggregates incrementally (see Section 11.5.7). PNA is based on a differential approach. Incoming positive and negative elements trigger the re-computation of the aggregate value. Prior to sending a new aggregate, the aggregation operator emits a negative aggregate built of the previous aggregate value and the timestamp of the new aggregate to signal expiration of the previous aggregate value. To be efficient, the differential approach requires *inverse* aggregate functions. Otherwise, aggregates have to be computed from scratch by inspecting the entire state on arrival of a negative element. Simple aggregate functions like SUM, COUNT, or AVG, are computable with a constant amount of state information. On the contrary, the state required for aggregate functions like MAX, TOP-k, and COUNT DISTINCT needs to keep all positive elements unless their negative counterpart arrives. The main difference between the aggregation for PNA and TIA is that TIA allows the state to be shared between multiple aggregate functions evaluated over a stream, whereas

PNA needs to maintain a separate state for each aggregate function. On the one hand, this characteristic can multiply the memory consumption for PNA if multiple complex aggregates are evaluated. On the other hand, multiple simple aggregate functions allocate only a constant amount of memory, which is an advantage over TIA.

In terms of processing costs, a similar behavior is observable. Given the aggregate function and its inverse, PNA can compute the new aggregate value with a single call from the previous aggregate and the incoming positive or negative element respectively. Complex aggregate functions incur higher costs due to state maintenance. A general implementation of the PNA aggregation which enforces a shared state for multiple aggregate functions has to keep all positive elements in the state until their negative counterpart arrives. This implementation trades reduced memory consumption for increased processing costs because new aggregates have to be computed by investigating the entire state on arrival of a negative element. However, new aggregates can be computed incrementally for incoming positive elements if the previously released aggregate is buffered. Concerning processing costs, PNA is superior to TIA for simple aggregates and a non-shared state because the aggregation has constant processing costs for an incoming element, while TIA has to perform a temporal range query to update all partial aggregates in the state that overlap with an incoming element. For complex aggregate functions, TIA outperforms PNA in general because it saves the costs for processing negative elements. This becomes particularly clear when PNA uses shared state. In this case, the size of the state is the same for TIA and PNA, but TIA updates aggregate values only for those elements in the state that overlap with the incoming element, while PNA fully traverses the state for each negative element to compute the new aggregate. As a result, TIA usually performs less invocations of the aggregate functions.

#### 14.2.5 General System Overhead

The fact that PNA doubles stream rates does not only affect the operators but also the rest of the DSMS. Because the system has to deal with twice as many elements when using PNA, it will reach its saturation level earlier. If techniques like load shedding [TCZ<sup>+</sup>03] are applied to keep resource usage in bounds, query results may be less accurate when using PNA. Furthermore, the overall memory consumption is increased since the number of elements in inter-operator queues is doubled. PNA also suffers from increased scheduling overhead compared to TIA.

### 14.3 Experimental Evaluation

To validate the aforementioned features, we conducted a series of experiments. Although the described implementational characteristics are independent from the underlying data, which suggests using only synthetic data sets, we implemented the continuous queries from the NEXMark benchmark to emphasize the practical relevance of our results. We used the Firehose Stream Generator [TTPM02] to generate the input streams and relations. The underlying data sets consisted of 1,000,987 bids for 99,990 auctions and 9,958 registered people. An item submitted for auction received on average 10 bids. Note

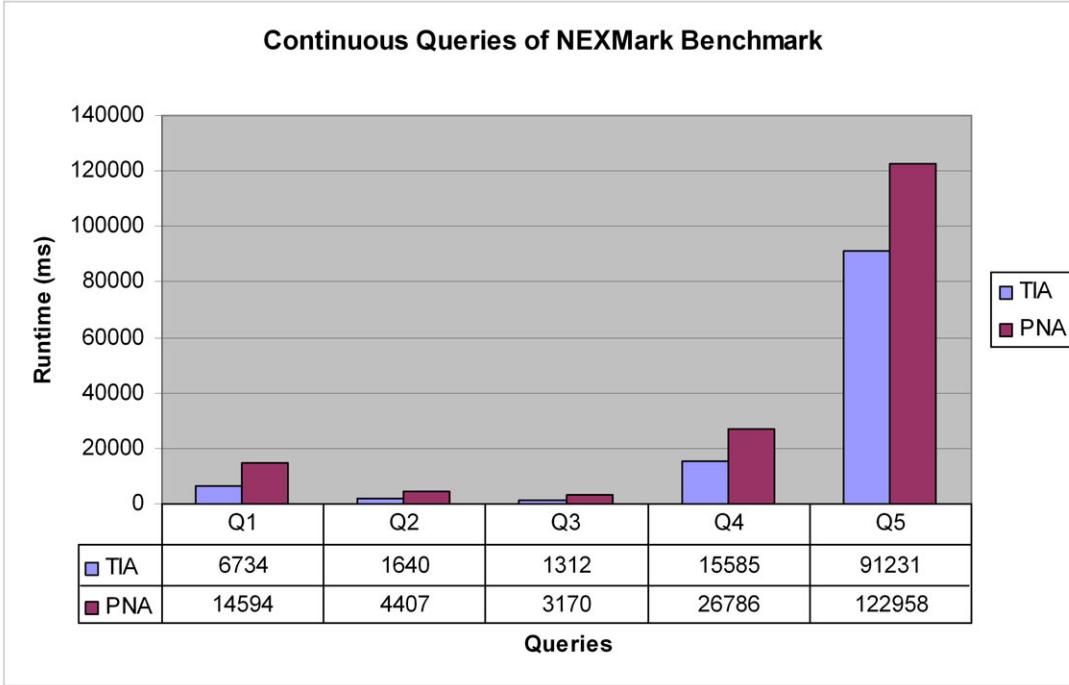


Figure 14.1: Total runtimes of NEXMark queries for the Time-Interval Approach (TIA) and the Positive-Negative Approach (PNA)

that the stream generator also simulates the bursty behavior of an online auction. Hence, it is often the case that multiple bids arrive at a single point in time.

To ensure a fair comparison, we have fully implemented PNA in our stream processing infrastructure PIPES. As a result, PNA and TIA rely on the same implementation of data structures and physical query plans are executed under the same DSMS runtime environment. Since PIPES and the benchmark data generator are publicly available, our experimental studies can be reproduced easily. Furthermore, this offers the interested reader the opportunity to delve into the algorithmic subtleties of our implementations, in particular the ones for PNA.

All experiments were executed on a Windows XP Professional workstation with a Pentium IV 3.0 GHz CPU and 1 GByte main memory using the Java 5 virtual machine. Our experiments investigate the output rates, memory consumption, and processing costs. To entirely reveal the processing costs of both approaches including all types of overheads, we pushed the input data as fast as possible through the query plans, while maintaining the original arrival order. As a consequence, the numbers for the stream rates correspond to the maximum input load the system could handle. We also measured the time required to execute a query at maximum system load, the so-called *system throughput*. Note that under maximum throughput application time and system time are no longer synchronous because the original arrival rate defined by the timestamps in the input streams does not correspond to input rate during execution. Here, the input rate depends on the processing capacity of the system and is significantly greater than the original stream rate because the

entire data set is available and processed as fast as possible. However, this methodology is commonly used to evaluate the efficiency of stream processing algorithms, see for instance [GÖ03a, BMWM05, GHM<sup>+</sup>07]. Furthermore, it correctly reflects the memory usage of operators during runtime because the application timestamps of the input streams were not modified. The acceleration neither affects the size of operator states nor the query answer as both solely depend on our temporal semantics (over application timestamps).

Figures 9.1 and 9.2 (see pages 56 and 57) show the plans for the queries of the NEXMark benchmark investigated here. The corresponding queries expressed in our query language can be found in Chapter 7. Figure 14.1 gives an overview of the total runtimes of the individual continuous queries for PNA and TIA.

### 14.3.1 Queries with Stateless Operators

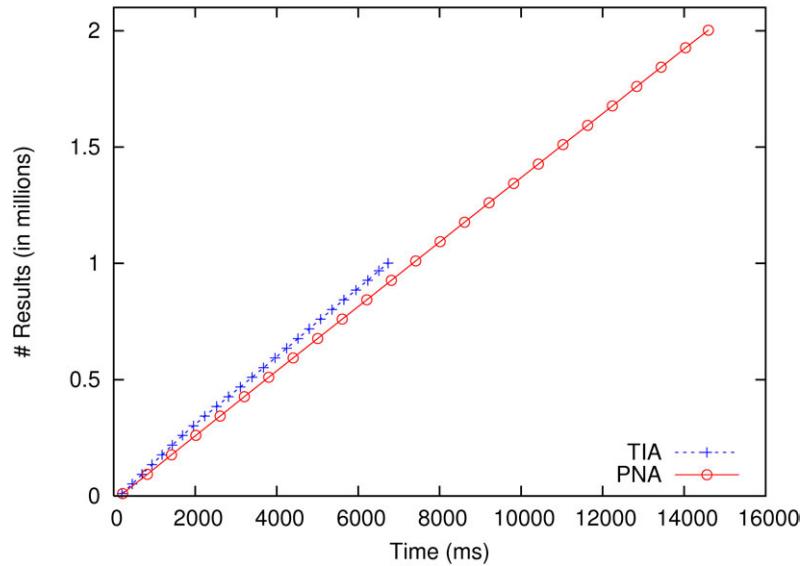
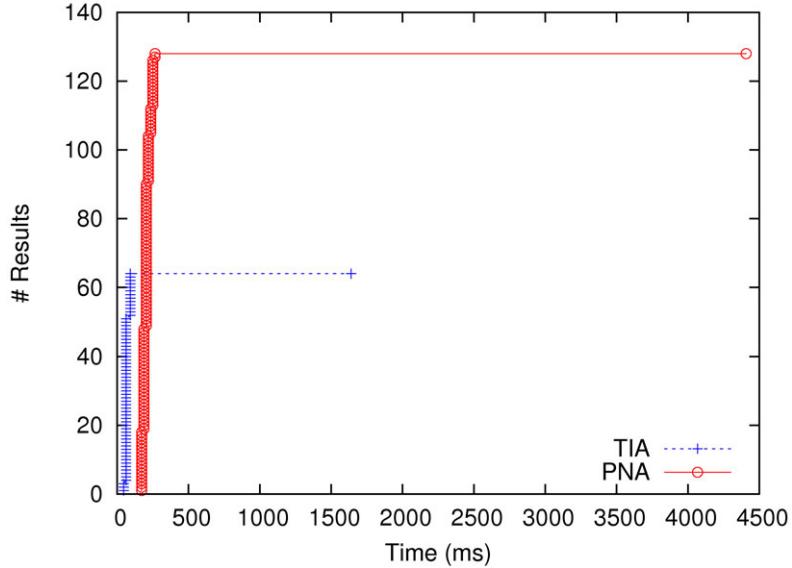


Figure 14.2: Currency conversion query ( $Q_1$ )

Query  $Q_1$  is the currency conversion query. Figure 14.2 shows the number of results over time. We counted both types of results for PNA – the positives and negatives – which explains why PNA produced twice as many results as TIA. Due to the fact that PNA has to convert twice as many elements, the CPU time spent for the mapping is doubled. The remaining time difference results from the window operator. TIA does not require a window operator at all due to the now-window. If we had placed such an operator, it would have computed the identity. In contrast, PNA necessitates a window operator to generate the negative elements. In comparison to the stateless TIA window operator, the PNA window operator is stateful. Even for a now-window, incoming elements need to be inserted into the state and trigger the release of negative elements from the state. These insertions and deletions cause additional processing costs which further slow down PNA.


 Figure 14.3: Selection query ( $Q_2$ )

Query  $Q_2$  selects all bids on a specified set of items, which is a filter over the bid stream. Figure 14.3 reveals the characteristics of this query. As all specified item identifiers are between 1007 and 2019, the bids for these items appear shortly after the start of the query. During the remaining runtime of the query no further bids on these items occur. Be aware that the slope of the curves corresponds to the output rate. At the beginning the output rate is extremely high and after this peak it remains zero. In addition, Figure 14.3 demonstrates the bursty nature of the input stream as multiple people bid on the same item in a short period of time. TIA is almost three times faster than PNA. A factor of two results from the doubled filter costs, the rest from the overhead of the PNA window operator as already explained for query  $Q_1$ . The processing overhead of PNA also causes query results to arrive slightly delayed, which is observable for the initial results in Figure 14.3. Compared to the first experiment, query  $Q_1$  is more expensive than  $Q_2$  because the map operator creates new elements, whereas the filter only forwards or drops incoming elements.

TIA is not only superior to PNA with regard to the processing costs, but also with respect to memory usage. For TIA, all operators in both plans are stateless and thus only allocate a negligible, constant amount of memory. The memory consumption of PNA depends on the window size due to the stateful window operator. Figure 14.4 plots the number of elements stored by the PNA window operator over time. For the now-window, the state contains between 0 and 17 elements, 6 elements on average, which again points out a feature of the benchmark, namely, that multiple bids can occur at the same time instant. If the window size in the queries is changed to 10 minutes, the state contains 97 elements on average. For a window of 5 hours, the state even consists of 2880 elements on average.

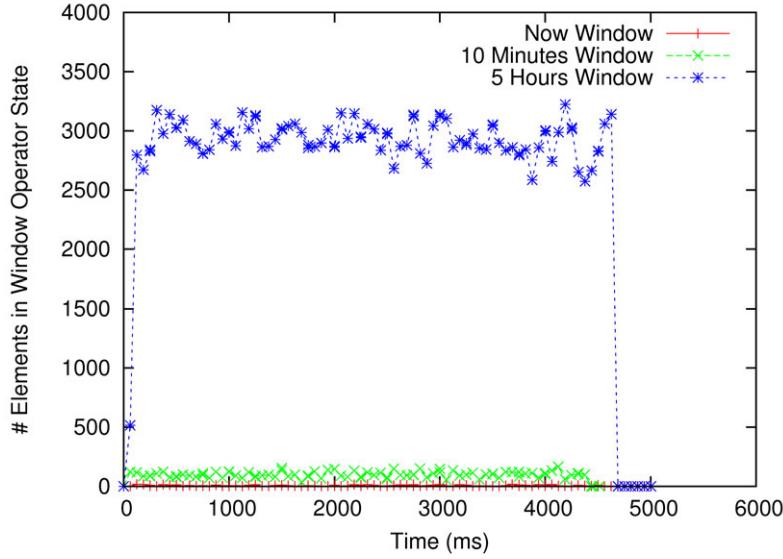
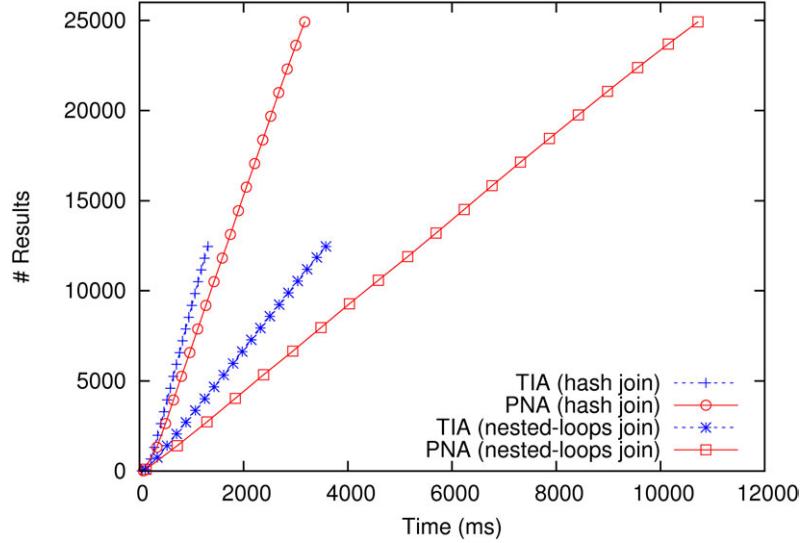


Figure 14.4: Memory allocation of time-based window operator for PNA

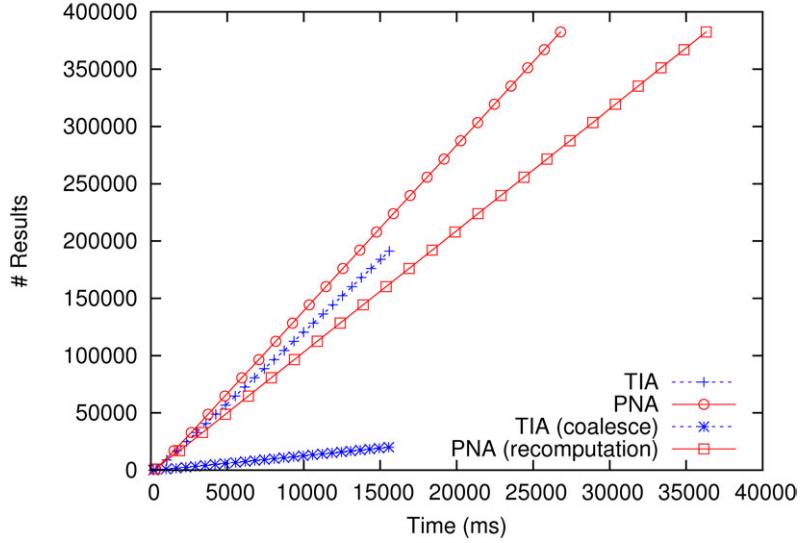
### 14.3.2 Queries with Stateful Operators

The third query  $Q_3$  reports all auctions that closed within five hours of their opening. The corresponding query plan consists of two window operators followed by a semi-join.<sup>1</sup> We run this query with different join implementations: hash and nested-loops. Although, for this query, a hash join is the best choice due to the equality predicate on item identifiers, we wanted to demonstrate that PNA behaves poorly for nested-loops joins. With regard to state maintenance, PNA and TIA perform the same number of insertions and deletions. For PNA, the presence of negative elements doubles the number of probings and result constructions, as the negative join results have to be computed. For hash joins, this computational overhead is of minor importance because probing costs can be considered as constant. PNA has the advantage that hashing can be used on arrival of a negative element to purge the corresponding positive element from the state with constant costs, whereas TIA maintains a separate temporal linkage to efficiently remove expired elements. As a result, PNA is still slower than TIA for hash joins, but less than the expected factor of two. This insight becomes particularly clear for queries  $Q_4$  and  $Q_5$ . However, for nested-loops joins, the doubled linear probing costs dominate (see Figure 14.5). Our nested-loops join implementation for PNA uses a hash table to benefit from fast insertions and removals, while probing is a sequential scan over all buckets. It might be surprising that TIA outperforms PNA by more than a factor of two. The reasons are: (i) TIA saves the costs for the now-window and (ii) we used our default join implementation with the priority queue at the output, which here is superior to PNA, aligning incoming elements according to temporal order at the join's input. We will investigate the effects of priority queue placement in more detail in Section 14.3.4.

<sup>1</sup>Now-windows are omitted for TIA.

Figure 14.5: Short auctions query ( $Q_3$ )

Query  $Q_4$  continuously computes the highest bids in the most recent 10 minutes. The physical plan given in Figure 9.1 resolves the subquery in the WHERE clause by applying a semi-join, implemented as a hash-join. With this query, we want to point out (i) the efficiency of different implementations for the scalar aggregation and (ii) the effect of coalesce as physical optimization. The curve for PNA in Figure 14.6 corresponds to the run that used an optimized implementation of the maximum aggregate, whereas the other curve for PNA (recomputation) used a linear scan over the entire state to compute the new aggregate on arrival of a negative element. The optimized aggregation organizes the state as an ordered list that aligns positive elements in ascending order by their values. While updates on the list cause logarithmic costs in the worst case, the maximum can be accessed with constant costs. The computation of a new aggregate for a negative element consequently causes logarithmic costs. In contrast, the recomputation variant uses a hash table to maintain the state. Aggregates for incoming positive elements are computed incrementally and the most recent aggregate is buffered. Whenever a negative element arrives, the new aggregate is computed by a linear scan over the updated state. The additional calls of the aggregate function increase the operator cost and consequently lead to a reduced output rate and a longer runtime as shown in Figure 14.6. If we compare TIA with the run for PNA using the optimized aggregation, it is observable that PNA performs better than expected. While TIA still outperforms PNA, the difference between both approaches is less than the intuitively expected factor of two (see also Figure 14.1). Both implementations rely on an ordered list. While TIA has to update all partial aggregates that overlap with an incoming element, the optimized PNA aggregation requires only a single call to the aggregate function for every incoming element. As in query  $Q_4$  it is likely that multiple bids overlap in the aggregation at the same time instant due to the FIFO expiration pattern generated by the window operator, PNA has less aggregate computations than TIA despite the overhead of a doubled input stream rate. Nevertheless,


 Figure 14.6: Highest bid query ( $Q_4$ )

we could reduce the number of invocation calls for the initializer, merger, and evaluator functions by optimizing Algorithm 9 towards a shared computation which avoids the redundant computation of identical aggregate values. Compared to the optimized PNA aggregation, the TIA implementation saves costs whenever an incoming element has a time interval that (i) is fully covered by a time interval in the state or (ii) does not intersect with any time interval in the state. These two cases occur frequently for derived streams, whose end timestamps are usually not monotonically increasing, e.g., in the output stream of a join. Here, the input stream of the aggregation satisfies a specific expiration pattern which is beneficial for PNA. The downside of the optimized PNA aggregation is an increased memory consumption because each aggregate function typically requires its own state information. The recomputation variant avoids this drawback at the expense of a raised number of aggregate function calls.

Besides the effects of the different PNA aggregation implementations, Figure 14.6 further demonstrates the benefit of coalesce. If the coalesce operator is applied to the output of the aggregation, it significantly reduces the rate of the join's right input stream (see also Figure 12.1 on page 113). This decreased stream rate, in turn, shrinks the cost of the join and also the join's output rate. Comparing the runtimes for the TIA plan and its extension with coalesce, the coalesce variant is slightly faster. This effect might be surprising at first glance due to the costs caused by the additional operator. However, the coalesce costs are offset by the reduced join costs. From the application perspective, it is reasonable to apply coalesce because during aggregation the maximum bid often remains unchanged when new bids on other items arrive.

Query  $Q_5$  reports the closing price and seller of each auction. We investigated two equivalent plans shown in Figure 10.2 (see page 63). Plan 1 computes the join with the OpenAuction stream after the grouping, plan 2 before. Figure 14.7 reveals that the first plan is superior to the second one. Hence, it is advisable to carry out the logical

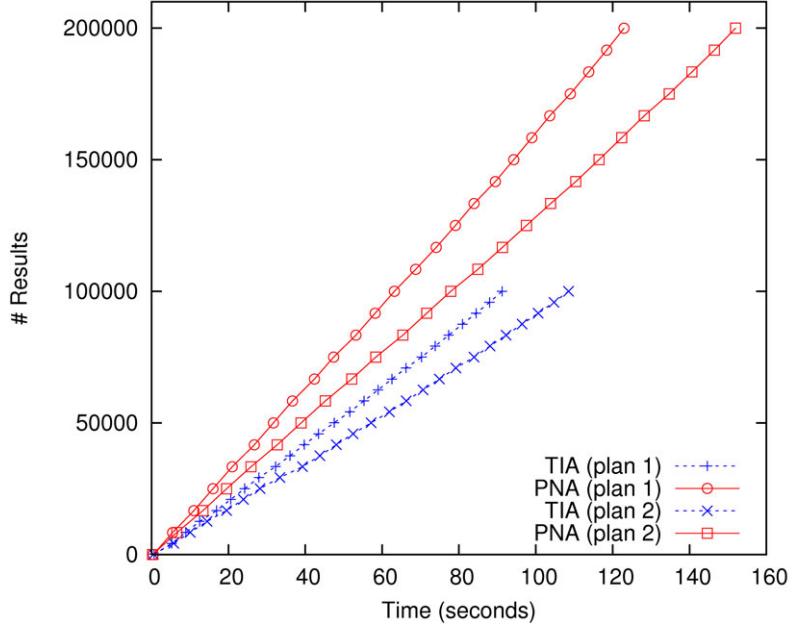


Figure 14.7: Closing price query (Q5)

optimization proposed in Chapter 10. Let us explain the reason. The first join produces a result for each bid and each open auction. All results have time interval length 1 due to the now-window over the `ClosedAuction` stream. According to the application semantics, all bids arrive before an auction is closed. Having a look at a single group, i. e., the join results for a single item, shows that all attached time intervals are equal. Moreover, the most recent join result for this item represents the maximum bid. If the grouping with aggregation is applied after the first join, it reduces the stream rate because it summarizes all join results referring to single item into a single aggregate. As a consequence, the cost of the second join, which depends on its input stream rates, is diminished.

One might propose substituting the grouping for a partitioned window of size 1. Unfortunately, the resulting plan would have an ambiguous semantics because the data contain multiple bids for the same item at the same point in time (see Section 8.2.2).

The rationale for the relatively short runtimes for PNA are the efficient hash-join that does not require a temporal linkage as its TIA counterpart and also the optimized aggregate computation of the maximum, which we already explained for query Q4.

### 14.3.3 Count-based Sliding Windows

Figure 14.8 demonstrates the downside of TIA, namely delayed results, when the query contains a count-based window. For this experiment, the plan consisted of a single count-based window over an input stream. We fed the input stream with 1000 random numbers. For the sake of clearness, we used a constant stream rate of 50 elements per second. Contrary to the other experiments pushing the data through the plan as fast as possible, we synchronized application and system time here to point out the delays in

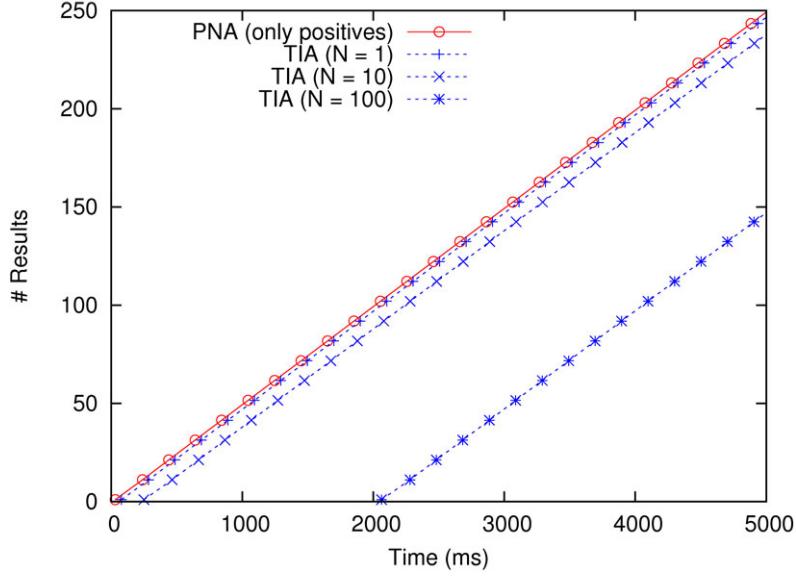


Figure 14.8: Latency of TIA for count-based windows

query response. While PNA outputs the positive elements instantly, independent of the window size, TIA suffers from increased latency for larger windows. Be aware that TIA outputs results at the same time as PNA emits negative results. For a count-based window of 100 elements, TIA generates the first result after 2 seconds due to the stream rate of 50 elements per second. Figure 14.8 captures only the first 5 seconds of the experiment in order to make the latency differences visible.

#### 14.3.4 Physical Optimizations

At the end of our experimental studies, we analyze the impact of two physical optimizations on query plans.

##### SweepArea Maintenance

The first optimization exploits the expiration pattern of an operator's input streams [GÖ05]. If the end timestamps are monotonically increasing, elements in the state expire in a FIFO manner. Most of our algorithms leverage SweepAreas that additionally align the elements according to end timestamps using a priority queue as secondary data structure. For a FIFO pattern, it is sufficient to replace the priority queue with a linked list (see Section 11.3.3). An insertion to the SweepArea appends the new element at the end of the list, a deletion removes the first element from the list. Our powerful SweepArea framework facilitates the exchange of the secondary data structure. Figure 14.9 visualizes the effect of this optimization for a windowed hash-join over two synthetic data streams. Each stream provided  $10^5$  uniformly distributed random integers in range  $[0, 1000]$ . The assigned timestamps simulated an inter-arrival time with a mean of 10 milliseconds

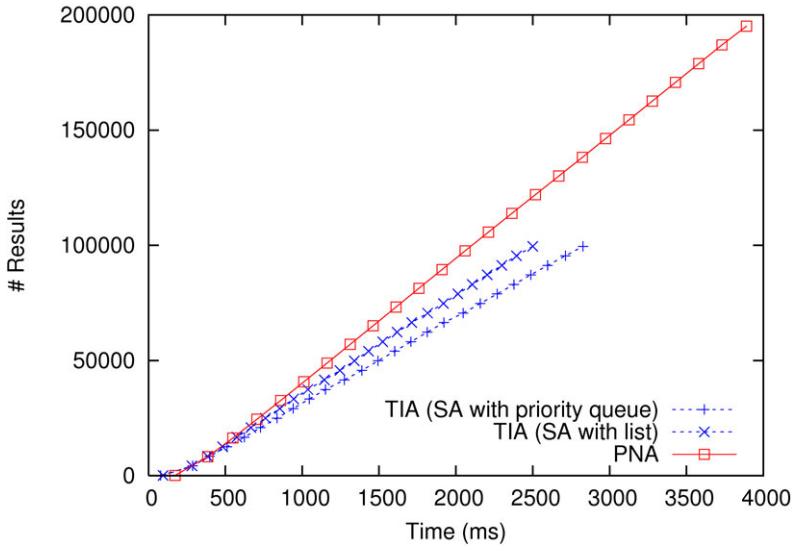


Figure 14.9: Adjusting SweepArea implementations to expiration patterns

based on a Poisson distribution. Both time-based windows had a size of 5 seconds. By measuring maximum throughput, Figure 14.9 shows that the SweepArea using the list as secondary data structure is faster than the queue variant. PNA does not allow such an optimization as it invalidates elements in the state by hashing on arrival of negative elements. Bear in mind that this physical optimization is not restricted to joins.

### Priority Queue Placement

Our last experiment investigates an optimization of the TIA join algorithm. Our default implementation uses a priority queue at the output to restore stream order (see Algorithm 6). In contrast, PNA brings the elements at the inputs into order. This is crucial to PNA because otherwise the hash-based invalidation of elements in the state would not be possible, which is one of the benefits of PNA. DSMSSs implementing PNA apply demand-driven query processing and connect operators with inter-operator queues. An operator consumes the elements from its input queues in temporal order. We can transfer this approach to data-driven, push-based processing by placing a priority queue at the input of the join. Whenever new elements arrive from one of the inputs, the queue rearranges them according to start timestamp order ( $\leq t_s$ ) prior to joining. To guarantee the ordering requirement of the output stream, only those elements can be extracted from the queue and joined whose start timestamp is  $\leq \min t_s$ , where  $\min t_s$  is defined as in Algorithm 4.

We found out that it is preferable for expansive joins to place the queue at the input, whereas it is better for selective joins to install the queue at the output. To illustrate this insight, we executed a windowed, binary hash-join and modified the join selectivity. For the first run, we fed each input stream with  $10^5$  uniformly distributed random integers in range  $[1, 1000]$ . For the second run, we shrank the range to  $[1, 100]$ . The window

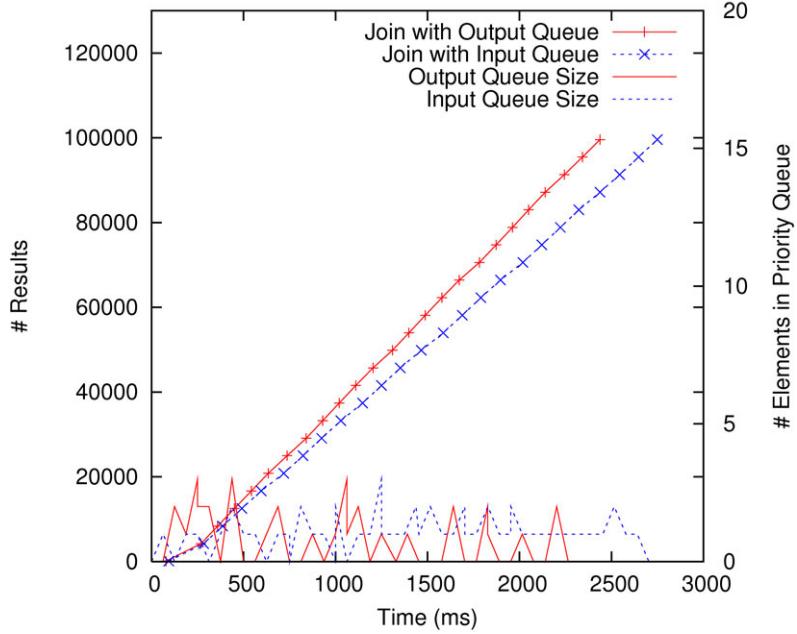


Figure 14.10: Priority queue sizes and output rates for a selective join

sizes and associated timestamps remained unchanged for both runs to ensure a fair comparison. We set the window size to 5 seconds and assigned the timestamps as done in the previous experiment, i. e., with a simulated average stream rate of 100 elements per second. To measure efficiency, we processed the data as fast as possible. At steady state each SweepArea of the join captured 500 elements on average. Because we run both experiments with identical input streams, the temporal selectivity of the join needs not be taken into account. For the first run, the probability that an incoming element joins with an element in the opposite SweepArea is 0.5. Because the join for the first run produces less than a single result for an incoming element on average, we denote the join as *selective* or *non-expansive*. On the contrary, a join, which outputs more than one result for each incoming element on average, is *expansive*. The second run represents an expansive join because an incoming element joins with 5 elements on average. Figures 14.10 and 14.11 refer to the first and second run respectively. For each run, they illustrate the output rates, total runtimes, and priority queue sizes. Let us consider the selective join first. If the queue is placed at the input, its average size is 1 since there is no application-time skew between the input streams. The average size of the queue positioned at the output is 0.5. Thus, compared with the queue at the input, less operations have to be performed. As a consequence, join costs are reduced, which in turn improves maximum throughput. The second run shows that due to the expansive join the queue at the output is not preferable because of the average size of 5 elements. A look at the output rates illustrates the increased costs of this join variant.

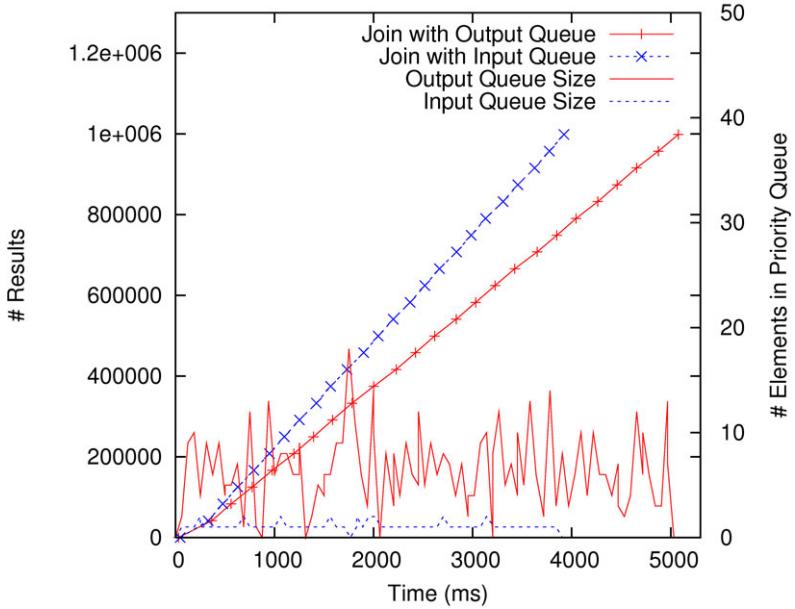


Figure 14.11: Priority queue sizes and output rates for an expansive join

## 14.4 Lessons Learned

TIA has definite advantages for *time-based* sliding window queries, which is the most common type of query in streaming applications:

- TIA roughly outperforms PNA by a factor of two. The reason is that TIA profits from lower processing costs because it does not have to handle negative elements.
- TIA needs to allocate less memory than PNA, as it does not have to store the elements in the window to generate negative elements.
- For queries composed of stateless operators, join, and duplicate elimination, results for TIA are available at the same time as the positive result arrives for PNA. TIA results display the full validity information for an element, i.e., not only the start but also the end of the validity. To get this information using PNA, the user has to wait for the corresponding negative element. TIA has a slightly delayed response time for grouping, aggregation, and difference because these operators cannot emit results as long as a temporal overlap with future elements is possible. However, our experiments demonstrate that this latency is negligible in general.
- We confirmed that the coalesce operator can be valuable for physical query optimization. Furthermore, we pointed out that a priority queue at the input of a join should be preferred for expansive joins, whereas placing the queue at the output is appropriate otherwise.

PNA profits from lower latency if *count-based* or *partitioned windows* are specified in a query. However, it is questionable to which extent these windows are used in practice because of their often ambiguous semantics.



# 15 Expressiveness

The use of a SQL-style declarative query language is a popular way for expressing continuous queries, see for instance [SQR03, JAF<sup>+</sup>06a, JAF<sup>+</sup>06b, MFHH05, TTPM02]. CQL allows the user to specify continuous sliding window queries over streams and relations (see Section 7.2.6). [ABW06] reveals that CQL is already more expressive than many other query languages. This chapter is aimed at proving the following theorem.

**Theorem 15.1.** *Our query language has at least the same expressive power as CQL.*

Section 15.1 proves this statement. Section 15.2 outlines directions for gaining additional expressive power by possible extensions of our query language and operator algebra.

## 15.1 Comparison to CQL

Just like CQL, our query language is based on SQL. Whenever our query language is used to formulate continuous queries, tuples in a stream are relational tuples and predicates and functions parameterizing the operators of our algebra adhere to the relational semantics. Recall that our approach is not restricted to relational structures in general, because tuples in a stream can be arbitrary composite objects to which operator functions and predicates are tailored. In the latter case, it is preferable to construct query plans from our operator algebra by means of a graphical user interface that allows users to pick out operators and specify code fragments for functions and predicates.

### 15.1.1 Query Translation

Given a CQL query over streams  $S_i$ ,  $i = \{1, \dots, n\}$ , and time-varying relations  $R_j$ ,  $j = \{1, \dots, m\}$ . The following steps describe how to translate any CQL query into an operator plan of our logical algebra.

1. Rewrite the CQL query into an equivalent query using `Rstream` as the only relation-to-stream operator. This step is always possible as indicated in [ABW06] since `Istream` und `Dstream` do not enhance the expressive power of CQL. An important rewriting rule frequently applied is to replace any unbounded time-based window followed by `Istream` with a now-window followed by `Rstream`. Expressed in CQL notation,

```
SELECT Istream(L) FROM S [Range Unbounded] WHERE C  
≡  
SELECT Rstream(L) FROM S [Now] WHERE C
```

where  $L$  is any select-list,  $S$  is a stream reference, and  $C$  is any condition.

2. Transform each time-varying input relation  $R_j$  into a raw stream by applying the `Rstream` operator. Formally, the raw stream is fed with  $\bigcup_{t \geq 0} (R_j(t) \times \{t\})$ , where  $R_j(t)$  denotes the instantaneous relation at time instant  $t$ ,  $t \in T$ . An instantaneous relation can be viewed as a snapshot.
3. Transform each input stream  $S_i$  into a logical stream. The definition of a *stream* in [ABW03, ABW06] matches with our notion of a raw stream for the relational case. Applied to a raw stream, function  $\varphi^{r \rightarrow l}$  generates the corresponding logical stream. Do the same for any raw stream obtained from a time-varying input relation.
4. Place the windowing operators. CQL requires the conversion of all streams into time-varying relations in order to apply the relational operators for processing. This is achieved by specific unary stream-to-relation operators, which are sliding window operators. CQL supports time-based, tuple-based, and partitioned windows. Our logical operator algebra contains an equivalent for each of these windows. In order to achieve a semantically equivalent query result, we substitute each stream-to-relation operator with the corresponding logical window operator.<sup>1</sup>
5. Set up the standard operators. Our logical operator algebra provides a stream-to-stream counterpart of the same arity for every relation-to-relation operator. For this reason, the relational operators in the plan for the CQL query can simply be replaced with their stream analogs.

### 15.1.2 Formal Reasoning

The above query translation is applicable and produces snapshot-equivalent results because of the following semantic properties of our window and standard operators.

#### Window Operators

Let  $S$  be a (raw) stream.

**Lemma 15.1.** *Let  $w$  be the window size of the time-based window.*

$$\forall t \in T. Rstream(S [Range w])(t) = \tau_t(\omega_w^{\text{time}}(\varphi^{r \rightarrow l}(S)))$$

*Proof.* Consider an arbitrary time instant  $t \in T$  and a tuple  $e$ . Let tuple  $e$  occur  $n$ -times in the instantaneous relation  $Rstream(S [Range w])(t)$  and  $m$ -times in the multiset  $\tau_t(\omega_w^{\text{time}}(\varphi^{r \rightarrow l}(S)))$ , where  $n, m \in \mathbb{N}$ . We have to show that  $n = m$ .

From the definition of the logical time-based window follows that the multiplicity of a tuple  $e$  at time instant  $t$  is given by the sum of all value-equivalent elements being valid at a time instant in the interval  $[\max\{t - w + 1, 0\}, t]$ . If tuple  $e$  occurs  $m$ -times in the snapshot of the window's output stream at instant  $t$ , then there exist  $m$  elements with tuple  $e$  in the snapshot of the window's input stream. According to the definition, all of these tuples necessarily have to be tagged with a timestamp in range  $[\max\{t - w + 1, 0\}, t]$ .

---

<sup>1</sup>We use the term *count-based* window instead of *tuple-based* window.

As the change in representation from a raw stream into its logical counterpart, and vice versa, does not have any semantic effects, all these tuples must occur with the associated timestamps in the input stream  $S$ .

If tuple  $e$  occurs  $n$ -times in the instantaneous relation  $Rstream(S [Range w])(t)$ , then the input stream  $S$  must contain all  $n$  tuples. According to the definition of the time-based window, the associated timestamps of these tuples have to be  $\leq t$  and  $\geq \max\{t - w + 1, 0\}$  (see [Ara06]). Hence, it follows that stream  $S$  contains  $n$  elements having tuple  $e$  with a timestamp in  $[\max\{t - w + 1, 0\}, t]$ .

In summary, it follows from the right side of the equation that input stream  $S$  must contain  $m$  elements built of a tuple  $e$  with a timestamp in range  $[\max\{t - w + 1, 0\}, t]$ , whereas the left side implies that stream  $S$  contains  $n$  elements having the same properties. As a consequence,  $n = m$ .  $\square$

**Lemma 15.2.** *Let  $N$  be the window size of the count-based window.*

$$\forall t \in T. Rstream(S [Rows N])(t) = \tau_t(\omega_N^{\text{count}}(\varphi^{r \rightarrow l}(S)))$$

*Proof.* Consider an arbitrary time instant  $t \in T$  and a tuple  $e$ . Let tuple  $e$  occur  $j$ -times in the instantaneous relation  $Rstream(S [Rows N])(t)$  and  $k$ -times in the multiset  $\tau_t(\omega_N^{\text{count}}(\varphi^{r \rightarrow l}(S)))$ , where  $j, k \in \mathbb{N}$ . We have to show that  $j = k$ .

Without loss of generality we require ties to be broken in the same fashion if there exist several elements with the  $N$ -th most recent timestamp. According to their definition both window operators capture the  $N$  tuples of  $S$  with the largest timestamps  $\leq t$ . Because tuple  $e$  occurs  $j$ -times in the output of the tuple-based window at  $t$ , the window must contain tuple  $e$   $j$ -times at time instant  $t$ . The same argumentation applied to the logical count-based window at  $t$  implies that tuple  $e$  has to be covered  $k$ -times by the window at  $t$ . As both window operators are fed with identical inputs and address a deterministic subset over this input, namely the  $N$  tuples with the largest timestamps,  $j$  has to be equal to  $k$ .  $\square$

**Lemma 15.3.** *Let  $A_1, \dots, A_k$  be the partitioning attributes, and  $N$  be the window size.*

$$\begin{aligned} \forall t \in T. Rstream(S [\text{Partition By } A_1, \dots, A_k \text{ Rows } N])(t) \\ = \tau_t(\omega_{A_1, \dots, A_k, N}^{\text{partition}}(\varphi^{r \rightarrow l}(S))) \end{aligned}$$

*Proof.* The logical partitioned window is a derived operator (see Chapter 8). It first partitions stream  $S$  into substreams, then evaluates a count-based window of size  $N$  on each substream, and finally merges the substreams into a single output stream. If each of these three steps produces semantically equivalent results, then the entire result will be semantically equivalent. The partitioning generates snapshot-equivalent substreams because the grouping function complies with the definition in [ABW06]. Due to Lemma 15.2, the output streams of the count-based windows match at snapshot-level. As the logical union is snapshot-reducible to its relational analog, the union over those substreams produces equivalent multisets for any snapshot.  $\square$

## Standard Operators

Our logical algebra provides a stream analog for every relation-to-relation operator in CQL. Be aware that relation-to-relation operators process time-varying relations, not classical relations [ABW06]. Every stream operator in our logical algebra produces snapshot-equivalent results to its relation-to-relation counterpart.

**Lemma 15.4.** *Let  $R_1, \dots, R_m$  be the time-varying input relations of an  $m$ -ary relation-to-relation operator  $op_{TR}$ . Let  $ops$  be the corresponding stream-to-stream operator in our logical operator algebra. We have to show that for any time instant the output of  $ops$  is semantically equivalent to the output of  $op_{TR}$ . Formally:*

$$\begin{aligned} \forall t \in T. \tau_t(ops(\varphi^{r \mapsto l}(Rstream(R_1)), \dots, \varphi^{r \mapsto l}(Rstream(R_m)))) \\ = op_{TR}(R_1, \dots, R_m)(t) \end{aligned}$$

*Proof.* To express a continuous query over time-varying relations with our logical operator algebra, it is necessary to transform all relations into logical streams first. For a time-varying input relation  $R_j$ ,  $j \in \{1, \dots, m\}$ ,  $\varphi^{r \mapsto l}(Rstream(R_j))$  returns the corresponding logical stream. After that, a stream-to-stream operator  $ops$  can be applied to the logical input streams. Let  $op_R$  be the counterpart of  $ops$  in the extended relational algebra.

$$\begin{aligned} \tau_t(ops(\varphi^{r \mapsto l}(Rstream(R_1)), \dots, \varphi^{r \mapsto l}(Rstream(R_m)))) \\ = op_R(\tau_t(\varphi^{r \mapsto l}(Rstream(R_1))), \dots, \tau_t(\varphi^{r \mapsto l}(Rstream(R_m)))) \\ = op_R(R_1(t), \dots, R_m(t)) \\ = op_{TR}(R_1, \dots, R_m)(t) \end{aligned} \tag{15.1}$$

The first equation in (15.1) results from the snapshot-reducibility property of standard operators in our algebra (see Chapter 10). The output stream of the  $Rstream$  operator applied to a time-varying relation  $R_j$  contains an element  $(e, t)$  whenever tuple  $e$  is in  $R_j$  at time instant  $t$ . Formally,  $Rstream(R) = \bigcup_{t \geq 0}(R(t) \times \{t\})$ . The expression  $\tau_t(\varphi^{r \mapsto l}(Rstream(R_j)))$  can be simplified to the instantaneous relation  $R_j(t)$ , because it first generates a stream from  $R_j$  providing all tuples valid at their associated time instants, then maps this stream to our logical representation, and finally projects to the snapshot at instant  $t$ . Hence, it just computes the multiset of tuples being valid in the time-varying relation  $R_j$  at instant  $t$ . Rewriting yields the second equation. The third equation directly follows from the snapshot-reducibility of relation-to-relation operators in CQL, formulated in [ABW06] as follows: “At any time instant  $t \in T$  the instantaneous output relation at instant  $t$  has to be computable from the instantaneous input relations  $R_1(t), \dots, R_m(t)$ .”  $\square$

## 15.2 Beyond CQL

In this section we sketch possible extensions of our language and operator algebra. As already mentioned, our approach is not restricted to relational structures, which is already a distinction from CQL. We can imagine manifold ways to enrich our approach.

- The definition of *novel window operators* would give the user more flexibility in specifying windowing constructs, e. g., support for landmark windows and fixed-band windows [PS06], or the general type of predicate windows introduced in [GAE06].
- For optimization purposes, we exploit snapshot-reducibility of standard operators. If we give up snapshot-reducibility, we could allow standard operators to access the temporal information attached to tuples. Consequently, it would be possible to define *complex temporal operators* over data streams combining information across different time instants, e. g., temporal stream joins derived from the temporal joins in [GJSS05]. This kind of extension tends to be a promising approach to efficiently dealing with complex event stream processing [DGH<sup>+</sup>06, Luc02].
- Incorporating *time granularity* into query processing is also a fruitful extension since users are often not interested in query answers at finest time granularity. Therefore, we developed a granularity conversion operator that allows for the evaluation of queries at a coarser user-defined time granularity [CKSV06] (see also Chapter 20 in Part III). In addition, Section 17.1 enhances our window operators to support different slide granularities.
- Entirely novel operators might be added to our algebra, optionally along with a grammatical extension of our query language. Potential candidates are the *BSort* and *Resample* operators proposed in [ACC<sup>+</sup>03].
- Enhancing our query language with support for *user-defined aggregates* (UDAs) natively coded in SQL facilitates the specification of complex aggregate functions [LWZ04] and data mining functionality [WZL03]. If we are able to support arbitrary UDAs, our query language will be  $\mathcal{NB}$ -complete [LWZ04]. However, we need further theoretical results on how we can map UDAs into our algebra appropriately. A suitable starting point is the generalized variant of our map operator that additionally incorporates the temporal information associated with tuples. As we permit the use of arbitrary mapping functions, including higher-order functions, a UDA can be expressed as a composition of functions. In our library XXL [BBD<sup>+</sup>01], the *invoke* method of a function may contain arbitrary Java code fragments. Furthermore, functions can have a state. Hence, the generalized map operator provides the essential building blocks for UDAs.

At the end of this outlook we want to mention that there is a general tradeoff between expressive power and the range of optimizations applicable. Recall that conventional transformation rules are restricted to snapshot-reducible subgraphs (see Chapter 10). Though it might be attractive to drop the snapshot-reducibility property of standard operators for flexibility and expressiveness reasons, it significantly complicates query optimization.



# 16 Related Work

A short version of Part II appeared in [KS05]. This paper includes neither our query language specification, the algorithms of our physical operator algebra, the discussion on the expressiveness of our approach, the detailed comparison with the positive-negative approach, nor the experimental evaluation. Due to space limitations in the conference paper, we had to focus on the logical algebra, i. e., the semantic foundation of our approach and the basic idea of how to implement this semantics using time intervals. In this work, we elaborate on the steps from query formulation to query execution with the aim to substantiate the practicability of our approach. To the best of our knowledge, no paper has been published that discusses stream query processing in such a coherent way giving seamless insight into the individual processing steps and their correlation.

Section 16.1 points out the inconsistent use of timestamps in stream processing approaches. While Section 16.2 gives an overview on related stream algebras, Section 16.3 reveals the connections to the extended relational and temporal algebra. We survey related stream processing systems in Section 16.4 and outline further related approaches in Section 16.5.

## 16.1 Use of Timestamps

As highlighted in the introduction (see Chapter 5), most stream processing approaches exploit the timestamps provided by stream elements. For instance, these timestamps are used to define windows or control expiration [BBD<sup>+</sup>02, GÖ03b]. In recent years we have observed a substantial divergence in the use and creation of timestamps among the various stream processing approaches. [BBD<sup>+</sup>02] and [CCC<sup>+</sup>02] propose that the user should specify a function for creating the timestamp assigned to a result of an operator. Such a flexible technique has the drawback that query semantics depends on user-defined functions as downstream operators are sensitive to timestamps. Without the capability to characterize the effects of user-defined functions on timestamps, query optimization becomes nearly impossible. Babcock et al. suggest the join result to be associated with the maximum timestamp of the two contributing join elements [BBDM03]. In contrast, the join in [CF02] takes the minimal timestamp. The multi-way stream join proposed in [GÖ03a] keeps all timestamps of qualifying elements in the resultant join tuple. The dynamic optimization for continuous queries addressed in [ZRH04] relies on specific temporal order relations because elements of derived streams have combined timestamps, i. e., multiple timestamps. Although the paper claims that the methods are applicable to arbitrary query plans, we are convinced that operators like aggregation and duplicate elimination incur serious problems. The reason is that the number of timestamps associated to a result in the output stream is no longer constant. Unfortunately, the

timestamp order lemma in [ZRH04] assumes elements within a stream to have the same number of timestamps. Nevertheless, keeping all timestamps in operator results works fine for SPJ queries but incurs the following disadvantages compared to our approach: (i) The size to store the temporal information of a stream element is not constant because every join in a sequence of joins adds a new timestamp attribute. As a result, the overall memory utilization will be slightly increased. (ii) More complex purging conditions are required whose evaluation raises operator processing costs.

This lack of consensus in the use of timestamps inspired us to develop an operator algebra that (i) consistently deals with timestamps and (ii) provides a precisely defined, sound semantics as a foundation for query optimization. Our physical algebra shows that it is sufficient to tag tuples with time intervals to accomplish a cohesive operator model.

## 16.2 Stream Algebras

A lot of papers focus on individual stream operators, e. g., on window aggregates [LMT<sup>+</sup>05, ZGTS02, DGGR02] and window joins [GÖ03a, DMRH04, KNV03, MLA04], rather than on a complete stream algebra. These approaches often rely on specific semantics and operator properties. Due to their restricted scope, they disregard aspects of the larger system picture such as operator plans and operator interaction. Unlike these approaches, we strive to establish a powerful algebra for stream processing that consistently unifies the various stream operators along with an appropriate stream model.

Despite the high relevance of a general-purpose algebra for continuous queries over data streams, only a few papers have addressed this complex and broad topic so far. [ABW03, ABW06] propose the abstract semantics of CQL. The authors motivate the language and its features by drawing various examples from the Linear Road Benchmark [ACG<sup>+</sup>04]. Although our work is closely related to [ABW06], there are important distinctions. (i) Our focus is not on the query language but rather on the operator semantics and implementation. (ii) Our logical algebra defines our operator and query semantics in a concrete and formal manner, rather than in an abstract manner. (iii) Although Arasu et al. sketch query execution, they do not discuss important implementation aspects such as algorithms. However, the implementation of these algorithms is non-trivial for stateful operators. Conversely, we proposed a novel and unique physical algebra that illustrates the implementation of our time-interval approach. (iv) We also implemented the positive-negative approach of CQL, which is explained in more detail by another research group [GHM<sup>+</sup>07], and compared it with our approach. (v) We proved that our query language is at least as expressive as CQL. Besides using the query language, a user is able to construct query plans with a graphical interface. Due to our generic and flexible operator design based on functions and predicates and the fact that our streams are not limited to relational tuples, our approach can easily be enhanced with more powerful operators. (vi) We introduced novel physical operators, namely split and coalesce. While the split operator is crucial to ensure a correct semantics for windowed subqueries, coalesce can be used for reducing stream rates. (vii) We explicitly discussed logical and physical query optimization issues, including novel transformation rules and the effects of coalesce and expiration patterns. A denotational semantics for CQL

is presented in [AW04a]. Unfortunately, the meaning functions are only illustrated for some example operators.

Tucker et al. [TMSF03] propose an interesting, alternative approach for dealing with unbounded stateful and blocking operators. Instead of using windowing constructs, a priori knowledge of a data stream is used to embed *punctuations* in the stream with the aim to permit the use of the aforementioned class of operators. A data stream is modeled as an infinite sequence of tuples. A punctuation can be considered as a predicate on stream elements that must evaluate to false for all elements following the punctuation. More precisely, a punctuation is an ordered set of patterns, where each pattern refers to an attribute of a tuple. Punctuations in a stream signal the end of substreams. Operators are stream iterators that access the input stream elements incrementally and make use of the embedded punctuations. Operators output a punctuated stream and store a state. Operator semantics and correctness depend on three behavior functions. The *pass* behavior controls which elements can be output based on the punctuation received. The *propagate* behavior describes additional punctuations that can be emitted according to the punctuation and elements received. The *keep* behavior specifies the portion of the local state that must be kept based on the punctuation. To obtain stream iterators that are appropriate counterparts of the corresponding relational operators, the authors define an invariant for every behavior function of a stream iterator. While the paper gives proofs for duplicate elimination and difference, the correctness of other operators including join and aggregation has to be confirmed in future work. The paper points out several significant open issues to validate the practicality of the punctuation approach. It is important to elaborate on how punctuations can be generated and embedded into raw streams automatically. Furthermore, the system should control how frequently punctuations are embedded with the goal to balance between the gain of punctuations and the resulting computational overhead. The major open problems address the questions whether a given query can profit from punctuations and to which extent, and which punctuation scheme will be beneficial. Nevertheless, punctuations could also be beneficial to our approach. Enhancing our operators for punctuated streams would be a general concept for optimizing state maintenance. We could substitute heartbeats for punctuations. Moreover, punctuations could reduce the size of the operator state not only according to temporal conditions, but also according to value-based conditions on tuples. For example, consider an equi-join and a punctuation indicating that value  $x$  will not appear in the join attribute in the remainder of an input stream. Then, it is not necessary to keep any element in the state having attribute value  $x$ .

A one-pass technique for sliding window aggregates is presented in [LMT<sup>+</sup>05]. The technique relies on a framework for defining window semantics. A sliding window is seen as a moving view that partitions a stream into subsets that may overlap, called *window extents*. The result of a window aggregate is computed for each window extent. To achieve this, each window extent is identified by a unique window-id. For each incoming stream element, a range of window-ids is calculated, namely, the window extents to which it belongs. Conceptually, the technique transforms windowed aggregation into conventional grouping with aggregation by attaching an additional grouping attribute. Another feature of this technique is that it does not require a specific physical stream order. Instead, it uses punctuations to deal with stream disorder [TMSF03].

Patroumpas and Sellis [PS06] describe a formal framework for expressing windows in continuous queries over data streams. They reveal several common properties for windows and present a reasonable taxonomy of the prevalent window variants. A parameterized scope function allows the specification of the window’s extent and its progression. Our logical window operators comply with the specifications given in [PS06], but [PS06] covers a variety of further windows, e. g., landmark or fixed-band windows. Similar to [ABW06], the combination of windows with relational operators yields the windowed analogs. These windowed operators process timestamped tuples as inputs and generate a time-varying relation as output. Note that [PS06] only addresses semantic issues and not the implementation.

### 16.3 Extended Relational and Temporal Algebra

Our work exploits the well-understood relational semantics and its temporal extensions to establish a sound semantic foundation for continuous queries over data streams. Hence, our work is closely related to these research areas. In the *extended relational algebra*, operator semantics is defined over *multisets* (bags) [DGK82, Alb91, GUW00]. A logical stream extends the multiset representation of a relation with an additional timestamp attribute. Every standard operator in our stream algebra is snapshot-reducible to its relational analog. This property ensures semantic compliance and allows us to carry over the algebraic equivalences from the extended relational algebra to our stream algebra. Despite the different types of queries, namely one-time versus continuous queries, there are further similarities between query processing in conventional DBMSs and in our approach. We also distinguish between a logical and a physical operator algebra and apply the approved steps from query formulation to query execution.

Since elements of raw streams have timestamps and therefore ordering, our semantics is related to *temporal databases* [TCG<sup>+</sup>93, SA85, ÖS95] and time-series databases [FRM94, Cha96]. We found the temporal algebra with support for duplicates and ordering proposed in [SJS00, SJS01] to be an appropriate starting point for our research. We extended this work towards continuous query processing over data streams as follows. (1) Slivinskas et al. [SJS01] do not provide a logical algebra as an abstraction of their semantics. They rather specify the semantics of their operations from an implementation point of view using the  $\lambda$ -calculus. However, we are convinced that separating the logical from the physical algebra entails several advantages. Our logical stream algebra defines operator semantics in an explicit, precise, and descriptive manner. Since a logical stream can be viewed as a potentially infinite, temporal multiset, such an algebra is beneficial to the temporal database community as well. It clearly illustrates the snapshot semantics because our logical algebra addresses time instants. We already mentioned this advantage in Section 8.1, e. g., the stream difference simply reduces to the difference in multiplicities of value-equivalent elements at every time instant. This definition is much more intuitive and compact than the one given in [SJS01] based on the  $\lambda$ -calculus. Moreover, the logical algebra can be leveraged to prove semantic equivalences of operator plans while abstracting from a specific implementation. (2) Despite the similarity of using time intervals to denote validity, the temporal operator algebra in [SJS01] does not satisfy the

demanding requirements of stream processing applications due to the following reasons. (i) The algebra is not designed for continuous queries. (ii) The operators assume the inputs to be finite. (iii) The implementation makes use of nested-loops techniques and therefore is blocking over unbounded inputs. (iv) The operator output is not computed incrementally at delivery of a new element. (v) The algebra does not provide any windowing constructs. Nonetheless, we believe that this temporal algebra is a fruitful and suitable basis which can be enhanced for stream processing. Because of this, we have introduced the notion of physical streams and have developed an appropriate physical stream algebra. A physical stream can be considered as a potentially infinite temporal relation ordered by start timestamps. Unlike temporal relations, the start and end timestamp attributes are not part of the stream schema. The development of our physical operators was a non-trivial task. While substituting the demand-driven operators in [SJS01] for data-driven ones [Gra93], we had to tackle the aforementioned problems. This paradigm shift includes that our operators exploit physical stream order and temporal expiration to ensure nonblocking behavior for stateful operators. In addition, the results of our standard operators have to be multiset-snapshot-equivalent to their temporal analogs to benefit from the extensive set of transformation rules specified in [SJS01]. Since we combine window operators with standard operators in query plans to achieve continuous query semantics, the novel window operators had to be implemented accordingly. Moreover, our physical algebra provides a novel split operator to cope successfully with nested window queries. A further enhancement is the use of the SweepArea concept in stateful operators to attain generic, yet efficient implementations. Eventually, we added novel transformation rules for the window operators.

Since our stream operators produce a snapshot-equivalent output to the temporal operators presented in [SJS01], our approach benefits from [SJS01] with regard to query optimization by making a plethora of conventional and temporal transformation rules applicable in the stream context, including the existing work on temporal query optimization presented in [GS90, LM93]. Altogether, our approach establishes a sound semantic basis for query optimization over data streams. The generality and comprehensiveness are major differences between our approach and other research addressing optimization issues in stream processing, e.g., [VN02, KNV03, AH00]. Those papers usually focus on a restricted operator set consisting of selection, projection, and join, and suggest cost models rather than discussing semantic aspects.

The formal framework for logical data expiration presented in [Tom03] explores techniques to limit the growth of historical data in warehouses by identifying parts of the data no-longer needed. It distinguishes between query-independent expiration policies and query-driven data expiration. Approaches belonging to the latter category remove unnecessary data based on the queries asked over the collection of data and the progression of this collection. In our physical algebra, window operators determine the expiration of tuples in stateful operators downstream. Due to the ordering property of physical streams, unnecessary elements in the state can be identified and removed. Punctuations [TMSF03] and inferring stream constraints [BSW04] are alternative techniques to indicate expiration of stream elements. [SJS06] incorporate the notion of expiration time into relational data management. It turns out that (materialized) non-monotonic operator and algebra expressions require recomputation because results may become invalid after data expire

in the base relations. In comparison to our approach, the user defines the expiration time by specifying the window in a continuous query, whereas expiration time in [SJS06] is given by the data source.

## 16.4 Stream Processing Systems

[BBD<sup>+</sup>02] and [GÖ03b] give a comprehensive survey on work related to data streams and continuous queries. Here we focus on publications addressing the language, semantics, and implementation of continuous queries over data streams. We classified them according to system prototypes.

### 16.4.1 STREAM

In recent years various SQL-like *query languages* have been proposed to formulate continuous queries. CQL used in the STREAM system is the closest to our language [ABW06]. While both enrich native SQL with window constructs, our language sticks more firmly to the standard SQL syntax. We just restricted the window constructs defined in SQL:2003 to preceding windows and shifted the specification from the SELECT to the FROM clause. We added support for *window slides* because sometimes continuous queries explicitly specify the granularity at which a window slides (see [SQR03, TTPM02]). Unlike CQL, our language allows processing streams directly rather than turning them into time-varying relations and back into streams. As a consequence, window operators are stream-to-stream operators in our case. Furthermore, our language does not require the syntactical extensions of CQL to support relation-to-stream operators. Keeping the stream perspective makes our language more appealing and intuitive, particularly in the case of windowed subqueries. Nevertheless, we make use of the relation-to-stream operators whenever data obtained from time-varying relations, e. g., materialized views [GM95], has to be combined with streams. Section 7.2.6 discussed the pros and cons of our query language in comparison with CQL. Independent of the user preferences for either query language, we proved that our language is at least as expressive as CQL (see Chapter 15). Moreover, Section 17.2 illustrates that it is even possible to use CQL as query language on top of our physical operator algebra. The main difference between STREAM and PIPES is not the query language and semantics but rather the *implementation* and *query execution*. STREAM employs the positive-negative approach, whereas PIPES prefers the time-interval approach (see Chapter 14 for a comparison). In STREAM [MWA<sup>+</sup>03], query plans are composed of operators connected with inter-operator queues. Synopses store the intermediate state similar to our SweepAreas. The processing of stream elements is implemented in a pull-based manner. Hence, an operator consumes elements from its input queues in temporal order and adds the results to its output queue. The scheduling strategy determines the order in which operators are executed, e. g., the *Chain* strategy [BBDM03]. In contrast to STREAM that uses a single thread for scheduling, PIPES relies on a multi-threaded architecture and push-based processing [CHK<sup>+</sup>07].

### 16.4.2 TelegraphCQ

*TelegraphCQ* [CCD<sup>+</sup>03] is a continuous query processing system for data streams arisen from a complete redesign and reimplementation of its predecessors *Telegraph* [SMFH01], *CACQ* [MSHR02], and *PSoup* [CF02]. *TelegraphCQ* primarily addresses scheduling and resource management for shared queries, adaptive query execution, Quality of Service (QoS) support, and parallel cluster-based processing and distribution. The dataflow modules of *Telegraph* can be compared with our operators. *Telegraph* provides a module for every operator of the extended relational algebra, but does not contain any window operators. *Fjords* [MF02] enable the communication between the dataflow modules either in a pull-based (synchronous) or push-based (asynchronous) manner. *Telegraph* does not rely upon a traditional query plan, but employs adaptive routing modules to control dataflow. *Eddies* [AH00] reoptimize execution plans continuously by routing single elements to query operators. This is contrary to our work that offers query optimization at a coarser level. Another interesting operator not supported in our approach is the *juggle* operator [RRH99]. This interactive operator dynamically reorders incoming elements according to user preferences. It allows the user to speed up processing in areas of interest at the expense of other areas. *Flux* [SHCF03] is a routing module that distributes elements among machines in a cluster to benefit from parallel processing. Due to the pipelined dataflow and the multi-threaded architecture, *Flux* represents a promising optimization for PIPES towards distribution and scalability. The flexible routing in *Telegraph* necessitates separate modules for state maintenance, called *State Modules* (SteMs) [RDH03]. In terms of functionality, SteMs are comparable with SweepAreas at first glance because they support build, probe, and eviction operations. However, SteMs do not have a secondary data structure exploiting the temporal information of stream elements. The timestamp information of elements is solely used for duplicate avoidance in SteMs. We tailor a SweepArea to the semantic requirements of an operator by specifying order relations and predicates for probing and eviction. In contrast, when routing through SteMs, the Eddy has to ensure correct query answers. While *Eddies* and SteMs work well for conjunctive queries, their extension to arbitrary query plans that may contain grouping or difference is non-trivial. Furthermore, the overhead of *Eddies* incurred by the additional flexibility is significant [Des04].

The declarative query language of *TelegraphCQ* is *StreaQuel*, a stream-only query language. Compared with our language, *StreaQuel* provides more sophisticated windowing capabilities. A for-loop construct at the end of a query specification declares the window transition behavior based on a variable  $t$  that iterates over time. The for-loop contains *WindowIs* statements defining the left and right end of the window as a function over  $t$  for each stream in the query. According to [ABW06], a stream-only query language derived from CQL would be quite similar to the purely stream-based approach implemented in *TelegraphCQ*.

### 16.4.3 Aurora

In *Aurora* [CCC<sup>+</sup>02, ACC<sup>+</sup>03], users construct query plans via a graphical interface. An *Aurora* query plan consists of boxes, which correspond to operators, and arrows that

connect the boxes indicating the dataflow. Stream elements are tuples consisting of application-specific data fields and a system timestamp set upon entry to the Aurora network. If operators generate a new tuple, it is tagged with the timestamp of the oldest tuple contributing to it. The Aurora query algebra, named SQuAl, supports seven operators. Similar to our approach, operators are parameterized with predicates and functions. This operator set is Turing complete. Order-agnostic operators are filter, map, and union. The filter differs from our filter operator because it evaluates a query predicate for any registered output stream. Similar to a case statement, the filter routes elements satisfying a predicate to the respective output stream. The map is a generalized projection operator that invokes a function on each attribute of a tuple. The union simply merges multiple streams disregarding the timestamp order. Order-sensitive operators are join, aggregation, BSort, and resample. These operators require an explicit order specification, which includes the ordered attribute, a slack parameter, and grouping attributes. It specifies the maximum disorder tolerated in a stream. An operator discards all tuples from its input stream that arrive more than slack elements out of order. Besides the order specifications, the join takes a size  $s$  and a join predicate as arguments and computes a binary band join [SH98], i.e., two tuples join if the query predicate holds and their timestamps are at most  $s$  units apart. The aggregation slides a window of fixed size over the input stream. The advance of the window is controlled by an additional parameter. An aggregate is the result of a user-defined function evaluated over all elements in the window. The user-defined function is composed of three functions to initialize the state, update the state, and to convert the state to a final result. We also use three functions to incrementally compute aggregates in analogy to [MFHH02] and [LWZ04]. Unlike Aurora, our algebra does not support BSort and resample. BSort is an approximate sort operator using a buffer of size  $slack + 1$ . Whenever the buffer is full, the element with the minimum value according to the order attribute is removed and appended to the output stream. Such an operator would not fit into our algebra as its output stream might violate our ordering requirement and thus correctness of downstream operators. The resample operator generates an interpolated value for every tuple in the first input stream by applying an interpolation function to a window of tuples from the second input stream. Similar to the join, resample restricts the elements in the window to be at most  $s$  time units apart from the element in the first input stream. It is somewhat difficult to compare the query semantics of Aurora with our semantics. Because stream elements are tagged with system timestamps, the semantics depends on element arrival and scheduling. Furthermore, the majority of operators are not snapshot-reducible. As a consequence, query optimizations are nearly impossible. Our work has opposed objectives, namely (i) the definition of an unambiguous query semantics to establish a foundation for query optimization and (ii) the development of a physical operator algebra that produces query answers invariant under scheduling and allowed optimizations. Aurora mainly aims at satisfying user-defined QoS specifications. *Borealis* [AAB<sup>+</sup>05] extends the core functionality of Aurora towards distributed stream query processing. An interesting feature of Borealis is the ability to process revision tuples to correct query results.

#### 16.4.4 Gigascope

*Gigascope* [CJSS03] is a high performance stream database for network monitoring applications with an own SQL-style query language termed GSQL, being mostly a restriction of SQL. Gigascope makes use of timestamps and sequence numbers by defining ordered attributes. GSQL supports selection, join, aggregation, and stream merge. Join and aggregation are nonblocking because the operators exploit properties inferred from the ordered attributes of their input streams. These properties are comparable with our stream ordering requirement and expiration patterns. Contrary to our work, Gigascope does not provide separate window operators. Instead, windows are strictly tied to the operators. The join predicate must evaluate a constraint defining a window on ordered attributes from both streams, and the aggregation should have at least one ordered grouping attribute. The merge operator corresponds to our union because it merges streams according to a common ordered attribute. Because any primary operation in GSQL is expressible in CQL [ABW06], it can also be expressed in our language.

#### 16.4.5 StreamMill

The *StreamMill* system [BTW<sup>+</sup>06] effectively applies user-defined aggregate functions on a wide range of applications including data stream mining [LTWZ05], streaming XML processing [ZTZ06], and RFID event processing. Queries are expressed through ESL, the *Expressive Stream Language*, which extends the current SQL:2003 standards. ESL supports the incremental computation of UDAs as proposed in ATLaS [WZL03] and provides additional support for window aggregates. Arbitrary UDAs can be defined by three different states of computation: INITIALIZE, ITERATE, and TERMINATE. This approach resembles the incremental aggregation in Aurora, [MFHH02], and the one in our physical algebra. However, all these approaches require a procedural language to implement the computation, whereas the computation can be specified natively in ESL. ESL supports physical and logical windows, whose semantics corresponds to our count-based and time-based windows respectively. A further similarity between ESL and our query language is the ability to express window slides. While ESL directly extends the built-in OLAP functions of SQL:2003 for windowed aggregates, we had to perform additional modifications to the BNF grammar so that windows are expressible over any stream referenced in the FROM clause. The evaluation of UDAs, in particular the delta-computation, exploits a novel EXPIRE construct to optimize the processing speed and memory utilization of window aggregates with slides and tumbles.

#### 16.4.6 Nile

The data stream management system *Nile* [HMA<sup>+</sup>04] provides two pipelined scheduling approaches for the efficient execution of continuous time-based sliding window queries, namely, the *Time Probing* and the *Negative Tuple* approach [GHM<sup>+</sup>07]. The first approach uses window-based operators to implement a query plan, e. g., the window join [HAE03]. The second approach uses a special operator, called *W-Expire*, to control the validity of stream elements by sending positive and negative elements according to the window

specification. The basic implementation concept derives from the maintenance of materialized views [GM95] and is also used in STREAM [ABW06]. We implemented the latter approach for our experimental comparison (see PNA in Chapter 14). Unfortunately, [GHM<sup>+</sup>07] outlines the basic approaches but does not present algorithms revealing the subtleties of more complex operators like grouping with aggregation or difference. Instead, they suggest two optimizations for alleviating the drawback of PNA, i. e., the doubled number of stream elements flowing through the query pipeline. The *time-message* optimization aims to avoid the processing of negative elements without affecting the output delay. The *piggybacking* optimization tries to reduce the number of negative elements in the query pipeline by merging multiple negative elements and/or time-messages into a single time-message. From the system perspective, Nile runs each operator in a separate thread and implements operator communication via FIFO queues.

The predicate-window query model introduced in [GAE06] is a generalization over the sliding window query model. Rather than considering the most recent elements of an input stream in the window, the predicate-window includes those elements from the input stream that satisfy a given predicate. Based on an explicit correlation attribute, the window operator determines whether an incoming stream element qualifying the window predicate enters the window content for the first time or represents an update to an element already stored in the window. In the first case, the incoming element is added to the operator state and a positive element is emitted. In the second case, the window operator replaces the existing element with the incoming one and outputs an update element. Otherwise, an incoming element does not qualify the window predicate. In that case, the window operator checks if it contains an element that has the same correlation attribute value as the incoming element. If so, the window operator evicts this element from the state and produces a negative element as output. In future work, we plan to adopt predicate-windows for the time-interval approach by implementing the corresponding window operator. Furthermore, it is desirable to study its expressive power.

#### 16.4.7 Tapestry

The *Tapestry* system [TGNO92] was the first that defined a semantics for continuous queries over an append-only database. In contrast to active databases [SPAM91], Tapestry does not make use of trigger mechanisms. Instead, queries are rewritten as monotonic queries and then converted into incremental queries that are evaluated periodically. Due to the problem of blocking operators, TQL, the language of Tapestry, is restricted to a subset of SQL. Law et al. identified in their later work [LWZ04] that the class of queries expressible by nonblocking operators corresponds to the class of monotonic queries. Remarkably, the continuous semantics defined for Tapestry already ensures snapshot-reducibility for sets. The continuous query answer corresponds to the set union over the results of the one-time queries executed on the snapshots of the database at every instant in time. However, Tapestry does not support any kind of window queries.

#### 16.4.8 Tribeca

The stream database manager *Tribeca* [Sul96, SH98] introduces fixed and moving window queries over network packet streams. The querying capabilities are limited because Tribeca supports only unary stream-to-stream operators, most importantly windowed aggregation. Tapestry and Tribeca are strictly less expressive than CQL [ABW06].

#### 16.4.9 Cape

*Cape* [RDS<sup>+</sup>04] is an adaptive processing engine for continuous queries. CAPE offers adaptation techniques for tuning different levels of query evaluation. At intra-operator level, reactive operators exploit punctuations to reduce resource usage and processing costs [DMRH04]. At plan level, CAPE applies dynamic plan migration to reoptimize subplans at runtime [ZRH04]. At system-wide level, adaptive scheduling techniques and plan distribution among multiple machines further improve system scalability [SLJR05]. In terms of query semantics, it is difficult to compare our approach with CAPE because CAPE provides query answers on a best effort strategy with the goal to meet the QoS requirements of users.

#### 16.4.10 Cayuga

The general-purpose event monitoring system *Cayuga* [DGP<sup>+</sup>07] extends traditional content-based publish-subscribe systems to handle stateful subscriptions. This enables users to formulate subscriptions correlating multiple events. The query language of Cayuga relies on a formal algebra [DGH<sup>+</sup>06]. Like our physical streams, an event stream represents a potentially infinite set of event elements, each composed of a tuple and two timestamps indicating start and end time. The operator algebra supports projection, selection, renaming, union, conditional sequence, iteration, and aggregation. The binary conditional sequence operator computes sequences of two consecutive and non-overlapping events, filtering out those events from the second input that do not satisfy a given selection predicate. This operator allows grouping to be expressed. The iteration operator acts as a fixed point operator and can be viewed as a repeated application of conditional sequencing. The implementation of Cayuga differs totally from our approach as it is based on nondeterministic finite state automata (NFA). Cayuga is designed to efficiently support multi-query optimization by merging equivalent states occurring in several automata. A comparison with STREAM demonstrates the superiority of Cayuga for queries using the iteration operator, which can be emulated with self-joins in CQL. However, theoretical results on the expressive power of Cayuga in comparison to stream processing systems are not available yet.

#### 16.4.11 XML Stream Processing

##### Niagara, NiagaraCQ, and OpenCQ

*Niagara* [NDM<sup>+</sup>01] aims to answer queries through crawling and monitoring of XML documents distributed across a wide-area network. Niagara uses its own XML-based

query language that facilitates the specification of joins over XML documents and the construction of complex results. The subsystem *NiagaraCQ* [CDTW00] employs techniques for grouping continuous queries with the aim to profit from subquery sharing. Within the same project, [STD<sup>+</sup>00] suggest a differential approach to overcome the problem of blocking operators in query plans by producing partial query results. Viglas and Naughton [VN02] propose a cost model based upon stream rates to enable query optimization over streaming data. Our work is a prerequisite for cost-based query optimization, as it defines a sound query semantics according to which the well-known transformation rules hold. *OpenCQ* [LPT99] pursues similar objectives as Niagara. The system continually monitors the arrival of crawled information and pushes it to relevant users. Unlike pure event-condition-action models, OpenCQ employs incremental query processing combined with a rich event model supporting time-based and content-based trigger conditions.

### **XFiler and Xyleme**

*XFilter* [AF00] and *Xyleme* [NACP01] are content-based filtering systems for XML documents. XFilter relies on a publish-subscribe architecture that disseminates information according to user profiles. XPath is used as profile language. Xyleme monitors changes in HTML and XML documents by utilizing a trigger-based evaluation of continuous queries. A common feature of both systems is that their continuous query semantics derives from the periodic execution of one-time queries, similar to Tapestry.

### **StreamGlobe**

The *StreamGlobe* infrastructure [KSKR05] distributes data stream processing based on peer-to-peer networking techniques and grid technologies. StreamGlobe processes XML data streams, primarily in the field of e-science applications, and employs XQuery to formulate subscriptions in the form of continuous queries. Through the dissemination of data streams, the system can gain from data stream sharing [KK06a, KK06b]. In addition, pushing operators into the network allows efficient in-network query processing. The event-based query engine FluX [KSSS04] executes query processing tasks and performs optimizations based on the schema information of data streams.

### **OSIRIS-SE**

The stream-enabled hyperdatabase infrastructure *OSIRIS-SE* [BSS05] aims to achieve a high degree of reliability required for telemonitoring applications in health care. It combines data stream management with distributed process management and provides advanced failure detection and handling mechanisms. Operator state backups and operator migration are salient features ensuring reliable query execution.

## 16.5 Further Related Approaches

In a broader context, our approach is also related to the great deal of work on main-memory databases [GS92] and real-time databases [KG95, ÖS95]. Moreover, there has been extensive research on approximate query processing, e.g., [ZS02, DGR03, GKMS01, DGM05, TCZ<sup>+</sup>03, AN04, ANWS06]. As our approach provides exact query answers for the specified windows rather than approximate ones, these approaches are altogether complementary. Nonetheless, a sampling operator can be incorporated into our algebra by using a filter along with a predicate implementing a probabilistic strategy, e.g., coin flipping. Unfortunately, sampling-based approaches often cannot give reliable guarantees on the accuracy of query answers [CM99]. As a consequence, query optimization becomes unfeasible [CMN99]. Nevertheless, high-quality approximate answers are acceptable when the system is at risk to saturate. In the following, we briefly summarize further research directions connected to our approach.

### 16.5.1 Sequence Databases

[SLR94, SLR95, SLR96] propose the design and implementation of a database system with support for sequence data. A sequence represents a many-to-many mapping from positions to records. A sequence query is an acyclic graph of sequence operators that can be either positional or record-oriented operators, assuming the full sequence is stored. The sequence operator algebra allows for the correlation of records across sequences in a flexible manner. Based on the scope of an operator, various query optimization techniques are proposed. The moving window aggregates introduced in [SLR96] apply aggregate functions with methods INITIALIZE(), ACCUMULATE(RECORD), and TERMINATE() for incremental computation. Aggregation in stream processing, e.g., in [MFHH02, ACC<sup>+</sup>03, LWZ04], resembles the latter principle. Unlike sequence databases, the language semantics in DSMSs refers to continuous – not to one-time – queries.

### 16.5.2 Materialized Views

A materialized view [GM95] can be considered as a stored continuous query which is updated whenever its base relations are modified. The *chronicle data model* [JMS95] provides operators over relations and chronicles. Like a raw stream, a chronicle represents an append-only ordered sequence of tuples. [JMS95] focuses on the complexity of incremental maintenance of materialized views over chronicles. Our notion of continuous queries differs from materialized views since query results are streamed rather than stored. Furthermore, continuous queries necessitate the use of adaptive query processing strategies to meet the challenging requirements of volatile application environments [BB05].

### 16.5.3 Active Databases

[PD99] gives a survey on active database systems. The majority of these systems support event-condition-action (ECA) rules. The work closest to ours is the *Alert* architecture [SPAM91] that allows the specification of ECA-style triggers on top of a

traditional (passive) DBMS to incorporate functionality of an active DBMS. Based on append-only tables, an extended cursor behavior models continuous queries, because tuples added to the table after the cursor opened also contribute to the query results. In general, rule-based approaches incur higher runtime overhead compared to SQL-based execution plans because matching incoming data to rules needs to be performed at runtime, whereas SQL-based plans can be optimized algebraically prior to their execution.

#### 16.5.4 Sensor Databases and Networks

Query processing in sensor networks makes allowances for the limited power and computational resources of sensing devices [BGS01, MF02, YG02, MFHH05]. This is in contrast to our work that relies on an abstraction from the type of data sources providing the data streams.

##### Cougar

The object-relational *Cougar* system [BGS01, YG02] optimizes query plans for in-network query processing. By pushing operators like selection and aggregation into the network, it trades costly communication for cheap local computation with the aim to extend the lifetime of a sensor network. The data model and long-running query semantics builds on the results of [SLR95]. Conceptually, a sensor database mixes stored data and sensor data. A continuous query represents a materialized view that is maintained incrementally during a given time interval. Treating such a view as a virtual relation enables the user to specify joins over sensor data and persistent data. The SQL-like query language supports signal-processing functions modeled as ADT functions to return sensor data. An every construct in the WHERE clause specifies the reevaluation frequency of a continuous query.

##### TinyDB

The query processor for sensor networks *TinyDB* [MFHH05] incorporates acquisitional techniques to minimize power consumption. The framework addresses issues of when, where, and how often data should be sampled and which data should be delivered in distributed, embedded sensor environments. The TinyDB query language extends the SELECT-FROM-WHERE-GROUPBY clause in SQL with novel syntax to define sample intervals, storage points saving intermediate results, and temporal aggregates. Furthermore, the language supports events as a mechanism for initiating data collection besides the continuous polling-based approaches. It is also possible to define the lifetime of continuous queries in advance. The latter constitutes a promising extension to our query language.

##### Global Sensor Networks

The Java middleware *GSN* (*Global Sensor Networks*) [AHS06] is a generic software platform enabling the flexible integration and deployment of heterogeneous sensor networks. Similar to our PIPES infrastructure, GSN consists of highly modular components modeling

the essential requirements and concepts of an application domain. Such an approach minimizes coding effort and also code replication since the generic components can easily be customized to the particular requirements of a novel application. The key to abstraction are virtual sensors which correspond either to real-world sensors or to query results. XML-based deployment descriptors specify query and virtual sensor properties. Interestingly, the actual query is embedded as a standard SQL query, while the deployment descriptors add time-related constructs such as windows.



# 17 Extensions

This chapter presents several enhancements of our approach. Section 17.1 extends our window operators to give them control over the window advance. Section 17.2 presents how the CQL-specific operators `Istream`, `Dstream`, and `Rstream` can be implemented with our time-interval approach. Support for these operators enables the use of CQL as query language on top of our operator algebra. Section 17.3 sketches a hybrid approach to query execution by combining PNA and TIA. Finally, Section 17.4 reveals how we incorporate relations into continuous query processing.

## 17.1 Window Advance

The stream query repository [SQR03] gives several examples of continuous sliding window queries that not only specify the window size but also the granularity at which the window slides. According to the examples, window specifications in CQL can optionally contain a `SLIDE` expression. Time-based sliding windows take a timespan as slide parameter, whereas count-based and partitioned windows take a positive integer instead. Because this language feature is not discussed in [ABW06] but supported in our query language (see Chapter 7), we want to clarify its semantic and implementational subtleties.<sup>1</sup> Throughout this chapter, we use the terms *window slide* and *window advance* interchangeably.

**Example 17.1. Highest Bid Query** – Every 10 minutes return the highest bid(s) in the most recent 10 minute period.

```
SELECT itemID, bid_price
  FROM Bid WINDOW(RANGE 10 MINUTES
                  SLIDE 10 MINUTES)
 WHERE bid_price = (SELECT MAX(bid_price)
                      FROM Bid WINDOW(RANGE 10 MINUTES
                                      SLIDE 10 MINUTES));
```

The example query contains a time-based sliding window of size 10 minutes that slides at a 10-minute granularity. The window does not slide for each element of the bid stream, but only once every 10 minutes. Let us assume the finest time granularity are seconds, then a chronon corresponds to a second. The window slides from minutes [0:00, 10:00) to [10:00, 20:00), then to [20:00, 30:00), and so on. The content of the window changes at the latest chronon in each 10 minute window. For the first window, this chronon [9:59, 10:00) matches with the first time instant at which a result is possible since the window

---

<sup>1</sup>Nonetheless, [Ara06] addresses at least the semantics of the CQL `SLIDE` expressions.

contains the stream elements of the past 10 minutes. The results remain unchanged until the chronon prior to minute 20 is reached, because then the window slides the next time. This particular type of window, in which the window size is equal to the slide granularity, is called a *tumbling window* [LMT<sup>+</sup>05, BTW<sup>+</sup>06].

### 17.1.1 Time-based Sliding Window

Let  $a \in T, a > 0$ , be the progression step at which the window advances. The window starts at the earliest time instant 0 and moves forward only once every  $a$  time units by an amount of  $a$  time units. The window content changes at time instants  $a - 1, 2 \cdot a - 1, 3 \cdot a - 1$ , and so on. We designate the start timestamp of such a particular chronon as *window start*. Relative to the window start, a window captures  $w$  time units of the past (including the window start). Up to this chapter, we used the default window advance  $a = 1$  when specifying windows because we omitted the optional SLIDE clause. The default window advance corresponds to the finest granularity of the time domain.

#### Semantics

At a time instant  $\hat{t}$  the output stream captures all tuples of the logical input stream  $S^l$  with a timestamp in the interval  $\left[\max\left\{\left\lfloor \frac{\hat{t}+1}{a} \right\rfloor \cdot a - w, 0\right\}, \left\lfloor \frac{\hat{t}+1}{a} \right\rfloor \cdot a - 1\right]$ . The time instant  $\left\lfloor \frac{\hat{t}+1}{a} \right\rfloor \cdot a - 1$  is the window start. It corresponds to the most recent time instant less than or equal to  $\hat{t}$  at which the window content changes. A tuple appears in the output at time instant  $\hat{t}$  if it appeared in the input within the most recent  $w$  time units prior to the window start.

$$\begin{aligned} \omega_{w,a}^{\text{time}}(S^l) := & \left\{ (e, \hat{t}, \hat{n}) \mid \hat{t} \geq a - 1 \wedge \exists X \subseteq S^l. X \neq \emptyset \wedge \right. \\ & X = \left\{ (e, t, n) \in S^l \mid \max\left\{\left\lfloor \frac{\hat{t}+1}{a} \right\rfloor \cdot a - w, 0\right\} \leq t \leq \left\lfloor \frac{\hat{t}+1}{a} \right\rfloor \cdot a - 1 \right\} \\ & \left. \wedge \hat{n} = \sum_{(e,t,n) \in X} n \right\}. \end{aligned} \quad (17.1)$$

The window advance  $a$  is an additional parameter of the window operator, expressed as subscript.

#### Implementation

The implementation of the time-based sliding window with control over the window advance can be done efficiently by rounding the start timestamp of the incoming element up to the next time instant at which the window content will change. The end timestamp is set relative to the new start timestamp but rounded down by considering that an element can at most be valid for  $w + a$  chronons while the window slides only once every  $a$  chronons. See Algorithm 18 for details. The rounding does not conflict with the required output stream order. Whenever multiple start timestamps of tuples in the input stream intersect with the same slide interval, those are rounded up to the starting instant of the

**Algorithm 18:** Time-based Sliding Window with Advance Parameter ( $\omega_{w,a}^{\text{time}}$ )

**Input** : chronon stream  $S_{in}$ ; window size  $w$ ; window advance  $a$   
**Output** : physical stream  $S_{out}$

```

1  $S_{out} \leftarrow \emptyset;$ 
2  $\tilde{t}_S, \tilde{t}_E \in T;$ 
3 foreach  $s := (e, [t_S, t_S + 1)) \leftarrow S_{in}$  do
4    $\tilde{t}_S \leftarrow \left\lceil \frac{t_S+1}{a} \right\rceil \cdot a - 1;$ 
5    $\tilde{t}_E \leftarrow \left\lfloor \frac{t_S+w+a}{a} \right\rfloor \cdot a - 1;$ 
6   if  $\tilde{t}_E \neq \tilde{t}_S$  then
7      $(e, [\tilde{t}_S, \tilde{t}_E)) \leftarrow S_{out};$ 

```

$$\omega_{50,10}^{\text{time}}(S_3)$$

Tuple	$t_S$	$t_E$
$b$	9	59
$a$	9	59
$c$	9	59
$a$	9	59
$b$	19	69

Table 17.1: Time-based window over  $S_3$  with window size 50 and advance 10 time units

succeeding slide interval. Hence, the start timestamps of the corresponding tuples in the output stream will be identical. The **if**-statement checks that time intervals do not collapse to empty intervals, a case possible for  $a > w$ .

**Example 17.2.** Table 17.1 shows the effects of the advance parameter. We used the same input stream and window size as in the example for the physical time-based window (see page 98), but adjusted the window advance to 10 time units. The slide intervals start at time instants 9, 19, 29, and so on. At the chronons associated with these instants the window slides. For this reason, the start timestamps between  $[0, 10)$  are rounded up to 9. The start timestamps between  $[10, 20)$  are rounded up to 19, and so on. Because  $a < w$  and  $w$  is a multiple of  $a$ , the end timestamps are set relative to the new start timestamps with an offset of  $w$  time units.

### 17.1.2 Count-based Sliding Window

Let  $A \in \mathbb{N}, A > 0$ . When no window advance is specified for a count-based window,  $A = 1$  by default. However, count-based windows can also have greater progression steps. According to our query language, the expression `WINDOW(ROWS N SLIDE A)` following a stream reference defines a count-based window of size  $N$  with advance  $A$  elements over the chronon stream. The window slides once every  $A$  elements. Let  $s_1, s_2, s_3, \dots$  denote the elements of the input stream. Then, the first *slide interval* contains the elements  $s_1$  to  $s_A$ , the second slide interval captures  $s_{A+1}$  up to  $s_{2A}$ , and so on. As a consequence, the window start is always an element with a sequence number  $i \in \mathbb{N}, i > 0$ , that is a multiple

of the window advance  $A$ . As the window has to contain the most recent  $N$  elements, the window end is the element with sequence number  $i - N + 1$ .

### Semantics

The following definition relies on the assumption that the logical input stream must not contain multiple elements being valid at the same time instant. As already mentioned in Section 8.2.2, this assumption ensures that count-based windows have an unambiguous semantics.

$$\begin{aligned} \omega_{N,A}^{\text{count}}(S^l) := & \left\{ (e, \hat{t}, \hat{n}) \mid n(\hat{t}) \geq A \wedge \exists X \subseteq S^l. X \neq \emptyset \wedge \right. \\ & X = \left\{ (e, t, n) \in S^l \mid \max\left\{ \left\lfloor \frac{n(\hat{t})}{A} \right\rfloor \cdot A - N + 1, 1 \right\} \leq n(t) \leq \left\lfloor \frac{n(\hat{t})}{A} \right\rfloor \cdot A \right\} \\ & \left. \wedge \hat{n} = \sum_{(e,t,n) \in X} n \right\} \end{aligned} \quad (17.2)$$

where  $n(t)$  is the stream size at time instant  $t$  defined by

$$n(t) = |\{(e, \tilde{t}, n) \in S^l \mid \tilde{t} \leq t\}|.$$

### Implementation

According to the previous restriction, the count-based window generates deterministic and thus well-defined results if the physical input stream is a chronon stream without duplicate start timestamps.

---

**Algorithm 19:** Count-based Sliding Window with Advance Parameter ( $\omega_{N,A}^{\text{count}}$ )

---

**Input** : chronon stream  $S_{in}$ ; window size  $N$ ; window advance  $A$

**Output** : physical stream  $S_{out}$

```

1  $S_{out} \leftarrow \emptyset;$ 
2 Let  $SA$  be the empty SweepArea( $\leq_{t_S}$ ,  $-$ ,  $p_{remove}$ );
3 Let  $Q$  be an empty FIFO queue;
4 foreach  $s := (e, [t_S, t_S + 1]) \leftarrow S_{in}$  do
5   if  $Q.\text{size}() < A - 1$  then  $Q.\text{enqueue}(s)$ ;
6   else
7     while  $\neg Q.\text{isEmpty}()$  do
8       Element  $(\tilde{e}, [\tilde{t}_S, \tilde{t}_S + 1]) \leftarrow Q.\text{dequeue}();$ 
9       UPDATESWEEPAREA( $(\tilde{e}, [\tilde{t}_S, \tilde{t}_S + 1])$ ,  $SA$ ,  $S_{out}$ );
10      UPDATESWEEPAREA( $s$ ,  $SA$ ,  $S_{out}$ );
11 TERMINATE( $SA$ ,  $S_{out}$ );

```

---

Algorithm 19 extends Algorithm 13 by a FIFO Queue  $Q$ . The SweepArea implementation and parameters are the same. For  $A > 1$ , incoming elements are first inserted into queue

---

**Procedure** UPDATESWEEPAREA(*element s, SweepArea SA, stream S<sub>out</sub>*)

---

```

Iterator results  $\leftarrow$  SA.extractElements(s, 1);
if results.hasNext() then
    Element  $(\hat{e}, [\hat{t}_S, \hat{t}_S + 1]) \leftarrow \text{results.next}()$ ;
    if  $\hat{t}_S \neq t_S$  then
         $(\hat{e}, [\hat{t}_S, t_S]) \hookrightarrow S_{out}$ ;
    SA.insert(s);
```

---

**Procedure** TERMINATE(*SweepArea SA, output stream S<sub>out</sub>*)

---

Lines 9 to 12 of Algorithm 13;

---

*Q*. Whenever queue *Q* reaches a size of  $A - 1$  elements, the sequence number of the incoming element is a multiple of  $A$ . This means, the incoming element triggers the advance of the window. In this case, all elements are extracted from queue *Q*. For each extracted element  $(\tilde{e}, [\tilde{t}_S, \tilde{t}_S + 1])$ , a new element is created from tuple  $\tilde{e}$  and the time interval of the incoming element (see line 9). Afterwards, the new element is handed over to the auxiliary procedure UPDATESWEEPAREA, which performs the further processing. Procedure UPDATESWEEPAREA enhances lines 4 to 8 of Algorithm 13 with an additional check for  $\hat{t}_S \neq t_S$  to filter out empty intervals that might appear in the case  $A > N$ . After queue *Q* has been emptied, the incoming element is processed (see line 10). The auxiliary procedure TERMINATE is called after the last element of the input stream has been processed. Since the window will not slide any more, it is sufficient to append the elements of the SweepArea to the output stream. Because the queue and the SweepArea process elements in a FIFO fashion, the output stream order is preserved.

$$\omega_{2,2}^{\text{count}}(S_3)$$

Tuple	t <sub>S</sub>	t <sub>E</sub>
<i>b</i>	3	7
<i>a</i>	3	7
<i>c</i>	7	$\infty$
<i>a</i>	7	$\infty$

Table 17.2: Count-based window with a size and advance of 2 elements over  $S_3$

**Example 17.3.** Table 17.2 shows the results from sliding a count-based window of size 2 over input stream  $S_3$  (see page 98). The window slides every 2 elements. It first contains  $(b, [1, 2])$  and  $(a, [3, 4])$  from the input stream and then shifts to  $(c, [4, 5])$  and  $(a, [7, 8])$ . The validity of tuples captured by the window over the input stream is adjusted so that elements in the output stream correctly reflect the desired window semantics. Due to the finite input stream, the window does not slide again. This means the output stream does not contain an element for  $(b, [10, 11])$ , which is stored in the queue *Q* but not in the SweepArea. The remaining elements in the SweepArea are appended to the output stream with  $\infty$  as end timestamp.

### 17.1.3 Partitioned Window

The definition of the partitioned window results from substituting the count-based window for the count-based window with control over the window advance in Formula (8.19). From the implementation side, Algorithm 14 can be modified similar to Algorithm 19 by combining the SweepArea of each group with a FIFO queue of size  $A - 1$ .

## 17.2 CQL Support

Our physical algebra can easily be used to implement arbitrary CQL queries [ABW06] because it provides stream-to-stream variants for every windowing construct in CQL. Furthermore, we have a stream-to-stream analog for every relation-to-relation operator in CQL. The `Rstream` operator corresponds to a split operator with output interval length  $l_{out} = 1$ . As already mentioned in Chapter 15, `Istream` and `Dstream` are derived operators. In our logical algebra, they can be expressed as follows:

$$\text{Istream}(S^l) := S^l \triangleright_=(\omega_2^{\text{time}}(S^l) - S^l) \quad (17.3)$$

and

$$\text{Dstream}(S^l) := (\omega_2^{\text{time}}(S^l) - S^l) \triangleright_+ S^l. \quad (17.4)$$

The snapshot  $\tau_t(\omega_2^{\text{time}}(S^l) - S^l)$  at time instant  $t$  contains all elements of the logical stream  $S^l$  that were valid at time instant  $t - 1$ . Due to the anti-equi-join,  $\tau_t(S^l \triangleright_=(\omega_2^{\text{time}}(S^l) - S^l))$  contains all elements of the logical stream  $S^l$  that do not match with an element in  $\tau_t(\omega_2^{\text{time}}(S^l) - S^l)$ . Hence, `Istream` computes  $\tau_t(S^l) - \tau_{t-1}(S^l)$ . As `Dstream` addresses deletions, it computes  $\tau_{t-1}(S^l) - \tau_t(S^l)$ .

To facilitate the replacement of stream-to-relation operators in our physical algebra, we present an efficient implementation of the `Istream` and `Dstream` operator for the time-interval approach. Both operators take a physical stream as input and generate a *chronon* stream as output. Furthermore, both operators implicitly perform a maximum coalesce on the input stream. `Istream` reduces the time intervals from the coalesced input stream to the chronons capturing the first timestamp the associated tuple was valid, which corresponds to  $[t_S, t_S + 1]$ . Contrary to that, `Dstream` shrinks the time intervals to the chronons that cover the last time instant the associated tuple was valid, i. e.,  $[t_E - 1, t_E]$ .

### 17.2.1 Istream

The `Istream` operator for the time-interval approach is shown in Algorithm 22.

#### SweepArea Implementation and Parameters

The SweepArea is a hash table. Elements inside the hash buckets are linked in ascending order by end timestamps. The order relation  $\leq_{t_E}$  enforces a secondary linkage of all elements in the SweepArea according to their end timestamps across bucket boundaries. The query predicate is identical to that of the coalesce. Viewed from a top-level perspective,

**Algorithm 22:** Istream

---

**Input** : physical stream  $S_{in}$   
**Output** : chronon stream  $S_{out}$

```

1  $S_{out} \leftarrow \emptyset;$ 
2 Let  $SA$  be the empty SweepArea( $\leq_{t_E}$ ,  $p_{query}$ ,  $p_{remove}$ );
3 foreach  $s := (e, [t_S, t_E]) \leftarrow S_{in}$  do
4   Iterator  $qualifies \leftarrow SA.\text{query}(s, 1);$ 
5   if  $\neg qualifies.\text{hasNext}()$  then
6      $(e, [t_S, t_S + 1)) \leftarrow S_{out};$ 
7   else
8      $qualifies.\text{next}();$ 
9      $qualifies.\text{remove}();$ 
10   $SA.\text{insert}(s);$ 
11   $SA.\text{purgeElements}(s, 1);$ 

```

---

`Istream` inherently performs a maximum coalesce. Therefore, the remove predicate merely contains the first condition of that specified for the coalesce operator.

$$p_{remove}(s, \hat{s}) := \begin{cases} \text{true} & \text{if } \hat{t}_E < t_S, \\ \text{false} & \text{otherwise.} \end{cases}$$

Istream( $S_2$ )

Tuple	ts	te
b	1	2
d	3	4
a	4	5
e	10	11

Table 17.3: Istream over physical stream  $S_2$ 

**Example 17.4.** Table 17.3 demonstrates the output of `Istream` over stream  $S_2$  (see page 73). Element  $(b, [7, 15))$  is dropped due to coalescing. The output is a chronon stream where the chronons match with the start timestamps of the coalesced input stream.

### 17.2.2 Dstream

Algorithm 23 presents the `Dstream` operator.

#### SweepArea Implementation and Parameters

The `Dstream` utilizes the same implementation as `Istream`. In contrast to `Istream`, where incoming elements are modified and directly appended to the output stream, the method `extractElements` provides the results. Due to the order relation  $\leq_{t_E}$ , the output stream is

**Algorithm 23:** Dstream

---

**Input** : physical stream  $S_{in}$   
**Output** : chronon stream  $S_{out}$

```

1  $S_{out} \leftarrow \emptyset;$ 
2 Let SA be the empty SweepArea( $\leq_{t_E}$ ,  $p_{query}$ ,  $p_{remove}$ );
3 foreach  $s := (e, [t_S, t_E]) \hookleftarrow S_{in}$  do
4   Iterator  $results \leftarrow SA.extractElements(s, 1);$ 
5   while  $results.hasNext()$  do
6     Element  $(\hat{e}, [\hat{t}_S, \hat{t}_E]) \leftarrow results.next();$ 
7      $(\hat{e}, [\hat{t}_E - 1, \hat{t}_E]) \hookleftarrow S_{out};$ 
8   Iterator  $qualifies \leftarrow SA.query(s, 1);$ 
9   if  $qualifies.hasNext()$  then
10     $qualifies.next();$ 
11     $qualifies.remove();$ 
12   $SA.insert(s);$ 
13 Iterator  $results \leftarrow SA.iterator();$ 
14 while  $results.hasNext()$  do
15   Element  $(\hat{e}, [\hat{t}_S, \hat{t}_E]) \leftarrow results.next();$ 
16    $(\hat{e}, [\hat{t}_E - 1, \hat{t}_E]) \hookleftarrow S_{out};$ 

```

---

ordered properly because the generated time intervals capture solely the last time instant a tuple is valid.

*Dstream( $S_2$ )*

Tuple	$t_S$	$t_E$
$a$	4	5
$d$	8	9
$b$	14	15
$e$	17	18

Table 17.4: Dstream over physical stream  $S_2$ 

**Example 17.5.** Table 17.4 illustrates the output of `Dstream` for stream  $S_2$  (see page 73). For each tuple, `Dstream` preserves the final time instant at which the tuple is valid. This includes an implicit coalescing. In our example, element  $(b, [1, 7])$  is dropped due to coalescing. Because start timestamps of the output stream are set to the end timestamps of the input stream minus one time unit, `Dstream` has to reorganize elements for satisfying the ordering requirement of the output stream, which is accomplished by the SweepArea.

### 17.3 Converters

In addition to the operator implementations for TIA and PNA, PIPES provides converters to change the query processing methodology from TIA to PNA and vice versa. These

converters are specific stateful physical operators that take a physical stream as input and generate a stream of positive and negative elements as output, or vice versa. We already indicated this translation process in Section 14.1. By installing these converters into a physical plan it is possible to partition the plan into disjoint subplans that are either processed using TIA or PNA. Hence, the converters enable hybrid query processing with the aim to benefit from the merits of both approaches.

## 17.4 Relations

Many real-world applications require a mixture of continuous queries over volatile data streams and persistent relations [SQR03, TTPM02]. Therefore, it is crucial to clarify how relations can be incorporated into our stream processing approach. The different techniques depend on the type of relations. We distinguish between the following types:

- A *time-varying relation* defines an unordered, finite multiset of tuples at any time instant  $t \in T$ . This definition corresponds to the one used in [ABW03, ABW06].
- A *temporal relation* defines an unordered, finite multiset of tuples having two specific attributes for denoting the time period start and end respectively. Temporal relations are common in temporal databases [SJS01].
- A *constant relation* is an unmodifiable relation in the standard relational model with no notion of time. A constant relation can be viewed as a temporal relation whose tuples are tagged with the interval  $[0, \infty)$ .

**Example 17.6.** Let us consider the relations in the NEXMark benchmark (see Figure 4). Examples for time-varying relations are `Person` and `Item`. The attribute `registrationTime` allows well-defined instantaneous relations to be obtained. An instantaneous relation refers to a certain time instant and captures all tuples in the relation being valid at this instant.

The relation `Category` is an example for a constant relation. It does not provide any temporal information. The content is assumed to be constant during query processing, i. e., all possible categories have to be registered in advance.

**Remark 17.1.** One might argue why we confine our approach to constant relations instead of traditional, non-temporal relations. The reason is that query semantics would otherwise depend on transaction time. Suppose a binary operator over a stream and a traditional relation. Whenever a continuous query accesses the relation on arrival of a new stream element, it operates on the current state of that relation. The continuous query is not aware of any modifications performed on that relation. As a consequence, the results depend on database transactions, i. e., the points in time when data facts get current in the relation and may be retrieved. In traditional database systems, serialization of transactions guarantees a sound semantics. However, a continuous query would hold a read-lock on the referenced relation unless the query is stopped. This argument reveals why we implicitly assume the underlying relation to be constant when trying to achieve semantic compliance with traditional database systems.

We support two different techniques for enabling continuous queries over streams and relations: (1) the *conversion of relations into streams*, and (2) the *provision of specific operators*.

### 17.4.1 Conversion into Streams

This technique applies to time-varying relations. To transform a time-varying relation into a raw stream, we adopt the relation-to-stream operators defined in [ABW06]. `Istream` applied to a relation  $R$  streams the insertions to the relation, whereas `Dstream` streams the deletions. `Rstream` applied to a relation  $R$  generates a stream element  $(e, t)$  whenever tuple  $e$  is in  $R$  at instant  $t$ . Since the definition of a stream in [ABW06] matches with our notion of a raw stream, those operators can be applied directly to an updatable relation for generating a raw stream, i. e., a sequence of tuple-timestamp pairs. The corresponding functionality can be implemented using triggers.

To give a user the opportunity to specify continuous queries over streams and time-varying relations in our query language, we enhanced the `FROM` clause. A reference to a time-varying relation  $R$  must have the following format:

`Istream(R) | Dstream(R) | Rstream(R).`

Be aware that the usage of the keywords `Istream`, `Dstream`, and `Rstream` differs from CQL where these constructs appear in the `SELECT` clause.

**Example 17.7. Monitor New Users Query** – Find people who put something up for sale within 12 hours of registering to use the auction service.

```
CREATE STREAM NewPersonStream AS
    SELECT P.id, P.name
    FROM   Istream(Person) P;

    SELECT DISTINCT(P.id, P.name)
    FROM   SELECT P.id, P.name
           FROM   NewPersonStream WINDOW(RANGE 12 HOURS) P, OpenAuction O
           WHERE  P.id = O.sellerID;
```

The first statement creates a new stream that contains the newly registered people along with their registration timestamp. We convert the time-varying relation `Person` into a stream by means of the `Istream` construct. Hence, a stream element  $(e, t)$  is generated whenever a tuple  $e$  representing information about a new person is added to the relation `Person` at instant  $t$ , where  $t$  corresponds to the registration time.

The second statement defines a time-based sliding window over the derived stream `NewPersonStream` and computes the join with the `OpenAuction` stream to retrieve all people that opened an auction within the last 12 hours. The duplicate elimination (`DISTINCT`) causes people that initiated multiple auctions during the specified period to be listed only once.

### 17.4.2 Provision of Specific Operators

This technique applies to temporal and constant relations. Often continuous queries make use of binary operators where one of the inputs is a stream whereas the other input

is a relation. Instead of a conversion, we provide specific implementations of the join and difference that reduce those operators to a unary stream-to-stream variant. Despite the single input stream, an operator still uses two SweepAreas, one for the input stream and one for the relation. The contents of the relation are reflected by the SweepArea which is usually implemented as an index over the relation. For instance, the symmetric stream join (see Algorithm 6) turns into an index join probing the incoming stream elements against an index over the relation. Our algorithms have to be modified insofar as time progression – usually monitored by variable  $min_{t_s}$  – is solely determined by the input stream. Furthermore, elements must not be deleted from the SweepArea associated with the input relation.

### 17.4.3 Output Relations

Some applications might require the results of a continuous query to be materialized [GPÖ06]. To cover that case we propose the following solutions. First, a sliding window over a physical stream can be stored as a temporal relation. For example, a temporal relation might store all stream elements of the past four weeks. Providing such a functionality in a DSMS enables historical queries [ACC<sup>+</sup>03]. Second, we can use our TIA-to-PNA converter on top of a physical query plan to transform a physical stream into a stream of positive and negative elements. The corresponding insertions and deletions of such a stream can be used to maintain a materialized view [GM95].



## 18 Conclusions

Manifold approaches to stream processing have been published in recent years. Despite this surge, the development of a sound semantic foundation for a general-purpose stream algebra has attracted only little research attention. While Part II addressed this challenge, it provided even more because it gave seamless insight into the query processing engine, illustrating the coherence between the individual query processing steps ranging from query formulation to query execution. Hence, Part II has fostered the fusion between theoretical results and practical considerations in the streaming context. Our logical algebra precisely defines the semantics of each operation in a direct way over temporal multisets. We are convinced that the logical algebra is extremely valuable as it separates semantics from implementation. By assigning an exact meaning to any query at any point in time, our logical algebra describes what results to expect from a continuous query, independent of how the system operates internally. Moreover, it represents a tool for exploring and proving equivalences. Our stream-oriented physical algebra relies on a novel stream representation that tags tuples with time intervals to indicate tuple validity. It provides push-based, nonblocking algorithms that harness sweepline-techniques and input stream order for the efficient eviction of elements expiring in the operator state due to windowing constructs. From a theoretical point of view, Part II carried over the well-known transformation rules from conventional and temporal databases to stream processing, based on the concept of snapshot-equivalence. We enhanced this large set of rules with novel ones for the window operators. Furthermore, we defined equivalences for streams and query plans, pointed out plan generation as well as algebraic and physical optimizations. We proved that our query language has at least the same expressive power as CQL, while our language stays closer to SQL:2003 standards. From the implementation side, our unique time-interval approach is superior to the semantically equivalent positive-negative approach for the majority of time-based sliding window queries because it does not suffer from doubling stream rates to signal tuple expiration. In summary, Part II established a semantically sound and powerful foundation for data stream management by gracefully fusing and enriching research from various database areas.



## **Part III**

# **Cost-Based Resource Management**



## 19 Introduction

Data stream management systems have emerged as a new technology to meet the challenging requirements for processing and querying data that arrives continuously in the form of potentially infinite streams from autonomous data sources. Standard database technology is not well suited for coping with data streams, and particularly, with the many long-running queries that have to be processed concurrently. Not only the long-running queries but also the fluctuating stream characteristics require a high degree of adaptivity in a DSMS. This involves the dynamic re-optimization and (re-)distribution of system resources, like memory, at runtime.

This part deals with a novel approach to adaptive resource management for continuous sliding window queries. Our resource manager applies the following two techniques, with the aim to dynamically control the system resources allocated by a query plan: (i) *adjustments to window sizes* and (ii) *adjustments to time granularities*. The degree of adjustments is restricted by user-defined quality of service constraints. Both techniques do not conflict with query re-optimizations at runtime [ZRH04, KYC<sup>+</sup>06] as query answers are still exact for the QoS setting chosen by the resource manager. Note that this is not the case for sampling-based adaptation techniques. For random load shedding [TCZ<sup>+</sup>03], which reduces system load by dropping elements randomly, such strong guarantees on the query results cannot be given. In the best case, the query result is a sample of the exact query result having specific qualities. Even in this case, query optimization becomes quite limited as, for example, [CMN99] identified the inability to commute sampling with join.

Let us illustrate our techniques and their new types of QoS constraints by considering the following real-world application scenario where data streams are obtained from sensors of an industrial production line [CHK<sup>+</sup>06]. Therefore, imagine sensors that measure the speed of the production line at different positions. In order to run the production line properly, the speed information delivered by two successive sensors is only allowed to differ by a constant factor within a given sliding time window. Such a scenario can be computed by applying a time-based sliding window join on both streams. In general, we observed that users tend to be too conservative when defining window sizes, i. e., they choose relatively long time windows. Hence, a quite natural approach for a QoS constraint would be to allow the user to specify an upper and a lower bound for the window size. By adjusting the window size within these bounds, the DSMS can increase or decrease its resource usage at runtime. Even for small windows and high system load, this technique preserves those results computed from elements that are temporally close to each other, which is just the basic idea of sliding windows [GÖ03b]. In contrast, load shedding [TCZ<sup>+</sup>03] is likely to drop some of these results. The second QoS constraint refers to the time granularity at which results, especially aggregates, are computed. For a lot of real-world queries, the user neither needs nor wants results at finest time granularity. In our example, it would suffice to provide the average speed of

the production line with a time granularity of seconds rather than milliseconds.

A prerequisite for the effective use of our adaptation techniques is an accurate *cost model* which allows the efficiency of a (re-optimized) query plan to be evaluated by estimating the required amount of resources. Although this part validates our cost model in the context of adaptive resource management, the model is general enough to serve as a foundation for cost-based query optimization as well, because it relies on fundamental parameters including stream rates, window sizes, and time granularities. The main contributions of this part are summarized as follows:

- We introduce two novel techniques for adaptive resource management in a DSMS, namely adjustments to window sizes and time granularities, along with their QoS constraints. We explain their semantics, suggest syntactical extensions for our query language, and outline their effects on query optimization.
- We develop an appropriate cost model for estimating the resource utilization of continuous sliding window queries. This model is more detailed and extensive than existing ones as it covers a variety of operators even aggregation, considers time granularity, and includes reorganization costs caused by the temporal expiration of elements in the operator state due to windowing constructs.
- We use this cost model to quantify the effects of changes to window sizes and time granularities on resource utilization. Furthermore, we analyze at which point in time these effects become fully observable.
- We sketch our adaptive resource management architecture based on our cost model, adaptation techniques, QoS constraints, and a suitable adaptation strategy.
- Thorough experimental studies on synthetic and real-world data streams (i) show that adjustments to window sizes and time granularities are suitable for adaptive resource management, (ii) validate the accuracy of our cost model, (iii) illustrate the differences between window reduction and load shedding, and (iv) prove the scalability of both, the techniques and our cost model.

The remainder of this part is organized as follows. Chapter 20 briefly outlines the underlying semantics and functionality of our novel granularity conversion operator. The basic techniques of our approach to adaptive resource management are presented in Chapter 21. Chapter 22 introduces our cost model for estimating the resource usage of a query plan. The use of this cost model with regard to adaptive resource management is addressed in Chapter 23. Chapter 24 summarizes our experimental studies. Related work is discussed in Chapter 25. Finally, Chapter 26 concludes this part.

# 20 Granularity Conversion

Because our second adaptation technique relies on the time granularity associated with a data stream, Section 20.1 formally introduces our notion of *time granularity* first. Section 20.2 proposes our query language extensions that allow the user to express the desired time granularity. After that, Sections 20.3 and 20.4 enrich our logical and physical algebra with the *granularity conversion* operator respectively. Section 20.5 describes the placement of the granularity conversion within a query plan. Finally, Sections 20.6 and 20.7 describe the impact of granularity conversions on logical and physical query plans.

## 20.1 Time Granularity

A *time granularity*  $\mathbb{G} = (G, \leq_{|G})$  is a non-empty subdomain of the time domain  $\mathbb{T}$ ,  $G \subseteq T, 0 \in G$ , with the same total order  $\leq_{|G}$ . We require the time distance  $g \in T, g > 0$ , between each pair of successive time instants in  $G$  to be equal.<sup>1</sup> This condition holds for the conventional time granularities such as milliseconds, seconds, minutes, and hours. We denote a time granularity  $\mathbb{G}'$  to be coarser than  $\mathbb{G}$  if  $\mathbb{G}' \subset G$ . This is identical to saying time granularity  $\mathbb{G}$  is finer than  $\mathbb{G}'$ . The time domain  $\mathbb{T}$  represents the finest available time granularity, for which the time distance  $g$  corresponds to the duration of a *chronon*. Any coarser time granularity groups consecutive chronons into larger segments, termed *granules* [JCE<sup>+</sup>94]. The first segment starts at instant 0, the second segment at instant  $g$ , and so on. Every segment contains  $g$  chronons. Therefore, we designate our time distance parameter  $g$  as *granule size*. Note that our notion of time granularity is actually the *timestamp granularity* because it specifies the resolution of timestamps.

**Definition 20.1** (Stream Granularity). Let  $\mathbb{G} = (G, \leq_{|G})$  be a time granularity. A *logical stream*  $S^l$  has time granularity  $\mathbb{G}$  if for all elements  $(e, t, n) \in S^l : t \in G$ . A *physical stream*  $S^p$  has time granularity  $\mathbb{G}$  if for all elements  $(e, [t_S, t_E])$  from  $S^p : t_S \in G \wedge t_E \in G$ .

## 20.2 Query Language Extensions for Time Granularity Support

We enable the user to formulate the desired time granularity for a stream by enhancing the FROM clause with an optional *granularity specification*. The granularity specification consists of the keyword GRANULARITY followed by a value and time unit specification in brackets. The granularity specification is appended to the window specification.

---

<sup>1</sup>Similar to the specification of the window size for the time-based sliding window (see Section 8.2.1), we interpret the notation  $g \in T$  as the period of (application) time spanned by the time interval  $[0, g]$ .

**Example 20.1.** Let us assume a raw data stream obtained from a sensor in a production line. Let the time(stamp) granularity, i. e., the resolution at which stream elements are recorded, be milliseconds.

```
SELECT AVG(temperature)
  FROM sensor WINDOW(RANGE 1 HOUR) GRANULARITY(1 MINUTE);
```

This query continuously calculates the average temperature within the most recent hour over the stream *sensor* with a time granularity of one minute. As a consequence, the timestamps associated to the aggregate values in the resultant stream refer to the beginning of a minute rather than to that of a millisecond. The query answer is accurate, which means exact, for any snapshot drawn at a time instant representing the start of a minute.

### 20.3 Semantics

The logical *granularity conversion*  $\Gamma : \mathbb{S}_{\mathcal{T}}^l \times T \rightarrow \mathbb{S}_{\mathcal{T}}^l$  takes a logical stream and a *granule size*  $g \in T$ ,  $g > 0$ , as input and produces a logical stream having the associated time granularity as output. The associated time granularity  $\mathbb{G} = (G, \leq_{|G})$  consists of all time instants  $t \in T$  that are multiples of  $g$ , i. e.,  $G := \{t \in T \mid \exists i \in \mathbb{N}_0. t = i \cdot g\}$ . The time granularity of the input stream has to be finer than the time granularity of the output stream.

An element of the logical input stream appears in the output stream if it is valid at a time instant  $t$  that belongs to the target granularity  $\mathbb{G}$  defined by the given granule size  $g$ . In our notation, the argument for the granule size is expressed as subscript.

$$\Gamma_g(S^l) := \{(e, t, n) \in S^l \mid t \in G\} \quad (20.1)$$

$\Gamma_2(S_1)$	
T	Multiset
1	$\langle \rangle$
2	$\langle a, a, a \rangle$
3	$\langle \rangle$
4	$\langle a, a, a, b, c \rangle$
5	$\langle \rangle$
6	$\langle b, b \rangle$

Table 20.1: Granularity conversion with granule size 2 over logical stream  $S_1$

**Example 20.2.** Table 20.1 displays the output stream of the granularity conversion applied to stream  $S_1$  with granule size  $g = 2$  (see Table 8.1 on page 41 for the definition of logical stream  $S_1$ ). The granule size implies that  $G := \{0, 2, 4, 6, \dots\}$ . Only at these time instants, the granularity conversion preserves the elements of the input stream in the output stream.

## 20.4 Implementation

Algorithm 24 shows the physical granularity conversion. Let  $g$  be a granule size greater than that of the input stream, i.e., the output stream will have a coarser granularity than the input stream. The granularity conversion is a stateless operator that rounds the start

**Algorithm 24:** Granularity Conversion ( $\Gamma_g$ )

---

<b>Input</b>	: physical stream $S_{in}$ ; granule size $g$
<b>Output</b>	: physical stream $S_{out}$

```

1  $S_{out} \leftarrow \emptyset;$ 
2  $\tilde{t}_S, \tilde{t}_E \in T;$ 
3 foreach  $s := (e, [t_S, t_E]) \leftarrow S_{in}$  do
4    $\tilde{t}_S \leftarrow \left\lceil \frac{t_S}{g} \right\rceil \cdot g;$ 
5    $\tilde{t}_E \leftarrow \left\lceil \frac{t_E}{g} \right\rceil \cdot g;$ 
6   if  $\tilde{t}_E \neq \tilde{t}_S$  then
7      $(e, [\tilde{t}_S, \tilde{t}_E]) \leftarrow S_{out};$ 

```

---

and end timestamp of an incoming element  $(e, [t_S, t_E])$  according to the granule size  $g$  given as argument. As a result, the new start and end timestamp  $\tilde{t}_S$  and  $\tilde{t}_E$ , respectively, will belong to the target granularity  $G$ . An element  $(e, [t_S, t_E])$  from the input stream is preserved in the output stream if  $[t_S, t_E] \cap G \neq \emptyset$ . This condition holds if  $\tilde{t}_E \neq \tilde{t}_S$  because start and end timestamps are rounded up. If  $\tilde{t}_E = \tilde{t}_S$ , the interval  $[\tilde{t}_S, \tilde{t}_E]$  did not intersect with a time instant in  $G$  and, consequently, no result is produced.

$\Gamma_2(S_1)$

Tuple	$t_S$	$t_E$
$c$	2	8
$a$	6	12
$d$	6	14
$b$	12	18

Table 20.2: Granularity conversion with granule size 2 over physical stream  $S_1$

**Example 20.3.** Table 20.2 demonstrates the output stream of the granularity conversion with granule size  $g = 2$  over physical stream  $S_1$  (see Table 11.1 on page 73). The element  $(a, [9, 10])$  from the input stream is dropped because the rounding generated the empty time interval  $[10, 10]$ .

## 20.5 Plan Generation

Whenever the user expresses a time granularity for a stream in the query statement, the query translator places the corresponding granularity conversion operator downstream of

the window operator in the *logical query plan*. This placement is always possible because any granularity specification requires a window specification. Recall that in the case of now-windows the window specification is implicit and can be omitted. As now-windows can also be omitted in plans, the granularity conversion may occur without the upstream window. During *physical plan generation*, the logical granularity conversion is replaced with its physical counterpart.

Basically the granularity conversion can be placed at any position in the corresponding query plan. If granularity conversions are placed close to the sources, more downstream operators can benefit from reduced costs due to the coarsened time granularity. However, downstream of the granularity conversion the query results are only exact for the new, coarser time granularity. When considering multiple continuous queries with granularity specifications and possibly common subexpressions, the placement of granularity conversions opens up new optimization opportunities with the objective to exploit subquery sharing.

To ensure sound and consistent query semantics, the time granularity is only allowed to become coarser downstream. Unfortunately, some operator combinations do not necessarily guarantee this property. Let us point out these combinations and suggest solutions:

1. In the case of nested queries, a time granularity violation occurs if a granularity conversion downstream has a finer granule size than a granularity conversion upstream. We overcome this problem by defining the constraint that the granule size downstream can only be set coarser.
2. If the window size of a time-based window downstream of a granularity conversion is not a multiple of the granule size, end timestamps might be set to instants that do not belong to the correct time granularity. We suggest that the user either has to specify the window size in dependency of the granule size, i. e., as a multiple of it, or a separate granularity conversion is applied to the output stream of the window operator to correct the time granularity.
3. The split operator can also produce time intervals with start and end timestamps that might conflict with the time granularity of the input stream. This is the case if the output interval length is chosen independently from the granule size. We solve this problem as done for the time-based window, i. e., we either limit the choice of the output interval length to multiples of the granule size or place a separate granularity conversion downstream of the split operator.

For the latter two cases, it would also be possible to enhance the respective algorithms with an additional parameter for the granule size and perform the timestamp rounding to meet the granularity requirements inside the operators.

## 20.6 Semantic Impact

Let us clarify how the granularity conversion complies with our notion of *snapshot-equivalence* introduced in Chapter 10.

**Lemma 20.1.** *Given two logical query plans  $Q_1$  and  $Q_2$  computing the same continuous query over the same inputs, but with different time granularities. Let  $S_1^l$  and  $S_2^l$  be the output streams with time granularities  $G_1$  and  $G_2$  respectively. Without loss of generality, let  $G_1$  be finer than  $G_2$ . It follows that the snapshots of both output streams are equal at time instants belonging to the coarser granularity  $G_2$ :*

$$\forall t \in G_2. \tau_t(S_2^l) = \tau_t(S_1^l). \quad (20.2)$$

*Proof.* Both plans consist of the same operators except for the granularity conversions. Therefore, they would produce snapshot-equivalent results without the granularity conversions (see Chapter 10). According to the definition of the logical granularity conversion, all results being valid at a time instant in  $G_2$  are preserved in  $S_2^l$  and all results being valid at a time instant in  $G_1$  are preserved in  $S_1^l$ . The condition that time granularity  $G_1$  is finer than time granularity  $G_2$  implies that  $G_2 \subset G_1$ . As a consequence, the snapshots of  $S_1^l$  and  $S_2^l$  are equal at any time instant  $t \in G_2$ .  $\square$

As we require the time granularity to become only coarser downstream, we conclude from Lemma 20.1 that granularity conversions preserve snapshot-equivalence at the coarsest time granularity in a query plan. Hence, the powerful set of transformation rules holding in our stream algebra remains applicable (see Section 10.2). If query optimizations are applied to a plan containing granularity conversions, the query answer is still exact for the coarsest time granularity.

## 20.7 Physical Impact

We have introduced the time granularity conversion because of its effect on the resource consumption of stateful operators. A coarser stream granularity leads to a reduced number of time instants at which the validity of a tuple can change because only instants of the coarser time granularity are regarded. The physical granularity operator achieves this goal by adjusting the start and end timestamps of time intervals to time instants of the target granularity. The coarser the output stream granularity is, the higher is the probability that multiple start and end timestamps, respectively, are rounded to identical time instants of the coarser granularity. As the validity of tuples can only start and expire at instants of the coarser granularity, stateful operators downstream of the granularity conversion benefit from reduced memory consumption and processing costs due to the propagation of less frequent changes.

The described effects become particularly clear for the *scalar aggregation* (see Algorithm 9). If a granularity conversion coarsens the granularity of the aggregation's input stream, the aggregate value changes less frequently over time. As a result, the output stream rate decreases. Furthermore, the state contains fewer partial aggregates than it would contain for a fine input stream granularity. Recall that partial aggregates are split whenever the start or end timestamp of an incoming stream element falls into the time interval attached to the partial aggregate (see Figure 11.3). A coarser granularity causes less splits and, thus, leads to a smaller state. Moreover, processing costs are reduced due to fewer aggregate computations. *Grouping with aggregation* also profits from coarsening the time granularity of the input stream for the same reasons.

Other operators are less affected by changes to the time granularity of their input streams. Among this category of operators are the *stateless* operators and the remaining *stateful* operators. They mainly profit from a coarser time granularity whenever an upstream granularity conversion drops elements, i. e., if it reduces stream rates. Often, however, the average validity of tuples within a physical stream is considerably larger than the granule size. In this case, the rounding performed by the granularity conversion has only marginal effects on the operator state because the validity of tuples remains roughly the same.

Note that a granularity conversion influences the entire downstream plan, not only the next downstream operator. Moreover, the effects accumulate. For instance, a coarser granularity shrinks the costs for the scalar aggregation as well as the output rate of the aggregation. The reduced stream rate, in turn, diminishes the costs of further downstream operators and so on. These indirect effects are important to the global resource consumption of query plans. They demonstrate that any physical operator can be affected by granularity conversions, independent of how sensitive it is concerning the time granularity, because the processing costs of any operator depend on its input stream rates.

# 21 Adaptation Approach

We propose the following two techniques for controlling resource usage: (i) *adjustments to window sizes* and (ii) *adjustments to time granularities*. First, Section 21.1 gives an overview of our resource management architecture. Thereafter, Section 21.2 introduces language extensions for specifying QoS bounds. Section 21.3 describes our adaptation techniques and their use. Further runtime aspects are discussed in Section 21.4. Eventually, Section 21.5 reveals why both adaptation techniques preserve our continuous query semantics. The latter property ensures that, even if query optimizations are performed at runtime, our resource management approach is still applicable and produces correct and meaningful results. This constitutes a major advantage over random load shedding [TCZ<sup>+</sup>03] where no strong guarantees on query results can be given if optimizations are performed.

## 21.1 Overview

The knowledge of our semantics and implementation concepts for continuous sliding window queries (see Part II) is a prerequisite for understanding the effects of our adaptation techniques and the development of our cost model. Since the resource usage of operators depends on their implementation, this part focuses on our physical operator algebra. Although we explain the adaptation techniques for our time-interval approach, notice that they can easily be transferred to related stream processing systems. We devoted a section in Chapter 25 to this aspect.

In accordance with [BSW04], our work assumes that resources spent on inter-operator queues are almost negligible for unsaturated systems as those are capable of keeping up with stream rates. Therefore, we focus on the operators inside a query plan and their resource allocation. Figure 21.1 gives an overview of our adaptive resource management architecture. The resource manager controls the resources allocated by the operators in the query graph. In order to adjust resource allocation, it applies our two adaptation techniques at runtime. The maximum extent to which the resource manager is allowed to change the window and granule size is determined by the user. For this purpose, we enriched our query language with additional syntax for specifying suitable *QoS bounds*. In order to make any change transparent to a user so that he/she can retrieve the QoS setting according to which a query result was computed, the resource manager maintains a *change history*.

## 21.2 Query Language Extensions for QoS constraints

Our idea is to give the user the chance to specify ranges for window and granule sizes. The query results are returned to the user according to the specified QoS bounds. The

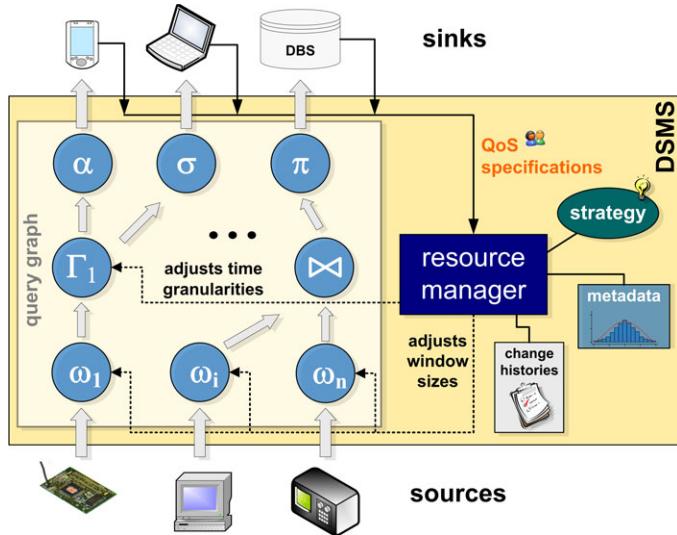


Figure 21.1: Adaptive resource management architecture

resource manager is only allowed to adjust the window size and the time granularity within these pre-defined bounds in order to adapt resource usage. However, it should aim at maximizing overall QoS.

**Example 21.1.** Let us explain our language extensions with the help of the following query.

```
SELECT AVG(temperature)
  FROM sensor WINDOW(RANGE MIN 12 HOURS, MAX 24 HOURS)
        GRANULARITY(COARSEST 2 HOURS, FINEST 10 SECONDS);
```

We have extended the window specification to accept two arguments introduced by the novel keywords **MIN** and **MAX** following the window's frame unit specification (see Section 7.2). The example shows a time-based window whose size can vary between 12 and 24 hours. We have enhanced the granularity specification in the same way by defining the two new keywords **COARSEST** and **FINEST**. In the above example, the stream *sensor* should have a time granularity of at least two hours and at most 10 seconds. The setting of the granularity affects the quality of the query answer, here, the average temperature computation over a time-based window. A fine granularity leads to frequent aggregate updates and gives the user a high temporal resolution of aggregate values, whereas a coarse granularity reduces the number of updates at the expense of the temporal resolution.

Although we focus on time-based sliding windows in this part, adjustments to window sizes are also applicable to count-based and partitioned windows. The extended window clause of a count-based window could look as follows: **WINDOW(ROWS MIN 50, MAX 200)**.

## 21.3 Adaptation Techniques

Let us now introduce our adaption techniques in more detail. Given a plan for a time-based sliding window query that contains possibly multiple windows and granularity conversions.

### 21.3.1 Adjusting the Window Size

#### Technique

Adjusting the size of a time-based window means changing it to a new value at a certain application time instant. The value is determined by the strategy of the resource manager and chosen in between the QoS bounds specified for this window.

#### Implementation

Whenever the resource manager wants to change the size of the time-based window at runtime, it informs the corresponding physical operator about the new window size and a future point in time  $t \in T$  at which the window size should be changed. Recall that  $t$  generally refers to application time, not system time. As soon as the start timestamps of the window operator's input stream are equal to or greater than  $t$ , the window operator substitutes the current window size for the new window size. As a result, from instant  $t$  onwards, it tags tuples with time intervals having the length of the new window size (see Section 11.6.1).

#### Impact

For time-based sliding window queries it is apparent that there is a direct correlation between the window sizes specified in a query and the memory usage of stateful operator in the corresponding physical query plan. Let us explain this correlation for our time-interval approach. We consider a stream element as relevant to the operator state as long as its time interval may overlap with that of future incoming elements. Hence, the time interval attached to a tuple indicates the validity. Based on temporal expiration, unnecessary elements are purged from the state. Because stateful operators have to keep all valid elements in their state, the state becomes larger the longer the validity is, and smaller the shorter the validity is. As a consequence, smaller windows imply that elements can be removed from the state earlier and, therefore, the state gets smaller. Furthermore, changes to the window size typically affect CPU load, because the smaller the state the faster becomes element retrieval and eviction. In the case of a nested-loops join, for example, halving a state would speed up probing by a factor of two.

### 21.3.2 Adjusting the Time Granularity

#### Technique

Adjusting the time granularity for a stream means changing the granule size of the corresponding granularity conversion operator at an explicit point in application time

$t$ . Let  $G$  be the current time granularity of this operator and  $\hat{G}$  be the new granularity. Both time granularities have to be within the user-defined QoS bounds. Whether  $\hat{G}$  is chosen coarser or finer than  $G$  depends on the current resource usage and the strategy of the resource manager.

In contrast to Definition 20.1 which associates a single time granularity to a stream, changes to the time granularity partition a stream into disjoint, successive *substreams* having different granularities. Changing the time granularity at instant  $t$  starts a new substream having granularity  $\hat{G}$ .

### Implementation

The granularity conversion adjusts incoming elements to the current granularity  $G$  as long as their start timestamp is less than  $t$ . Thereafter the new granularity  $\hat{G}$  is applied. Changes from a finer to a coarser granularity are always possible in Algorithm 24 by simply replacing the granule size. However, the inverse direction might violate the ordering requirement of the output stream as elements are directly appended to the output stream after rounding. Under the assumption that the time granularity is only allowed to be changed from a coarser to a finer granularity at time instants belonging to the coarser granularity, rounding the interval starts and endings does not violate the temporal order of the output stream. This restriction limits the feasibility only marginally, but facilitates the implementation towards a mapping on the time intervals as performed in Algorithm 24.

### Impact

Increasing the granule size affects any downstream operator as described in Section 20.7, particularly scalar aggregation and grouping with aggregation. The effects include savings in memory allocation and computational costs but also reductions to stream rates. In contrast, decreasing the granule size inverts the effects.

## 21.4 Runtime Considerations

In the following, we address some runtime aspects of our adaptation approach.

### 21.4.1 Granularity Synchronization

As already mentioned in Section 20.6, we require the granularity to become only coarser downstream. This means, that multiple granularity conversions on a path need to be synchronized so that the resource manager cannot set the granule size of an upstream granularity conversion greater than the granule size of any downstream granularity conversion. Therefore, when changing the granule size, the resource manager (i) has to ensure that the new granule size is at most equal to the smallest granule size of any downstream granularity conversion or (ii) it also has to adjust the granule sizes downstream to avoid inconsistencies.

### 21.4.2 QoS Propagation

Let us assume that continuous queries are expressed with our query language and its extensions for QoS constraints. As a DSMS runs a query graph representing a composition of possibly shared single queries, we have to clarify what happens to QoS constraints if subqueries are shared. When posing a new query, it is transformed into a physical query plan and integrated into the running query graph. The user-defined QoS constraints specified in the query statement are stored as metadata at the corresponding *sink*, i. e., the node in the query graph where the results for this query are transferred to (see Figure 21.1). Through a depth-first traversal starting at this sink, the QoS metadata is propagated upstream to update the existing QoS bounds at the window and granularity conversion operators.

Whenever a window operator with bounds  $[w_{min}, w_{max}]$  is reached for which QoS constraints  $\hat{w}_{min}$  and  $\hat{w}_{max}$  have been specified, its lower bound  $w_{min}$  is set to  $\max(w_{min}, \hat{w}_{min})$  and its upper bound  $w_{max}$  is set to  $\min(w_{max}, \hat{w}_{max})$ . Similarly, the set of allowed granularities  $\{g_{coarsest}, \dots, g_{finest}\}$  is updated whenever a granularity conversion is traversed for which QoS constraints with granularities  $\{\hat{g}_{coarsest}, \dots, \hat{g}_{finest}\}$  have been defined. We define the new granularity range as  $\{g_{coarsest}, \dots, g_{finest}\} \cap \{\hat{g}_{coarsest}, \dots, \hat{g}_{finest}\}$ . Notice that subquery sharing is only permitted if the computed QoS constraints for the shared segments are compatible, i. e., for all windows the lower bound has to be less than or at least equal to the upper bound and for all granularity conversions the sets of permitted granularities have to be non-empty. Otherwise, the new query has to be run in a separate part of the query graph.

### 21.4.3 Change Histories

The query answer provided to a sink in a query graph can depend on several windows and granularity conversions. In order to give the user information about the window sizes and time granularities actually used to compute the query results, every sink stores a list with references to the windows and granularity conversions it depends on. At each window operator in the query graph, a *history* of window sizes is maintained as a temporally ordered list of tuples  $(t, w)$ , where  $t$  denotes the time instant from which the window size  $w$  has been applied onwards. Analogously, each granularity conversion maintains a history of applied granularities.

Whenever the resource manager changes a time granularity or a window size, it updates the corresponding history by appending a new entry to the list. This operation has constant costs. However, the maintenance of the histories is of particular importance for giving strong guarantees on the query results. In order to keep a history small, all entries can be removed whose timestamp is less than the minimum start timestamp of the latest query result of all sinks referencing this history. Deletions can be performed with constant costs on average as each list (history) is implicitly ordered by time.

In order to determine the chosen QoS settings that contributed to the computation of a query result  $(e, [t_S, t_E])$ , the histories of the involved windows and granularity conversions are probed. Probing returns all history entries whose timestamp intersects with  $[t_S, t_E]$ . To achieve an efficient search, each sink keeps an additional reference to the history

entry with the smallest timestamp equal to or greater than the start timestamp of the predecessor of result ( $e, [t_S, t_E]$ ). Probing a history means starting a list traversal at this referenced entry and returning all entries with a timestamp  $t < t_E$ . Correctness results from the property that each physical stream is ordered by start timestamps.

## 21.5 Semantic Guarantees

*Snapshot-equivalence* is the key to understanding why our adaptation techniques are consistent with query optimization at runtime. Be aware that for a reduced window size and a coarsened time granularity, the query answer is still an exact subresult of the original query answer with full window size, but projected to snapshots of the coarser granularity. This exact subresult is invariant under any snapshot-equivalent query optimization (see also Section 20.6).

Since the resource manager is only allowed to change the QoS settings at runtime within the pre-defined QoS bounds, all query results have at least the minimum QoS demanded by the user. In general, the QoS picked out by the resource manager is somewhere in between the user-defined bounds. Due to the change history maintained at every window and granularity conversion, the system can determine the exact QoS setting at this operator for any time instant. Therefore, the system can provide the user not only with query results but also with the corresponding QoS settings that contributed to the computation of a result. As a consequence, all query results are precisely defined and exact with regard to the chosen window sizes and time granularities. According to our semantics and snapshot-reducible operators, this property is still valid under query re-optimizations at runtime due to the snapshot-equivalence of query plans (see Chapter 10). The interested reader is referred to [KYC<sup>+</sup>06, YKPS07] and Part IV for details on dynamic plan migration. Recall that all conventional transformation rules known from the extended relational algebra, e. g., join reordering or selection push-down, can be applied in our approach. On the contrary, sampling-based techniques such as random load shedding [TCZ<sup>+</sup>03] do not permit such powerful optimizations in general as shown for the join in [CMN99]. Notice that the join over samples of its input does not return a sample of the join's output.

# 22 Cost Model

This chapter presents our cost model for estimating the resource utilization of continuous queries. It is structured as follows. First, Section 22.1 substantiates the general purpose of our cost model. Section 22.2 defines our model parameters to describe a physical stream, called *stream characteristics* in the following. Based on the input stream characteristics of an individual operator, we estimate its output stream characteristics and resource consumption (see Figure 22.2). Section 22.3 shows our parameter estimation, while Section 22.4 presents estimations for the resource usage of operators. The resource utilization of an entire continuous query is finally obtained by summing up the resource consumption of the individual operators forming the query plan.

## 22.1 Purpose

A solid cost model is crucial to any DSMS for the following reasons:

- While measurements only give information about the presence and past, a cost model makes it possible to estimate resource usage in the future.
- Continuous monitoring of memory and processing costs at operator-, query-, and system-level may be expensive. Using a cost model instead saves computational costs as resource usage is re-estimated only when current stream characteristics differ significantly from past ones.
- Whenever a new query is posed, it allows the system to decide in advance whether sufficient resources are available to run the query or not.
- A cost model can also be used to estimate the impact of (re-)optimizations on query plans, which includes information about stream rates and resource usage. Thus, it is a prerequisite for cost-based query optimization and adaptive resource management.

## 22.2 Model Parameters

During our work we identified three parameters describing a physical stream as important to operator resource estimation. We model the *characteristics* of a physical stream  $S$  as a triple  $(d, l, g)$  with  $d, l \in \mathbb{R}_+, g \in T$ , where

- $d$  is the average time *distance* between the start timestamps of successive stream elements in  $S$ , i. e., the average inter-arrival time,

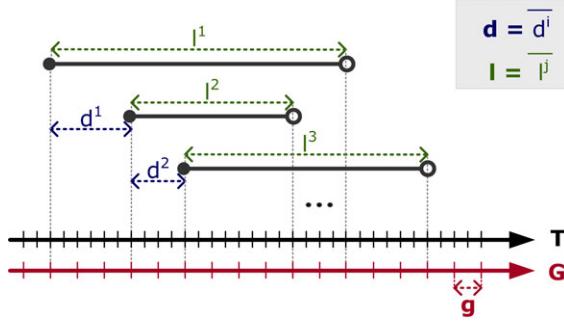


Figure 22.1: Model parameters

- $l$  is the average *length* of time intervals in  $S$ , i. e., the average validity of tuples, and
- $g$  is the constant time distance of the time *granularity* of  $S$ , i. e., the granule size.

Figure 22.1 demonstrates how these parameters are derived from a physical stream. Note that all three parameters refer to application time. PIPES stores those stream characteristics as metadata in its runtime environment so that they are accessible for the resource manager [CKS07].

### 22.2.1 Meaning

For a stateful operator, the knowledge of these three parameters for each input stream allows us to estimate the size of the status structures comprising the operator state. Informally,  $l$  describes the average time an element is relevant to the operator state and usually depends on the windows specified. Parameter  $d$  represents the average time progression within a stream. Thus,  $\frac{1}{d}$  corresponds to the application stream rate. Be aware that the application stream rate is independent of the internal stream rates of a DSMS which highly depend on scheduling, i. e., the number of elements transferred per unit in system time [VN02]. Our parameter  $d$  is not affected by scheduling at all. If  $d$  is known for each input stream of an operator, the average number of elements entering the operator state per application time unit can be computed. The additional information about  $l$  for each input stream makes it possible to determine the average number of elements being removed from the state per application time unit due to temporal expiration. Recall that the size of the operator state depends on the time granularity of the input streams as well, for instance, for the scalar aggregation. Therefore, our model incorporates the granule size  $g$  as an additional parameter.

### 22.2.2 Initialization

#### Parameter $d$

As our cost model works bottom up in a query plan starting at the sources, the model parameters of the raw streams have to be obtained first. Parameter  $d$  is either known in advance because it is given for a certain application or the DSMS can easily compute it by

an online average [HHW97]. For instance, if an autonomous data source such as a sensor produces a stream element every  $k$  time units,  $d$  would simply be  $k$ .

### Parameter $l$

Parameter  $l$  is set to the average length of time intervals in a stream. For physical base streams obtained from raw streams,  $l = 1$  as those streams are chronon streams. If the application itself tags tuples with time intervals instead of timestamps, parameter  $l$  either has to be set accordingly, e. g., to the size of an epoch [MFHH05], or has to be computed by an online average [HHW97]. Note that after a time-based sliding window operator,  $l$  is constant and corresponds to the window size.

### Parameter $g$

If the time granularity of a raw stream is not provided by the application, it is set to the finest time granularity available in the DSMS by default.

#### 22.2.3 Updates

Our definition of parameter  $d$  assumes stream rates to be steady on average. This is a common assumption for cost models [VN02, AN04], which for instance holds for Poisson-distributed arrival rates. Whereas this seems inadequate for many applications in the long term because stream rates can change over time, it is usually appropriate in the short term. In order to detect changes in parameter  $d$  for a raw stream, we suggest computing an online average over the start timestamp offsets with exponential smoothing as temporal weighting strategy [Gar85, Cha96]. Whenever the difference between the current online average and the parameter  $d$  used in the latest estimation exceeds a certain threshold, we propose to replace  $d$  with the new online average. Thereafter all dependent cost estimates need to be updated. The trade-off between estimate accuracy and computational update overhead can be controlled by the threshold parameter. Additional updates can be triggered periodically or by an application to further increase the degree of adaptivity of the DSMS.

#### 22.2.4 Justification for the Choice of Model Parameters

One might argue that our cost model is oversimplified because it only takes averages as parameters, namely the average start timestamp offset ( $d$ ) and interval length ( $l$ ) of a physical stream. Note that both parameters,  $d$  and  $l$ , correspond to the expectation values of the unknown distributions for start timestamp offsets and time interval lengths respectively. A more detailed and precise model would represent a data stream as a stochastic process to infer properties of these distributions. However, such a model would be extremely complex and expensive to compute, and thus unsuitable for adaptive resource management. Moreover, it is likely that in general those properties no longer hold for the output stream of an operator because each operator affects those distributions. In other words, the corresponding distributions for the output stream of an operator usually

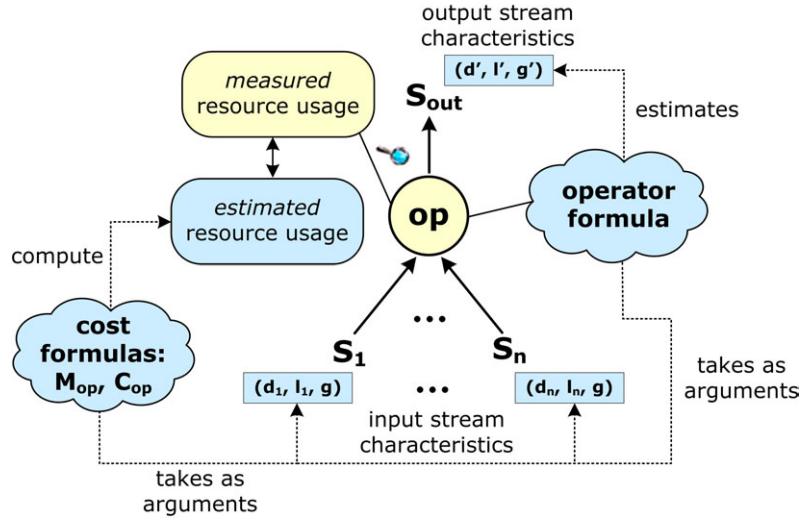


Figure 22.2: Use of stream characteristics

differ from the distributions for its input streams. For instance, consider the Cartesian product which computes the intersection of time intervals; it is hard to characterize these distributions. Hence, fairly strong assumptions have to be made to develop appropriate operator formulas, even for a complex model based on stochastic processes.

It is arguable whether or not such a model is better suited for cost estimation than ours, bearing in mind that the estimates may be more precise, but at the expense of significantly higher computational costs. Recall that one important objective for the development of our cost model is scalability. The more queries are executed in a system concurrently, the more operators are installed, and the more cost formulas have to be evaluated. Scalability necessitates an efficient cost estimation. Our formulas can be evaluated with constant costs per operator and thus cause only low computational overhead.

Although it would surely be possible to compute more accurate cost estimates based on distributions obtained from online density estimators, e.g., the ones proposed in [HS05], this approach does not scale as it is far too expensive to maintain a density estimator for every data stream in a DSMS. Instead, we suggest the use of density estimators over raw streams to initialize the expectation value  $d$ , and to trigger a bottom-up update of the cost estimates whenever the current expectation value differs significantly from the one used in the latest estimation.

## 22.3 Parameter Estimation

For each individual operator, our cost model provides an *operator formula* that estimates the output stream characteristics based on the input stream characteristics (see Figure 22.2). For a given query, we compute intermediate stream characteristics by starting at the source characteristics and applying our operator formulas bottom up in the plan. For the case of subquery sharing, i.e., when a new query is placed on top of already running

queries, we use the stream characteristics at the connection points to initialize the operator formulas of the shared parts. The stream characteristics of the non-shared parts are initialized as described above.

### 22.3.1 Preliminaries

Given potentially infinite input streams. The operator formula of a unary operator takes a triple  $(d, l, g)$  as input stream characteristics and produces a triple  $(d', l', g')$  as output stream characteristics. Let  $(d_i, l_i, g_i), i \in \{1, \dots, n\}$  denote the input stream characteristics for  $n$ -ary operators, as shown in Figure 22.2. For the special case that all input streams of an operator are empty, the operator has the empty stream as output by default – a case not explicitly handled in the following. Our operator formulas for estimating the output stream characteristics are based one the following assumptions:

1. In the case of multiple input streams, we require their time granularities to be equal. This can easily be achieved by applying a granularity conversion to all inputs, setting the granularity to the coarsest common subgranularity.
2. The estimation of our operator formulas assumes the time domain to be continuous. Despite this premise, our estimates turn out to be sufficiently accurate for our physical algebra based on the discrete time domain  $\mathbb{T}$ .
3. We assume no correlation between the tuple values appearing in a data stream and the timestamps. This is a common assumption for stream cost models [VN02, AN04]. The re-estimations performed for adaptation purposes, however, diminish any existing correlation effects so that our cost model still provides good estimates, a fact also confirmed by our experiments with real-world data streams (see Chapter 24).

### 22.3.2 Operator Formulas

The following formulas estimate the output stream characteristics for the operators of our physical algebra (see Chapter 11). The appended explanation substantiates our estimation.

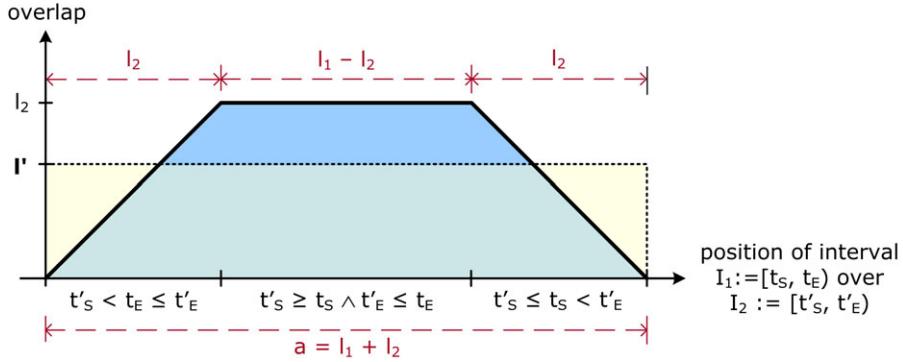
**Operator Formula 22.1** (Filter). Let  $sel \in \mathbb{R}$  be the selectivity of the filter predicate,  $0 < sel \leq 1$ . For the filter, we estimate

$$d' = \frac{1}{sel} \cdot d, l' = l, \text{ and } g' = g. \quad (22.1)$$

**Explanation.** As a stream element is dropped with a probability of  $sel$  on average, the average time distance between start timestamps of successive stream elements is increased to  $d' = \frac{1}{sel} \cdot d$ . The average length of time intervals remains unchanged under the premise that the filter predicate does not prefer any elements having a specific interval length.  $\square$

**Operator Formula 22.2** (Map). For the map, we estimate

$$d' = d, l' = l, \text{ and } g' = g. \quad (22.2)$$


 Figure 22.3: Estimation of  $l'$  for the Cartesian product

**Explanation.** Because the mapping function is applied to the tuple component of stream elements, the mapping has no effect on the attached time intervals. Thus, the parameters modeling the characteristics of the input stream remain the same for the output stream.  $\square$

**Operator Formula 22.3 (Union).** For the union of  $n$  input streams, we estimate

$$d' = \frac{1}{\sum_{i=1}^n \frac{1}{d_i}}, l' = \frac{\sum_{i=1}^n \frac{l_i}{d_i}}{\sum_{i=1}^n \frac{1}{d_i}}, \text{ and } g' = g. \quad (22.3)$$

**Explanation.** For each input stream, the average input rate  $r_i$  per time unit corresponds to  $\frac{1}{d_i}$ . As the union merges all input streams, the rate  $r'$  of the output stream is the sum of the input rates,  $r' = \sum_{i=1}^n r_i = \sum_{i=1}^n \frac{1}{d_i}$ . Due to the reciprocal relation between the average stream rate and the inter-arrival time, it follows that  $d' = \frac{1}{r'} = \frac{1}{\sum_{i=1}^n \frac{1}{d_i}}$ .

The average length of time intervals in the output stream  $l'$  is the weighted average over the  $l_i$ ,  $i \in \{1, \dots, n\}$ , where the weight is the stream rate  $r_i = \frac{1}{d_i}$ . Intuitively,  $r_i$  expresses how many time intervals with average length  $l_i$  start at each single time instant. Thus, the average output length  $l'$  results from the sum of the weighted  $l_i$  divided by the sum of the weights.  $\square$

**Operator Formula 22.4 (Cartesian Product).** For the Cartesian product of two streams (see Algorithm 6), we estimate

$$d' = \frac{d_1 \cdot d_2}{l_1 + l_2}, l' = \frac{l_1 \cdot l_2}{l_1 + l_2}, \text{ and } g' = g. \quad (22.4)$$

**Explanation.** For each input stream, the Cartesian product maintains a separate Sweep-Area that keeps all elements as long as they can have a temporal overlap with future incoming elements. Hence, the elements of input stream  $i$  have to be kept for  $l_i$  time instants on average. According to the input stream rates  $\frac{1}{d_i}$ , the average size of SweepArea  $i$  results from  $\frac{l_i}{d_i}$ . Due to the symmetric implementation, the second SweepArea is probed whenever an element from the first input stream arrives, and vice versa. Since all elements

of the probed SweepAreas qualify in a Cartesian product, the number of results produced on average at a single time instant is  $\frac{1}{d'} = \frac{1}{d_1} \frac{l_2}{d_2} + \frac{1}{d_2} \frac{l_1}{d_1}$ .

The Cartesian product generates a result whenever an element from the first input stream overlaps with one of the second input stream in terms of time intervals. The time interval attached to the result is the intersection of the contributing time intervals. Therefore, the average length  $l'$  of time intervals in the output stream corresponds to the average overlap of time intervals from the input streams. We thus investigate the overlap combinations for shifting a representative time interval  $I_1$  from the first input stream over a representative time interval  $I_2$  from the second input stream. Let  $l_1$  and  $l_2$  be the sizes of  $I_1$  and  $I_2$  respectively. Without loss of generality, let  $l_1 \geq l_2$ . Figure 22.3 shows the degree of overlap on the y-axis and the relative position of  $I_1$  on the x-axis. The resulting figure has the shape of a trapezoid. Under the assumption that the overlap combinations are equiprobable, the average overlap  $l'$  is the point at the y-axis where the surface area  $A_R$  of the rectangle defined by width  $a$  and height  $l'$  is equal to the surface area of the trapezoid  $A_T$ . This means,  $l' = \frac{A_R}{a} = \frac{A_T}{a} = \frac{\frac{1}{2}(l_1+l_2+l_1-l_2)l_2}{l_1+l_2} = \frac{l_1l_2}{l_1+l_2}$ .  $\square$

**Operator Formula 22.5** (Duplicate Elimination). Let  $n \in \mathbb{N}$ ,  $n > 0$ , denote the number of unique tuples in the input stream. Under the premise that tuples occur uniformly distributed, we estimate  $g' = g$ ,  $d' = d$ , and

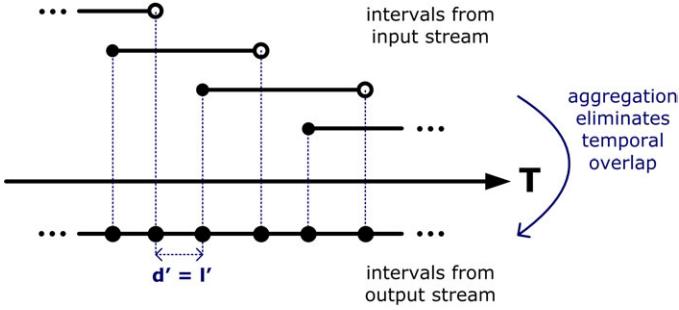
$$\begin{aligned} l' &= n \cdot d && \text{if } n \cdot d \leq l, \\ l' &= l && \text{otherwise.} \end{aligned} \tag{22.5}$$

**Explanation.** We distinguish between two cases. In the case  $n \cdot d > l$ , an incoming stream element does not match with an element in the state of the duplicate elimination in terms of value-equivalence due to temporal expiration. Such an element is appended directly to the output stream (see Algorithm 7). Because time intervals are not altered in this case,  $d' = d$  and  $l' = l$ .

The case  $n \cdot d \leq l$  implies that an incoming element finds a value-equivalent element in the state, a *duplicate*. The average offset between the start timestamps of these two elements is  $n \cdot d$ . As the average time interval length of both elements is  $l$ , the interval belonging to the incoming element exceeds that of the element in the state by an amount of  $n \cdot d$ . The duplicate elimination stores the tuple of the incoming element with the non-overlapping time interval fragment in the SweepArea and additionally causes the newly adjusted element to be output by inserting it into the priority queue  $Q$  (see line 15 in Algorithm 7). Hence, the length of time intervals in the output stream corresponds to the average length of non-overlapping time interval fragments, which is  $n \cdot d$ . Because any outgoing time intervals for value-equivalent elements are consecutive, their start timestamp distance is  $n \cdot d$ . The overall start timestamp distance  $d'$  results from  $\frac{1}{n} \cdot n \cdot d$  which implies  $d' = d$ .  $\square$

**Operator Formula 22.6** (Scalar Aggregation). For the scalar aggregation, we estimate  $g' = g$ ,

$$\begin{aligned} d' &= l' = \frac{g}{1 - (1 - \frac{g}{l})^{\frac{2l}{d}}} && \text{if } d < l, \\ d' &= d, \text{ and } l' = l && \text{otherwise.} \end{aligned} \tag{22.6}$$

Figure 22.4: Estimation of  $d'$  and  $l'$  for the scalar aggregation (case  $d < l$ )

**Explanation.** For  $d \geq l$ , each incoming element usually generates a new aggregate because there is no overlap between the time interval of the incoming element and that of a partial aggregate in the state. The average size of the state is 1 because any partial aggregates being in the state prior to the insertion of the new partial aggregate are released due to temporal expiration. Thus,  $d' = d$  and  $l' = l$ .

For  $d < l$ , each incoming element causes the removal of an aggregate from the state as well. However, the average size of the state usually exceeds 1 because partial aggregates are split at incoming start and end timestamps (see Algorithm 9 and Figure 22.4). The number of splits determines the parameters  $d'$  and  $l'$ . This split frequency can be computed analogously to Cardenas's formula [Car75] as follows. Let  $X$  be a sufficiently large timespan.  $\frac{X}{d}$  time intervals obtained from the input stream intersect with  $X$  on average. Hence,  $\frac{X}{d}$  start timestamps plus  $\frac{X}{d}$  end timestamps fall into  $X$  on average. Under the assumption that these timestamps are uniformly distributed throughout  $X$ , the probability that such a timestamp matches with an arbitrary but single time instant in  $G$  is  $\frac{g}{X}$ .<sup>1</sup> The probability that all  $2 \cdot \frac{X}{d}$  timestamps do not hit this time instant in  $G$  is  $(1 - \frac{g}{X})^{\frac{2X}{d}}$ . Consequently, the probability of the complementary event, namely that a start or end timestamp hits that time instant, is  $1 - (1 - \frac{g}{X})^{\frac{2X}{d}}$ . As  $X$  covers  $\frac{X}{g}$  time instants of  $G$ , the expected number of instants matched by at least one start or end timestamp is  $(1 - (1 - \frac{g}{X})^{\frac{2X}{d}}) \cdot \frac{X}{g}$ . In order to determine the average time distance  $d'$ , we have to compute  $\frac{X}{(1 - (1 - \frac{g}{X})^{\frac{2X}{d}}) \cdot \frac{X}{g}}$ . Replacing the auxiliary variable  $X$  by  $l$  returns  $d' = \frac{g}{1 - (1 - \frac{g}{X})^{\frac{2X}{d}}}$ . As the state is a list of aggregates having consecutive time intervals,  $l' = d'$  (see Figure 22.4).  $\square$

**Remark 22.1.** The estimation above becomes imprecise if the implicit assumption of uniformly distributed timestamps in  $X$  is violated. The extreme case is a scalar aggregation over an input stream with the constraints that (1)  $d$  results from a constant start timestamp distance, (2)  $l$  results from a constant time interval length, and (3)  $l$  is a multiple of  $d$ . To overcome the deficiency for this particular case, we propose to estimate  $d' = l' = d$ . In all other cases, the formula above provides acceptable estimates as confirmed by our experiments.

<sup>1</sup>Recall that the time spans between instants in  $G$  are equidistant with size  $g$  according to our definition of time granularity.

**Operator Formula 22.7** (Grouping with Aggregation). Let  $n \in \mathbb{N}$ ,  $n > 0$ , be the number of groups. For grouping with aggregation, we estimate  $g' = g$ ,

$$\begin{aligned} d' &= \frac{g}{n\left(1-\left(1-\frac{g}{l}\right)^{\frac{2l}{nd}}\right)}, \text{ and } l' = \frac{g}{1-\left(1-\frac{g}{l}\right)^{\frac{2l}{nd}}} \quad \text{if } n \cdot d < l, \\ d' &= d, \text{ and } l' = l \quad \text{otherwise.} \end{aligned} \tag{22.7}$$

**Explanation.** Recall that grouping with aggregation can be viewed as a combination of the following three steps. The first step is grouping. Based on a grouping function, the input stream is partitioned into  $n$  disjoint substreams, denoted by  $S_1, \dots, S_n$ , where each substream contains the elements of a group. The second step is the scalar aggregation which is applied to each substream  $S_i$  independently. The third step is the union of all aggregation output streams which provides the final output stream.

Based on the assumption that grouping distributes the elements of the input stream uniformly among the groups  $S_1, \dots, S_n$ ,  $d_i = n \cdot d$  and  $l_i = l$  for each substream  $S_i$ . Applying the operator formula for the scalar aggregation to the stream characteristics of the  $S_i$  delivers the input stream characteristics for the union formula whose result, in turn, gives us the final formulas for  $d'$  and  $l'$  listed above.  $\square$

**Operator Formula 22.8** (Split). Let  $l_{out} \in T$ ,  $l_{out} > 0$ , be the output interval length of the split operator (see Algorithm 15). For the split operator, we estimate  $g' = g$ ,

$$\begin{aligned} d' &= \frac{d \cdot l_{out}}{l}, \text{ and } l' = l_{out} \quad \text{if } l_{out} < l, \\ d' &= d, \text{ and } l' = l \quad \text{otherwise.} \end{aligned} \tag{22.8}$$

**Explanation.** In the case  $l_{out} < l$ , the split operator generates  $\frac{l}{l_{out}}$  outgoing elements for every incoming element of the input stream. Hence, the output rate  $\frac{1}{d'} = \frac{1}{d} \cdot \frac{l}{l_{out}}$ . We estimate the average interval length in the output stream at  $l_{out}$  due to the functionality of the split operator. If parameter  $l_{out} \geq l$ , then no intervals are split. Consequently, the output stream characteristics equals that of the input stream.  $\square$

**Operator Formula 22.9** (Granularity Conversion). The granularity conversion is the only operator that affects model parameter  $g$ . Let  $G$  be the time granularity of the input stream with granule size  $g$ . Let  $\hat{G}$  be the time granularity of the output stream with granule size  $\hat{g}$ , i.e.,  $\hat{g}$  corresponds to the granule size parameter given to the granularity conversion (see Algorithm 24). Because we do not permit the time granularity to become finer downstream,  $\hat{g} \geq g$ . For the special case that  $\hat{g} = g$ , we estimate  $d' = d$ ,  $l' = l$ ,  $g' = g$ . Otherwise, we estimate  $g' = \hat{g}$ ,

$$\begin{aligned} d' &= d, \text{ and } l' = l \quad \text{if } l \geq \hat{g}, \\ d' &= \frac{\hat{g}}{l} \cdot d, \text{ and } l' = \hat{g} \quad \text{otherwise.} \end{aligned} \tag{22.9}$$

**Explanation.** The first case is the common one in practice, namely  $l \geq \hat{g}$ . Because the average interval length is equal to or greater than the new granule size, any time interval intersects with a time instant in  $\hat{G}$ . As a consequence, no element is dropped by the granularity conversion. In order to determine  $l'$ , we have to estimate the effects of

rounding on the average time interval length. By assuming start timestamps to occur uniformly distributed within a granule, the average amount a time interval is shortened at its start turns out to be  $\frac{\hat{g}}{2}$ . The same assumption applied to end timestamps implies that, on average, a time interval is enlarged by  $\frac{\hat{g}}{2}$  at its end, since end timestamps are rounded up to the next larger time instant belonging to  $\hat{G}$ . Overall, the average length of time intervals remains the same.

In the second case, the average interval length  $l$  is less than  $\hat{g}$ . In this case, elements are dropped if they do not intersect with a time instant in  $\hat{G}$ . While assuming the time intervals to be uniformly distributed within granules, this event happens with a probability of  $\frac{\hat{g}-l}{\hat{g}}$ . The inverse probability  $1 - \frac{\hat{g}-l}{\hat{g}}$  corresponds to the probability that an element is appended to the output stream. Since  $l < \hat{g}$ , the rounding performed by the granularity conversion ensures that any time interval attached to an outgoing tuple has length  $\hat{g}$ . Dropping elements causes the average output stream rate  $\frac{1}{d'}$  to become less than the input stream rate  $\frac{1}{d}$ , here, by a factor of  $1 - \frac{\hat{g}-l}{\hat{g}}$ . From this follows that  $d' = \frac{\hat{g}}{l} \cdot d$ .  $\square$

**Operator Formula 22.10** (Time-Based Sliding Window). Let  $w \in T$  be the window size, usually  $w \gg g$ . For the time-based sliding window, we estimate

$$d' = d, l' = w, \text{ and } g' = g. \quad (22.10)$$

**Explanation.** The time-based window does not modify the start timestamps of incoming stream elements (see Algorithm 12). Therefore,  $d' = d$ . As the time-based window operator sets the interval length to the window size,  $l' = w$ .  $\square$

**Operator Formula 22.11** (Count-Based Sliding Window). Let  $N \in \mathbb{N}$  be the window size, where  $N > 0$ . For the count-based sliding window operator, we estimate

$$d' = d, l' = N \cdot d, \text{ and } g' = g. \quad (22.11)$$

**Explanation.** Count-based windows do not change the start timestamps of stream elements, thus,  $d' = d$ . As the end timestamp is set to the  $N$ -th future start timestamp, we estimate  $l' = N \cdot d$ .  $\square$

**Operator Formula 22.12** (Partitioned Window). Let  $n \in \mathbb{N}$ ,  $n > 0$ , be the number of groups, and  $N \in \mathbb{N}$  be the window size,  $N > 0$ . For the partitioned window without  $\Delta$ -optimization (see Algorithm 14), we estimate

$$d' = d, l' = n \cdot N \cdot d, \text{ and } g' = g. \quad (22.12)$$

**Explanation.** Similar to grouping with aggregation, the partitioned window can be viewed as a derived operator (see Section 8.2.3). Therefore, the cost formula is deduced analogously to the one of the grouping with aggregation operator, while replacing the formula for the scalar aggregation with the one for the count-based window in order to estimate the output stream characteristics of the groups.  $\square$

## 22.4 Estimating Operator Resource Allocation

Based on the previous estimations for stream characteristics, this section reveals how we obtain an estimate for operator resource consumption in terms of memory and processing costs (see Figure 22.2). Our estimations refer to the resources required at steady state, i. e., after the initialization of an operator according to the sliding windows, and under acceptable system load.

- $M_{op}$  denotes the average memory costs of an operator, i. e., the amount of memory allocated for state maintenance.
- $C_{op}$  denotes the average processing costs of an operator per unit in application time.

Based on our cost functions, the resource utilization of a query plan is computed by summing up the individual costs of the involved operators. The following analysis considers the dominant operator costs for the algorithms presented in Chapter 11. Our cost variables must be set according to the specific operator implementation.

### 22.4.1 Filter and Map

Filter and map are stateless, hence, no memory is allocated. We estimate the *memory consumption* at

$$M_\sigma = M_\mu \approx 0 \quad (22.13)$$

and the *processing costs* per time unit at

$$C_\sigma \approx \frac{1}{d} \cdot C_P + \frac{1}{d'} \cdot C_O \text{ and} \quad (22.14)$$

$$C_\mu \approx \frac{1}{d} \cdot (C_F + C_C) + \frac{1}{d'} \cdot C_O. \quad (22.15)$$

For every incoming element, on average  $\frac{1}{d}$  per time unit, the filter predicate is evaluated with costs  $C_P$ . Appending a result to the output stream costs  $C_O$ .  $\frac{1}{d'}$  results are generated by the filter during a single time unit, where  $d'$  is the estimated time distance between start timestamps of successive elements in the output stream (see parameter estimation in Section 22.3). For the map operator,  $C_F$  is the cost of a single evaluation of the mapping function and  $C_C$  denotes the costs to create a new element.

### 22.4.2 Union

We consider the union as a stateless operator in our analysis, despite the use of a priority queue in Algorithm 4 (see page 75).

**Remark 22.2** (Priority Queue). Our analysis disregards any priority queues employed in operators with multiple inputs such as the join and union due to the following reasons. First, the use of those queues is specific to the push-based processing methodology and needed to satisfy the ordering requirement of the output stream. Recall that this approach does not enforce a synchronized temporal processing of the input streams as

required in many other stream systems, e.g., [ACC<sup>+</sup>03, HMA<sup>+</sup>04, ABW06], but enables multi-threaded query processing. In those systems, the overhead of the priority queue is distributed across the inter-operator queues installed for operator communication. According to [BSW04], resources spent on inter-operator queues are negligible for unsaturated systems. Therefore, we argue that resources spent on priority queue maintenance are negligible in general. Second, under the premise that application time skew between streams is low, the queue size is small. Furthermore, PIPES is able to cope with application time skew between streams and latency in streams by applying *heartbeats* [SW04] (see Section 13.5). As a consequence, the memory and processing costs spent on queue maintenance are unimportant, a fact also confirmed by our experiments.

Our estimation for the *memory usage* is

$$M_U \approx 0. \quad (22.16)$$

For the *processing costs*, we estimate

$$C_U \approx \frac{1}{d'} \cdot C_O \quad (22.17)$$

where  $C_O$  denotes the costs to append a single element to the output stream and  $\frac{1}{d'}$  corresponds to the estimated output stream rate.

### 22.4.3 Join

$C_B$	cost of a single hash bucket operation: insertion and removal
$C_H$	costs for a single evaluation of the hash function
$C_P$	costs for a single evaluation of the query predicate
$C_R$	costs for a single evaluation of the remove predicate
$C_F$	costs for building a join result
$C_O$	costs for appending a single result to the output stream
$\sigma_{\bowtie}$	join selectivity factor
$s_i$	size of an element from input stream $S_i$ in bytes
$\#B_i$	number of hash buckets in SweepArea $i$
$C_{L_i}$	costs for insertion/deletion on the secondary data structure of the SweepArea $i$

Table 22.1: Variables specific to join costs

Algorithm 6 shows the computation of the binary join (see page 78). The state consists of two SweepAreas and a priority queue. According to Remark 22.2, the costs for the queue are not taken into account. Here, we discuss the costs for the symmetric hash join (SHJ), the most common join technique in stream processing. Our analysis assumes elements to be distributed equally among the hash buckets. Be aware that the symmetric nested-loops join can be viewed as a special SHJ with a single bucket in each hash table and zero costs to evaluate the hash function. Table 22.1 lists our cost variables.

**Remark 22.3** (Secondary Data Structure). Before we delve into the details of our cost analysis, we want to mention some important aspects with regard to the secondary data structure of SweepAreas. The secondary data structure links the elements of the primary data structure according to some order relation to efficiently identify expired elements. A separate cost variable, here named  $C_L$ , denotes the costs for maintaining the secondary data structure. The actual cost of  $C_L$  depend on the implementation. Recall that we choose the secondary data structure dependent upon the expiration pattern (see Section 11.3.3). In the case of a priority queue, some operations are logarithmic in the size of the SweepArea for the worst case. For this reason, we have two different variables for  $C_L$ , one for each SweepArea.

Our analysis of the memory consumption does not pinpoint the memory overhead incurred by the secondary data structure. Since the secondary data structure only contains references, one for each element in the primary data structure, the additional amount of allocated memory can be modeled by increasing the amount of variable  $s_i$  with the space requirements for a single reference in bytes.

### Memory Usage

We can deduce the average size of SweepAreas at steady state from the ratio between the average interval length and the average time progression induced by an incoming element. Recall that the average interval length corresponds to the average validity of elements. At delivery of a new element the opposite SweepArea is purged of expired elements. Let us consider the first SweepArea. Directly after purging, the average size of the first SweepArea is  $\frac{l_1}{d_1}$ , because only elements are kept that have a temporal overlap with the incoming element (see Section 11.2.4). Until the next eviction takes place, the SweepArea increases by  $\frac{d_2}{d_1}$  elements. Thus, the average size of the first SweepArea is  $\frac{l_1}{d_1} + \frac{d_2}{2d_1}$  elements. Due to symmetry arguments, the average size of the SweepArea structure is  $\frac{l_2}{d_2} + \frac{d_1}{2d_2}$ . Consequently, we can estimate the total join memory allocation by

$$M_{\bowtie} \approx \left( \frac{l_1}{d_1} + \frac{d_2}{2d_1} \right) \cdot s_1 + \left( \frac{l_2}{d_2} + \frac{d_1}{2d_2} \right) \cdot s_2. \quad (22.18)$$

### Processing Costs

We estimate the processing costs of the join by summing up costs for insertion, probing, result construction and purging:

$$C_{\bowtie} \approx C_{\text{insert}} + C_{\text{probe}} + C_{\text{result}} + C_{\text{purge}}. \quad (22.19)$$

**Insertion Costs ( $C_{\text{insert}}$ )** For every incoming element, (i) the hash function is called, (ii) the element is inserted into the primary data structure of the SweepArea, i. e., into a hash bucket, and (iii) the secondary data structure is updated. The secondary data structure of a SweepArea links elements by end timestamps to support efficient eviction. Adjusting

the linkage costs  $C_{L_i}$  for SweepArea  $i$ . As  $\frac{1}{d_1} + \frac{1}{d_2}$  elements arrive per time unit,

$$C_{insert} := \left( \frac{1}{d_1} + \frac{1}{d_2} \right) \cdot (C_H + C_B) + \frac{1}{d_1} \cdot C_{L_1} + \frac{1}{d_2} \cdot C_{L_2}. \quad (22.20)$$

**Probing Costs ( $C_{probe}$ )** Each incoming element is probed against the opposite SweepArea. This includes the evaluation of the hash function and the iteration over all elements in the corresponding hash bucket. As we assume the elements to be distributed equally among all buckets, a bucket of SweepArea  $i$  contains  $\frac{1}{\#B_i} \frac{l_i}{d_i}$  elements on average,  $i \in \{1, 2\}$ .

$$C_{probe} := \frac{1}{d_1 d_2} \cdot \left( \frac{l_2}{\#B_2} + \frac{l_1}{\#B_1} \right) \cdot C_P + \left( \frac{1}{d_1} + \frac{1}{d_2} \right) \cdot C_H \quad (22.21)$$

**Result Construction ( $C_{result}$ )** These costs arise from the number of join results produced per time unit multiplied by the costs for composing a result tuple. Eventually, all results need to be appended to the output stream.

$$C_{result} := \sigma_{\bowtie} \cdot \frac{l_1 + l_2}{d_1 \cdot d_2} \cdot (C_F + C_O) \quad (22.22)$$

**Purge Costs ( $C_{purge}$ )** On arrival of a new element, an eviction of expired elements from the opposite SweepArea is performed. While traversing the linkage provided by the secondary data structure, at least the first element needs to be accessed to check the remove predicate. For a single time unit, this check causes cost of  $\left( \frac{1}{d_1} + \frac{1}{d_2} \right) \cdot C_R$ . Whenever the remove predicate evaluates to true, the corresponding element is removed from the SweepArea. This includes the removal from the hash table (primary data structure) as well as the removal of the first element from the linkage (secondary data structure). The first removal can be done with constant costs, even for the nested-loops case, because the secondary data structure provides a reference to the element in the primary data structure. Be aware that the cost variables  $C_{L_i}$  denote the worst case costs for adjusting the secondary data structures, i. e., the linkage. Depending on the expiration pattern, the secondary data structure can be a priority queue or a list. In the first case,  $C_{L_i}$  is constant, whereas it is logarithmic in the size of the corresponding SweepArea in the second case. The traversal stops at the first element that does not satisfy the remove predicate. Because each element in the status will expire due to the sliding windows specified, cost of  $\left( \frac{1}{d_1} + \frac{1}{d_2} \right) \cdot (C_R + C_B) + \frac{1}{d_1} \cdot C_{L_2} + \frac{1}{d_2} \cdot C_{L_1}$  result from removals. Altogether, we define the purging costs per time unit as follows:

$$C_{purge} := \left( \frac{1}{d_1} + \frac{1}{d_2} \right) \cdot (2C_R + C_B) + \frac{1}{d_1} \cdot C_{L_2} + \frac{1}{d_2} \cdot C_{L_1}. \quad (22.23)$$

#### 22.4.4 Duplicate Elimination

The costs analysis for the duplicate elimination refers to the hash-based SweepArea implementation described in Section 11.5.5 (see page 80). Table 22.2 displays the cost

$C_B$	cost of a single hash bucket operation: insertion, removal, update
$C_L$	costs for insertion/deletion on the secondary data structure of the SweepArea
$C_U$	costs for an update on the secondary data structure of the SweepArea
$C_H$	costs for a single evaluation of the hash function
$C_P$	costs for a single evaluation of the query predicate
$C_R$	costs for a single evaluation of the remove predicate
$C_Q$	cost of a single priority queue operation: insertion, extract minimum
$C_M$	costs for accessing and comparing the minimum of the priority queue
$C_E$	costs for a check if the priority queue is empty
$C_C$	costs for creating a new element
$C_O$	costs for appending a single result to the output stream
$n$	number of unique tuples in the input stream
$s$	size of an element from the input stream in bytes
$\#B$	number of hash buckets

Table 22.2: Variables specific to duplicate elimination costs

variables. We assume the elements to be distributed equally among the hash buckets. The state consists of a SweepArea and a priority queue. Recall that the SweepArea additionally links the elements in the hash table according to end timestamps. Depending on the type of expiration pattern, this secondary data structure is usually a priority queue or a linked list (see Section 11.3.3). Costs  $C_L$  and  $C_U$  have to be set accordingly.

### Memory Usage

At steady state  $\frac{l}{d}$  elements of the input stream are valid at an arbitrary time instant on average. Let us consider the case  $n \cdot d \leq l$ . The SweepArea contains only elements having unique tuples, whereas the priority queue temporarily stores results obtained from incoming elements by removing the time interval overlap with the duplicate in the state. As a consequence, the size of the SweepArea is  $n$  and the size of the priority queue is  $\frac{l}{d} - n$ . Altogether, the total size of the state is

$$M_\delta \approx \frac{l}{d} \cdot s. \quad (22.24)$$

If  $n \cdot d > l$ , no duplicates can be detected, hence, the priority queue is empty but the size of the SweepArea is  $\frac{l}{d}$ .

### Processing Costs

We estimate the processing costs for the duplicate elimination at

$$C_\delta \approx C_{\text{purge}} + C_{\text{probe}} + C_{\text{insert}} + C_{\text{transfer}}. \quad (22.25)$$

The contributing costs are defined as follows.

**Purge Costs ( $C_{\text{purge}}$ )** In the case  $n \cdot d \leq l$ , no elements can be purged from the SweepArea because elements are updated continuously (see line 14 in Algorithm 7). Nevertheless, the remove predicate is evaluated once at delivery of a new element. Otherwise, expired elements are removed as usual. As the input rate is equal to the output rate, every incoming element purges an element from the SweepArea on average. This necessitates two evaluations of the remove predicate (one hit and one miss) plus the removal from the SweepArea, including the deletion from the bucket and the update of the linkage.

$$C_{\text{purge}} := \begin{cases} \frac{1}{d} \cdot C_R & \text{if } n \cdot d \leq l, \\ \frac{1}{d} \cdot (2C_R + C_B + C_L) & \text{otherwise.} \end{cases} \quad (22.26)$$

**Probing Costs ( $C_{\text{probe}}$ )** Every incoming element probes the SweepArea. This action requires a call to the hash function to search for value-equivalent elements. In the case  $n \cdot d \leq l$ , the corresponding bucket contains  $\frac{n}{\#B}$  elements for which the query predicate is evaluated. Otherwise, the average bucket size is  $\frac{1}{\#B} \frac{l}{d}$ .

$$C_{\text{probe}} := \begin{cases} \frac{1}{d} \cdot (C_H + C_P \cdot \frac{n}{\#B}) & \text{if } n \cdot d \leq l, \\ \frac{1}{d} \cdot (C_H + C_P \cdot \frac{1}{\#B} \frac{l}{d}) & \text{otherwise.} \end{cases} \quad (22.27)$$

**Insertion Costs ( $C_{\text{insert}}$ )** Our insertion cost estimate covers not only the costs for inserting new elements but also the costs for updating elements. This is particularly important for the case  $n \cdot d \leq l$  where updates occur frequently. In this case, a new incoming element is typically identified as a duplicate, i. e., there exists a value-equivalent element in the SweepArea that has a temporal overlap. As a consequence, the element in the SweepArea is replaced with the new element whose time interval has been set to the non-overlapping interval fragment. Variable  $C_C$  denotes the costs to create a new element. This substitution causes an update on the bucket plus costs  $C_U$  to update the linkage. In addition, this element is inserted into the priority queue  $Q$  with costs  $C_Q$ . For the other case ( $n \cdot d > l$ ), no element inside the SweepArea satisfying the query predicate is found and the incoming element can be inserted directly into the SweepArea and appended to the output stream (see lines 8 – 10 in Algorithm 7).

$$C_{\text{insert}} := \begin{cases} \frac{1}{d} \cdot (C_H + C_B + C_U + C_C + C_Q) & \text{if } n \cdot d \leq l, \\ \frac{1}{d} \cdot (C_H + C_B + C_L + C_O) & \text{otherwise.} \end{cases} \quad (22.28)$$

**Transfer Costs ( $C_{\text{transfer}}$ )** The transfer costs show the additional costs arising in the case  $n \cdot d \leq l$  from the use of the priority queue. The costs for inserting elements into this queue have been considered in  $C_{\text{insert}}$ . Independent of the case, it is checked whether the queue is empty or not at delivery of a new element. Per application time unit, this check causes costs of  $\frac{1}{d} \cdot C_E$ . For the case  $n \cdot d \leq l$ , the queue is not empty at steady state. On average, a single element is extracted from the queue and appended to the output stream because the input rate is equal to the output rate. The minimum can be extracted from the queue if its start timestamp is less than  $\min_{t_s}$ . This check is performed twice for an

incoming element on average (one hit and one miss). In addition, it is checked twice that the queue is not empty. Removing the minimum from the queue has cost  $C_Q$ .

$$C_{transfer} := \begin{cases} \frac{1}{d} \cdot (2(C_E + C_M) + C_Q + C_O) & \text{if } n \cdot d \leq l, \\ \frac{1}{d} \cdot C_E & \text{otherwise.} \end{cases} \quad (22.29)$$

#### 22.4.5 Scalar Aggregation

$C_M$	costs for a single evaluation of the <i>merger</i> function
$C_I$	costs for a single evaluation of the <i>initializer</i> function
$C_E$	costs for a single evaluation of the <i>evaluator</i> function
$C_C$	costs for the creation of a new partial aggregate
$C_P$	costs for a single evaluation of the query predicate
$C_R$	costs for a single evaluation of the remove predicate
$C_L$	cost of a single list operation: insert and remove
$C_O$	costs for appending a single result to the output stream
$s$	average size of a partial aggregate in the state in bytes

Table 22.3: Variables specific to scalar aggregation costs

The following cost analysis refers to Algorithm 9 (see page 90). Table 22.3 explains the cost variables. The SweepArea is implemented as a list ordered by start timestamps. If the list supports positional access, the list operations can be performed with constant costs due to the following reasons. First, elements are removed from the SweepArea by the method *extractElements* at the beginning of the list or via the iterator *qualifies*. Second, an insertion either appends a new partial aggregate at the end of the list or inserts it at a specific position during the partial state traversal performed to update the state.

#### Memory Usage

Due to temporal expiration all partial aggregates with an end timestamp less than or equal to the start timestamp of the incoming element are removed from the state (see lines 26 – 29 in Algorithm 9). Prior to this extraction, the state is updated. Two new partial aggregates are generated whenever the start timestamp of an incoming time interval splits the time interval of a partial aggregate (see left part of Figure 11.3). The new partial aggregate whose end timestamp is set to the value of the incoming start timestamp is removed from the state by the method *extractElements*. Hence, this type of split does not increase the size of the state. Therefore, only the number of splits triggered by end timestamps have to be considered when computing the size of the state. We deduce the average memory usage of the scalar aggregation from Cardenas's formula [Car75] as done in the explanation of Operator Formula 22.6, but regard solely splits by end

timestamps.

$$M_\alpha \approx \begin{cases} \frac{l}{g} \left(1 - \left(1 - \frac{g}{l}\right)^{\frac{l}{d}}\right) \cdot s & \text{if } d < l, \\ s & \text{otherwise.} \end{cases} \quad (22.30)$$

In the case  $d \geq l$ , the state contains only a single element on average.

### Processing Costs

We estimate the processing costs of the scalar aggregation as the sum of costs to initialize and update partial aggregates, and to extract partial aggregates from the state, including the computation and transfer of final aggregates:

$$C_\alpha \approx C_{init} + C_{update} + C_{extract}. \quad (22.31)$$

**Initialization Costs ( $C_{init}$ )** The ratio  $\frac{d}{d'}$  corresponds to the number of new aggregates generated for an incoming element on average. A new partial aggregate is created whenever (i) an existing partial aggregate is split, or (ii) the start or end timestamp of an incoming element is greater than the end timestamp of the last aggregate in the list. While in the first case the new partial aggregate inherits the tuple of the existing partial aggregate, the *initializer* function is evaluated on the tuple of the incoming element in the second case. In both cases, the new partial aggregates are inserted into the SweepArea (list) with costs  $C_L$ . Induced by our model parameters  $d$  and  $l$ , a stream exhibits a FIFO expiration pattern. Under this premise, both cases happen with the same probability. Therefore, we estimate the costs for an incoming element in the case  $d < l$  at  $\frac{1}{2} \frac{d}{d'} (C_C + C_L) + \frac{1}{2} \frac{d}{d'} (C_C + C_I + C_L)$ . As  $\frac{1}{d}$  elements arrive on average per time unit, we estimate

$$C_{init} := \begin{cases} \frac{1}{2d'} \cdot (2C_C + C_I + 2C_L) & \text{if } d < l, \\ \frac{1}{d} \cdot (C_C + C_I + C_L) & \text{otherwise.} \end{cases} \quad (22.32)$$

For the case  $d \geq l$ , the average size of the state is 1. Hence, each incoming element produces a new partial aggregate which has to be initialized and inserted into the SweepArea.

**Update Costs ( $C_{update}$ )** For the case  $d < l$ ,  $\frac{l}{g} \left(1 - \left(1 - \frac{g}{l}\right)^{\frac{l}{d}}\right)$  end timestamps intersect with a time interval of length  $l$  on average. This means, the time interval of an incoming element overlaps with that number of partial aggregates in the state on average which, in turn, have to be updated. Recall that prior to an update the query predicate is evaluated. While Algorithm 9 describes the update by removing the element in the list at the current position of the iterator and inserting the new partial aggregate, our implementation actually performs an update, i. e., the corresponding partial aggregate in the list is simply replaced with the new one. For this reason, our cost formula includes only a single list operation of cost  $C_L$ . In the other case ( $d \geq l$ ), every incoming element causes the initialization of a new partial aggregate. The costs for the initialization are captured in

$C_{init}$ , nonetheless, at least a single call to the query predicate is necessary to verify that the incoming element does not overlap with a partial aggregate in the state.

$$C_{update} := \begin{cases} \frac{1}{d} \cdot \frac{l}{g} \cdot \left(1 - \left(1 - \frac{g}{l}\right)^{\frac{l}{d}}\right) \cdot (C_P + C_C + C_M + C_L) & \text{if } d < l, \\ \frac{1}{d} \cdot C_P & \text{otherwise.} \end{cases} \quad (22.33)$$

**Remark 22.4.** Although this approximation predicts that all elements in the state have to be updated on average, the elements at the start and end of the list do not overlap necessarily with the incoming element. Hence, our formula slightly overestimates.

**Extraction Costs ( $C_{extract}$ )** Extraction is a sequential scan of the list triggered by every incoming element. During traversal all partial aggregates having an end timestamp less than or equal to the start timestamp of the incoming element are extracted from the state and the corresponding final aggregate is emitted as result. Each expired element consequently causes a call to the *evaluator* function to compute the final aggregate value. Since eviction is performed after updating the state, the traversal stops with a miss at the partial aggregate having the start timestamp of the latest incoming element. Those overlap misses cost  $\frac{1}{d} \cdot C_R$  per time unit. As each incoming element produces  $\frac{d}{d'}$  aggregates on average, which are removed from the state sometime due to finite windowing constructs, cost of  $\frac{1}{d} \frac{d}{d'} (C_R + C_E + C_L)$  result from temporal overlap hits followed by list removals. Recall that  $d = d'$  for the case  $d \geq l$ .

$$C_{extract} := \frac{1}{d'} \cdot (C_R + C_E + C_L + C_O) + \frac{1}{d} \cdot C_R \quad (22.34)$$

#### 22.4.6 Grouping with Aggregation

$C_G$	costs to determine the group identifier
$C_A$	costs to acquire the SweepArea belonging to a group identifier
$C_Q$	cost of a single priority queue operation: insertion, extract minimum
$C_M$	costs for accessing and comparing the minimum of the priority queue
$C_E$	costs for a check if the priority queue is empty
$C_C$	costs for attaching the group information to a final aggregate
$s_p$	average size of a partial aggregate in the state in bytes
$s_a$	size of a final aggregate in bytes
$s_g$	size of a group identifier in bytes
$s_m$	size of a map entry in bytes
$n$	number of groups

Table 22.4: Variables specific to grouping with aggregation costs

Algorithm 10 shows the implementation of the grouping with aggregation (see page 95). Table 22.4 lists the cost variables for the following analysis. The state consists of  $n$  SweepAreas (groups), two priority queues, and a map to manage the groups (see

Section 11.5.8). As indicated by Table 22.4, we use the same cost variables for both priority queues since we assume both queues to have the same size at steady state. Furthermore, our analysis requires the grouping function to uniformly distribute incoming elements across the groups.

Since grouping with aggregation uses the SweepArea implementation of the scalar aggregation, our cost formulas are based on the corresponding formulas in Section 22.4.5. However, these formulas have to be modified to reflect the effects of the grouping correctly. All occurrences of parameter  $d$  need to be substituted for  $n \cdot d$ , including the case differentiation and the formula that estimates  $d'$ . We label the accordingly modified cost functions with a prime, e.g.,  $C'_{extract}$ .

### Memory Usage

We derive the size of a single SweepArea from the cost formula  $M_\alpha$  by replacing  $d$  with  $n \cdot d$ . For the case  $n \cdot d < l$ , the map *groups* and the priority queue  $G$  contain  $n$  entries at steady state. Furthermore, we estimate the size of the priority queue  $Q$  merging the final aggregates at  $n$ , i.e., on average, the queue contains an aggregate from every group. Actually, the size of  $Q$  depends on the time skew between start timestamps across groups (compare with Remark 22.2). The variables denoting the respective element size for the various data structures can be found in Table 22.4.

$$M_\gamma \approx \begin{cases} n \cdot \frac{l}{g} \left( 1 - \left( 1 - \frac{g}{l} \right)^{\frac{l}{n \cdot d}} \right) \cdot s_p + n \cdot (s_m + s_g + s_a) & \text{if } n \cdot d < l, \\ n \cdot s_m + s_p + s_g + s_a & \text{otherwise.} \end{cases} \quad (22.35)$$

In the case  $n \cdot d \geq l$ , only a single group contains a partial aggregate. For this reason, the average size of queues  $G$  and  $Q$  is also 1. However, the map *groups* still contains an entry for every group.

### Processing Costs

We estimate the processing costs by summing up the costs for (i) updating the groups, (ii) collecting the final aggregates from the groups, and (iii) transferring the results:

$$C_\gamma \approx C_{update} + C_{gather} + C_{transfer}. \quad (22.36)$$

**Update Costs ( $C_{update}$ )** The update costs include the costs to determine, access, and update the SweepArea (group) to which an incoming element belongs. The computation of the group identifier and the retrieval of the corresponding SweepArea from the map *groups* costs  $\frac{1}{d} \cdot (C_G + C_A)$  because the operations are performed for every incoming element. The costs incurred by procedure **UPDATE** are  $n \cdot (C'_{init} + C'_{update})$  where  $C'_{init}$  and  $C'_{update}$  denote the initialization and update costs for a single SweepArea per time unit.

$$C_{update} := \frac{1}{d} \cdot (C_G + C_A) + n \cdot (C'_{init} + C'_{update}) \quad (22.37)$$

**Gathering Costs ( $C_{gather}$ )** The costs  $C_{gather}$  comprise the costs to (i) extract final aggregates from the SweepAreas, (ii) decorate them with grouping information, and (iii) insert them into queue  $Q$  to bring them into the right order. We estimate the costs to extract final aggregates at  $n \cdot C'_{extract}$  per time unit. Be aware that parameter  $d'$  here refers to the inter-arrival time in the output of the grouping with aggregation operator, whereas parameter  $d'$  in  $C'_{extract}$  describes the inter-arrival time in the output of a single group.

An incoming element leads to  $\frac{d}{d'}$  final aggregates on average ( $1 \leq \frac{d}{d'} \leq 2$ ). Therefore,  $\frac{1}{d} \frac{d}{d'}$  aggregates can be extracted from the SweepAreas across all groups per time unit at steady state. Because the method *extractElements* needs to be called on the SweepArea with the minimal start timestamp, the priority queue  $G$  is accessed before (see line 19 in Algorithm 10). This induces costs  $C_E$  to check that  $G$  is not empty, costs  $C_M$  to access the minimal group identifier, costs  $C_A$  to retrieve the corresponding SweepArea, and costs  $C_Q$  to extract the minimum from queue  $G$ . During extraction each final aggregate is decorated with grouping information, which costs  $C_C$ , and inserted into queue  $Q$  afterwards to ensure the output stream order requirement. Eventually, the group identifier is inserted back into queue  $G$  (see line 31). Each of the latter two insertions costs  $C_Q$ .

$$C_{gather} := \frac{1}{d'} \cdot (C_E + C_M + C_A + 3C_Q + C_C) + n \cdot C'_{extract} \quad (22.38)$$

Note that for the case  $n \cdot d \geq l$ , this cost formula is still valid because  $d = d'$ . However, in this case, the size of queues  $Q$  and  $G$  is 1.

**Transfer Costs ( $C_{transfer}$ )** Variable  $C_{transfer}$  describes the costs arising from calls to procedure TRANSFER (see line 33 in Algorithm 10 and also page 76). The first summand  $\frac{1}{d'} \cdot (C_E + C_M + C_Q)$  specifies the costs for extracting elements from queue  $Q$  if all conditions in the procedure are satisfied. Recall that  $\frac{1}{d'}$  corresponds to the output rate. Because the procedure is called for every incoming element, the second summand  $\frac{1}{d} \cdot (C_E + C_M)$  denotes the costs for checks that caused the internal loop to break.

$$C_{transfer} := \begin{cases} \frac{1}{d'} \cdot (C_E + C_M + C_Q) + \frac{1}{d} \cdot (C_E + C_M) & \text{if } n \cdot d < l, \\ \frac{1}{d'} \cdot (C_E + C_M + C_Q) + \frac{1}{d} \cdot C_E & \text{otherwise.} \end{cases} \quad (22.39)$$

For the case  $n \cdot d \geq l$ , we estimate the size of queue  $Q$  at 1, hence, the loop breaks because the queue is empty. Note that the costs to append results to the output stream are included in  $C'_{extract}$ .

### 22.4.7 Split

The split operator (see Algorithm 15 on page 105) employs a SweepArea and a priority queue for state maintenance. The SweepArea consists of two lists, the primary list is ordered by start timestamps, the secondary list by end timestamps.

$C_I$	costs to insert a single element into the SweepArea
$C_D$	costs to delete a single element from the SweepArea
$C_R$	costs for a single evaluation of the remove predicate
$C_Q$	cost of a single priority queue operation: insertion, extract minimum
$C_E$	costs for a check if priority queue is empty
$C_C$	costs for creating a new element
$C_S$	costs to check if a split is required
$C_O$	costs for appending a single result to the output stream
$s$	size of an element from the input stream in bytes

Table 22.5: Variables specific to split costs

### Memory Usage

Let us consider the case  $l_{out} < l$  first. We estimate the size of the SweepArea at  $\frac{l}{d} \cdot s$  because  $\frac{l}{d}$  elements intersect on average at an arbitrary time instant. The procedure SPLITUP fills the initially empty priority queue with new elements by splitting all elements from the SweepArea. On average  $\frac{l}{d} \cdot \frac{d}{l_{out}}$  elements enter the queue when processing an incoming element. We estimate the average queue size at one half of this amount, i. e.,  $\frac{1}{2} \frac{l}{l_{out}}$ . Multiplied by the size of an element, we deduce the following formula:

$$M_\zeta \approx \begin{cases} \left(\frac{l}{d} + \frac{1}{2} \frac{l}{l_{out}}\right) \cdot s & \text{if } l_{out} < l, \\ 0 & \text{otherwise.} \end{cases} \quad (22.40)$$

In the case  $l_{out} \geq l$ , incoming elements are directly appended to the output stream.

### Processing Costs

The processing costs of the split operator are composed as follows:

$$C_\zeta \approx C_{insert} + C_{split} + C_{purge}. \quad (22.41)$$

**Insertion Costs ( $C_{insert}$ )** For the case  $l_{out} < l$ , every incoming element is inserted into the SweepArea. Cost variable  $C_I$  corresponds to the entire insertion costs, i. e., the costs to insert the element into both lists. In the case  $l_{out} \geq l$ , no split is required. Therefore, elements are not inserted into the SweepArea.

$$C_{insert} := \begin{cases} \frac{1}{d} \cdot C_I & \text{if } l_{out} < l, \\ 0 & \text{otherwise.} \end{cases} \quad (22.42)$$

**Split Costs ( $C_{split}$ )** Besides the costs of procedure SPLITUP, variable  $C_{split}$  also includes the costs for the check whether a split is required or not. This check causes cost of  $\frac{1}{d} \cdot C_S$  per time unit. For the case  $l_{out} < l$ , every incoming element is split which costs  $C_C$  because a new element is created. As the average time progression between two successive elements

arriving at the operator is  $d$ , the procedure SPLITUP generates  $\frac{d}{l_{out}}$  new elements for every element in the SweepArea during a single execution on average. Hence,  $\frac{d}{l_{out}} \cdot \frac{l}{d}$  elements are created per time unit. Prior to the creation of a new element the split condition is verified. Each of these new elements is inserted into the priority queue. At the end of the same procedure call, every element is removed from the queue and appended to the output stream. Therefore, each element leads to two queue operations with costs  $C_Q$  and a single operation with costs  $C_E$ . The latter is required to check that the queue is not empty. For the case  $l_{out} \geq l$ , the SweepArea is empty. Therefore, the split condition evaluates to false and the element is appended directly to the output stream.

$$C_{split} := \begin{cases} \frac{1}{d} \cdot (C_S + C_C + \frac{l}{l_{out}}(C_S + C_C + 2C_Q + C_E + C_O)) & \text{if } l_{out} < l, \\ \frac{1}{d} \cdot (C_S + C_O) & \text{otherwise.} \end{cases} \quad (22.43)$$

We omitted the costs for evaluating the query predicate. Our implementation does not check this predicate for the split operator because it always returns *true* (see Section 11.7.1).

**Purge Costs ( $C_{purge}$ )** For the case  $l_{out} < l$ , an incoming element triggers the eviction of an element from the SweepArea on average. Thus, costs of  $2C_R + C_D$  arise from the evaluation of the remove predicate (one hit and one miss) and the actual costs for removing the expired element. Note that  $C_D$  summarizes the removal costs of the primary and secondary data structure of the SweepArea. Even in the case  $l_{out} \geq l$ , it needs to be checked if the SweepArea is empty.

$$C_{purge} := \begin{cases} \frac{1}{d} \cdot (2C_R + C_D) & \text{if } l_{out} < l, \\ \frac{1}{d} \cdot C_R & \text{otherwise.} \end{cases} \quad (22.44)$$

#### 22.4.8 Granularity Conversion

The granularity conversion rounds the start and end timestamps of each incoming element according to the new time granularity. Since no state is required, we estimate the *memory consumption* at

$$M_\Gamma \approx 0 \quad (22.45)$$

and the *processing costs* at

$$C_\Gamma \approx \begin{cases} \frac{1}{d} \cdot (2C_S + C_T + C_C + C_O) & \text{if } l \geq \hat{g}, \\ \frac{1}{d} \cdot (2C_S + C_T) + \frac{1}{d'} \cdot (C_C + C_O) & \text{otherwise,} \end{cases} \quad (22.46)$$

where  $C_S$  denotes the costs to round a timestamp,  $C_C$  the costs to create a new element,  $C_O$  the costs to append that element to the output stream, and  $C_T$  the costs to check if the time interval is well-defined (see Algorithm 24 on page 187). In the first case, the interval length is sufficiently large so that no elements are dropped due to rounding. In the second case ( $l < \hat{g}$ ), the rounding and time interval check are performed for every incoming element, but an element is only created and appended to the output stream

with a probability of  $1 - \frac{\hat{g}-l}{\hat{g}}$ . This fact is included in the estimation of the average output stream rate  $\frac{1}{d'}$ .

### 22.4.9 Time-Based Sliding Window

The cost analysis for the time-based sliding window resembles that of the granularity operator because both operators are stateless and simply create a result for every incoming element by keeping the tuple and attaching a new time interval. We estimate the *memory usage* at

$$M_{\omega^{time}} \approx 0 \quad (22.47)$$

and the *processing costs* at

$$C_{\omega^{time}} \approx \frac{1}{d} \cdot (C_S + C_C + C_O) \quad (22.48)$$

where  $C_S$  denotes the costs to compute a new end timestamp. Costs  $C_C$  and  $C_O$  are defined as in the analysis of the granularity conversion (see Section 22.4.8). Be aware that we here refer to Algorithm 12 (see page 97) and not to the extended version with control over the window advance.

### 22.4.10 Count-Based Sliding Window

$C_R$	costs for a single evaluation of the remove predicate
$C_C$	costs for creating a new element
$C_I$	costs to insert a single element into the SweepArea
$C_D$	costs to delete a single element from the SweepArea
$C_O$	costs for appending a single result to the output stream
$s$	size of an element from the input stream in bytes
$N$	window size

Table 22.6: Variables specific to count-based window costs

In contrast to the time-based sliding window, the count-based variant requires a state of  $N$  elements implemented as a linked list (see Algorithm 13 on page 99). Therefore, we estimate the average *memory usage* at

$$M_{\omega^{count}} \approx N \cdot s. \quad (22.49)$$

We deduce the *processing costs* per time unit from the input rate and SweepArea operations as follows:

$$C_{\omega^{count}} \approx \frac{1}{d} \cdot (C_R + C_D + C_C + C_O + C_I). \quad (22.50)$$

At steady state every incoming element causes a call to the remove predicate followed by an extraction of the first element from the SweepArea. Be aware that the remove predicate always evaluates to true at steady state. A new element is created from the

extracted one by copying the tuple and the start timestamp. The end timestamp is set to the start timestamp of the incoming element. Variable  $C_C$  includes these costs. Thereafter, the new element is appended to the output stream and the incoming element is inserted into the list (SweepArea).

#### 22.4.11 Partitioned Window

$C_G$	costs to determine the group identifier
$C_A$	costs to acquire the SweepArea belonging to a group identifier
$C_Q$	cost of a single priority queue operation: insertion, extract minimum
$C_M$	costs for accessing and comparing the minimum of the priority queue
$C_E$	costs for a check if the priority queue is empty
$C_C$	costs for attaching the group information to a final aggregate
$C_P$	costs for a single evaluation of the query predicate
$C_I$	costs to insert a single element into the SweepArea
$C_D$	costs to delete a single element from the SweepArea
$C_O$	costs for appending a single result to the output stream
$s$	size of an element from the input stream in bytes
$s_m$	size of a map entry in bytes
$s_g$	size of a reference to a SweepArea in bytes
$n$	number of groups
$N$	window size

Table 22.7: Variables specific to partitioned window costs

The state of the partitioned window is composed of  $n$  SweepAreas, one for each group, two priority queues, and a map (see Algorithm 14 on page 102). Our analysis relies on the same assumptions as already stated for the grouping with aggregation, i.e., elements are distributed uniformly across groups and both priority queues have equal sizes. Furthermore, our investigation does not regard the  $\Delta$ -optimization (see Section 11.6.3) because we assume application time skew between the different groups to be negligible. In this case, the input rate of the partitioned window is equal to the output rate. Moreover, the secondary list proposed in Section 11.6.3 for the SweepArea implementation is superfluous. Hence, the costs  $C_I$  and  $C_D$  to adjust the SweepArea simply correspond to the costs for appending an element at the end of the list and removing the list's first element respectively (see Table 22.7).

#### Memory Usage

As usual, let us consider steady state. The size of each SweepArea is  $N$ . The size of the map  $groups$  and the queue  $G$  is  $n$  because both data structures store references to all SweepAreas. As in the cost analysis of the grouping with aggregation (see Section 22.4.6), we estimate the size of the priority queue  $Q$  merging the results of the various groups at  $n$ . By multiplying the sizes of the data structures with the corresponding element sizes

shown in Table 22.7, we infer the following formula:

$$M_{\omega^{\text{partition}}} \approx n \cdot (N \cdot s + s_m + s_g + s). \quad (22.51)$$

### Processing Costs

We have four different cost factors identifying costs to (i) access the right group, (ii) extract an element from that group, (iii) insert the incoming element, and (iv) transfer the results:

$$C_{\omega^{\text{partition}}} \approx C_{\text{group}} + C_{\text{extract}} + C_{\text{insert}} + C_{\text{transfer}}. \quad (22.52)$$

**Grouping Costs ( $C_{\text{group}}$ )** At delivery of a new element the grouping function is invoked on its tuple to determine the corresponding group identifier. Thereafter a reference to the corresponding SweepArea is obtained from the map *groups*.

$$C_{\text{group}} := \frac{1}{d} \cdot (C_G + C_A) \quad (22.53)$$

**Extraction Costs ( $C_{\text{extract}}$ )** During extraction (see lines 16 – 20 in Algorithm 14) the query predicate is evaluated once to verify that the SweepArea contains  $N$  elements. Then, a new element is created whose end timestamp is set to the one of the incoming element. The new element is inserted into the priority queue  $Q$  and deleted from the SweepArea.

$$C_{\text{extract}} := \frac{1}{d} \cdot (C_P + C_C + C_Q + C_D) \quad (22.54)$$

**Insertion Costs ( $C_{\text{insert}}$ )** Every incoming element is inserted into the SweepArea with costs  $C_I$  (see line 21). In addition, the entry in the priority queue  $G$  has to be updated because the priority of the corresponding SweepArea – determined by the element with the smallest start timestamp – changed due to the prior extraction (see lines 24 – 25). We use cost variable  $C_Q$  to model the costs for decreasing the value of a SweepArea’s key in the priority queue  $G$  because the worst case costs of this operation are equal to those of the other operations covered by  $C_Q$  (see Table 22.7), namely,  $\mathcal{O}(\lg n)$  if the implementation is based on a binary heap.

$$C_{\text{insert}} := \frac{1}{d} \cdot (C_I + C_Q) \quad (22.55)$$

**Transfer Costs ( $C_{\text{transfer}}$ )** Under the assumption that the application time skew between the groups is negligible and parameter  $\Delta$  is sufficiently large, the input rate is equal to the output rate. We estimate the costs for the procedure TRANSFER (see line 33) at

$$C_{\text{transfer}} := \frac{1}{d} \cdot (2(C_E + C_M) + C_Q + C_O). \quad (22.56)$$

The explanation is identical to that of the transfer costs for the duplicate elimination (see Section 22.4.4 case  $n \cdot d \leq l$  for details).

# 23 Adaptive Resource Management

Our cost model enables the system to quantify the impact of both adaptation methods on the resource consumption of a query plan. In the following, Section 23.1 clarifies when a DSMS needs to apply our adaptation methods, Section 23.2 describes their runtime behavior, and Section 23.3 proposes a suitable adaptation strategy. The chapter concludes with an outline on future work.

## 23.1 Runtime Adaptivity

The resource manager is the runtime component that controls resource utilization (see Figure 21.1). Based on our cost model, the resource manager is able to estimate the average resource utilization of query plans with only low computational overhead – constant costs per operator. We distinguish between two cases that require adaptive resource management: 1) *changes in query workload*, and 2) *changes in stream characteristics*. In both cases, the resource manager gains from our cost model because it can determine in advance the degree of adjustments necessary to keep resource usage within bounds. As a consequence, resource management is *proactive* rather than being reactive as would be the case without a cost model.

### 23.1.1 Changes in Query Workload

Whenever a new query is posed, the DSMS checks with our cost model if sufficient resources are available to evaluate the query. In particular, it considers adjustments to window sizes and time granularities to release resources in the running query graph. If enough resources can be allocated without violating QoS constraints, the query is accepted and integrated into the running query graph by the query optimizer. This includes updates to the QoS bounds at graph nodes and the assignment of memory to the newly installed stateful operators. Otherwise, the query is rejected. Whenever a query is removed and thus a significant amount of resources is released, our cost model can be used to determine settings for window sizes and time granularities that improve overall QoS.

### 23.1.2 Changes in Stream Characteristics

To keep up with changing stream rates, the resource manager monitors the average inter-arrival time (parameter  $d$ ) of all raw streams, and triggers the necessary updates based on the threshold parameter as explained in Section 22.2.3. A low threshold leads to high adaptivity at the expense of higher costs due to frequent re-estimations. Based on the new estimates, the resource manager can react appropriately to adjust overall

resource utilization. As adjustments to window and granule sizes need a certain time to become fully effective, a DSMS should reserve a separate amount of resources to keep up with unpredictable behavior such as bursts in raw streams. From our empirical experiences, we suggest allocating 80 percent of the overall resources for query execution, while keeping the remaining 20 percent in reserve. A greater reserve allows the value of the threshold parameter to be increased since adaptivity is less crucial in this case. If upper bounds for the stream rates are known for an application, the resource manager can take advantage of this information to compute the resource utilization for the worst case, namely, if bursts occur. Proactive resource management is feasible in those applications where forecasting techniques predict either trends in stream rates or recurring stream patterns.

## 23.2 Adaptation Effects

While our adaptation methods affect stateless operators instantly, stateful operators react to adaptations with a delay due to temporal expiration (see Section 11.2.4). Hence, it is important to know the points in time (i) when these effects start becoming visible and (ii) when they are fully achieved. As both events rely on the temporal expiration of elements in the operator state, they depend on the concrete operator implementation and the application time progression inside the operator. At first, adaptations influence the next operators downstream of the corresponding window or granularity conversion. Thereafter, the effects proceed downstream with the transmission of stream elements.

### 23.2.1 Changes to the Window Size

If  $w$  and  $\hat{w}$  denote the old and new window size, respectively, and the change to the window size is performed at application time instant  $t$ , the change starts to be observable downstream of the window at  $t + \min\{w, \hat{w}\}$  and reaches its full impact at  $t + \max\{w, \hat{w}\}$ . This inference results directly from temporal expiration.

### 23.2.2 Changes to the Time Granularity

If  $g$  and  $\hat{g}$  denote the old and new granule size, respectively, a change at time instant  $t$  affects operators downstream of the granularity conversion immediately, i. e., at delivery of the first element which has been adjusted to the new time granularity. The effects of the granularity change are fully achieved after the maximum interval length existent in the downstream plan. The duration corresponds to the maximum application time progression required for purging the state of old elements, i. e., elements being in the state prior to the granularity change. Be aware that the window and coalesce operators define an upper bound for the maximum interval length.

## 23.3 Adaptation Strategy

In order to validate the suitability of our cost model for adaptive resource management, we implemented a strategy for the resource manager which has to tackle the following optimization problem.

### 23.3.1 Optimization Problem

For a query graph, let  $C$  and  $M$  be the given system resource bounds for processing costs and memory usage respectively. The resource manager has to adjust the window sizes and time granularities of the corresponding operators in the graph within the pre-defined QoS ranges with the objective function to maximize overall QoS, while keeping the estimated overall resource utilization within bounds.

### 23.3.2 Heuristic Solution

Let  $[min, max]$  denote the QoS range. For the granularity conversion,  $min$  denotes the coarsest and  $max$  the finest time granularity. The basic idea of our adaptation strategy is to adjust all windows sizes and time granularities to the same fraction  $f \in \mathbb{R}_{[0,1]}$  of their QoS range. This means to set them to the closest allowed value nearest to  $min + f \cdot (max - min)$ . Due to our definition of time granularity (see Section 20.1), the number of possible granularities within the QoS bounds is limited. In contrast, window sizes can be changed in a more fine-grained manner. Therefore, our strategy is implemented in two steps: 1) a rough approximation which sets the granularities followed by 2) a fine tuning of the window sizes. More precisely:

1. Let  $p \in \mathbb{N}$ ,  $p \geq 1$ , be an approximation parameter. For  $j = 0, 1, \dots, p$ , the resource manager computes the overall resource estimates  $C_j$  and  $M_j$  using our strategy above with  $f = \frac{p-j}{p}$  until the constraints  $C_j \leq C$  and  $M_j \leq M$  are satisfied. The granularities are fixed to their values in the final step.
2. A fine tuning of factor  $f$  for the window sizes is achieved by a bisection strategy.

Note that the strategy relies on a positive correlation between the factor  $f$  and the overall resource utilization.

## 23.4 Extensions and Future Work

More sophisticated strategies for adaptive resource management constitute an interesting direction for future work, in particular strategies that take subquery sharing into account. Furthermore, we do not have an operator formula for coalesce and difference yet. We also attempt to improve our cost model by replacing the strict case differentiations specified for some operators, e. g., the scalar aggregation, with more continuous transitions.

Although our cost model already produces relatively accurate estimates based on the average inter-arrival time and interval length given by a physical stream, it might be

preferable to consider the variance of these parameters in addition, especially in highly dynamic environments. Recall that we already take fluctuations of the average start timestamp offsets into account for raw streams to trigger updates of cost estimates (see Section 22.2.3). It is arguable whether or not more sophisticated statistical models are better suited for cost estimation than ours, bearing in mind that the estimates are likely to be more precise, but at the expense of higher computational costs that might conflict with scalability issues (see discussion in Section 22.2.4). Note that our cost model is already more extensive and complex than existing ones because it covers a broader set of operators and relies on more advanced stream characteristics.

Studying the use of forecasting and pattern recognition techniques with the aim to extract information about future changes in stream characteristics and query workload is a promising approach towards proactive resource management. In this part, however, we put emphasis on the presentation, semantics, and effects of our adaptation techniques, the resource management architecture, and the development of our cost model since those aspects are prerequisites for adaptive resource management.

# 24 Experiments

This chapter summarizes the most significant results extracted from a set of experiments under various settings, including real-world as well as synthetic data sets. To validate our cost estimations for numerous applications, we picked out a different data set for each experiment. At the beginning, Sections 24.1 and 24.2 demonstrate the effects of adjustments to the window size and time granularity on operator resource usage. Afterwards, Section 24.3 compares window reduction with random load shedding in terms of impact on query results and operator resource consumption. Finally, the experiment investigated in Section 24.4 validates the scalability of our adaptation approach by executing a large query graph under increasing query workload for several hours.

We used the PIPES stream processing infrastructure as platform [KS04, CHK<sup>+</sup>06]. The experiments were performed on a Windows XP workstation with 1 GB main memory and a 2.8 GHz CPU. We measured the actual costs for the cost variables used in our model and counted the corresponding events at runtime, e.g., calls of the hash function. For a better overview, we plotted the measured processing costs as a percentage of saturation, i.e., 100% processing costs correspond to the capacity the system became saturated. Some figures have two y-axes. The y1-axis depicts memory costs, whereas the y2-axis shows processing costs.

## 24.1 Adjustments to the Window Size

In our first experiment, we point out that the resource usage of the temporal join operator is linear in the window size. This experiment was based on data streams obtained from a commercial system, called *i-Plant*, which monitors industrial production processes [CHK<sup>+</sup>06]. Two industrial sensors provided the input streams for our experiment. The first one measured the flow rate of a machine drying paper, while the second one measured the rate at which the paper is rolled up by the next machine. Significant differences in these rates indicate that the paper is at risk to tear. We computed a window similarity join to detect pairs of rates that differ by more than a given tolerance. Note that this join does not produce many results due to its alarming nature. Because we run this continuous query standalone, the utilized system resources were relatively low. Recall that a DSMS executes multiple thousands of such queries concurrently.

Figure 24.1 shows the effects of adjusting the window sizes together with the estimations obtained from our cost model. At the beginning the window sizes were set to 60 seconds, and after 80 seconds the windows were reduced to 30 seconds. The resulting effect started to be visible after the new window size ( $80 + 30 = 110$ ) and reached its full impact after the old window size ( $80 + 60 = 140$ ), which agrees with our predictions. At 150 seconds, we set the window size back to 60 seconds. This time, the effects began to be observable

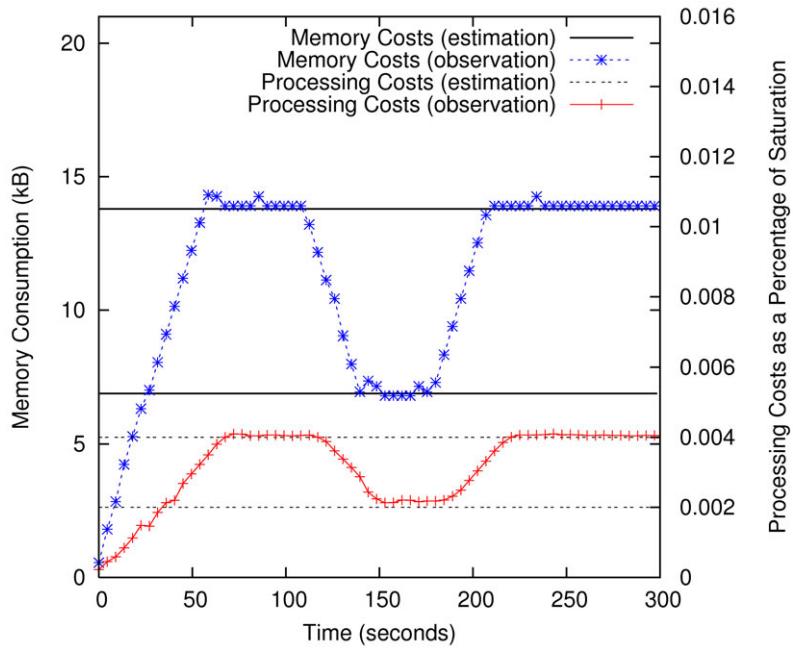


Figure 24.1: Changing the window size

after the old window size ( $150 + 30 = 180$ ) and were fully visible after the new window size ( $150 + 60 = 210$ ).

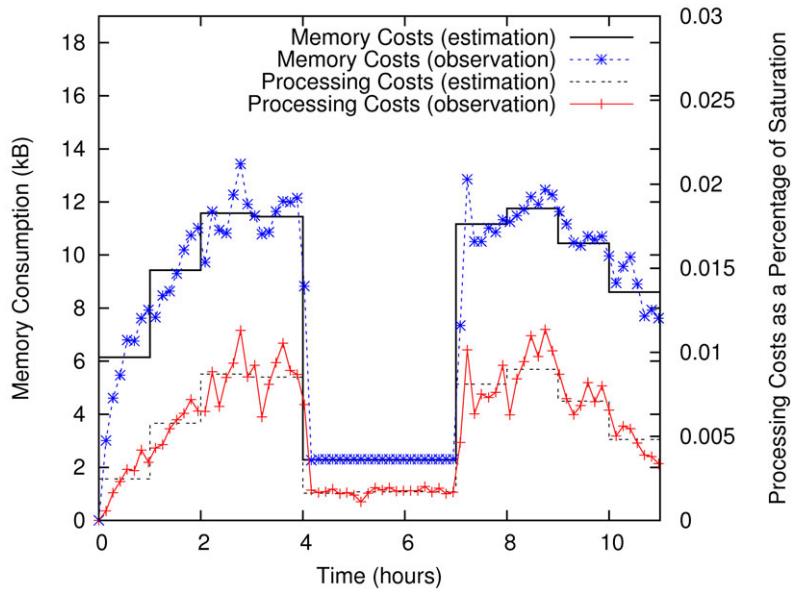


Figure 24.2: Changing the time granularity

## 24.2 Adjustments to the Time Granularity

Our second experiment demonstrates the impact of a time granularity adjustment on the resource usage of a scalar aggregation. We used real-world data streams with traffic data collected in the Freeway Service Patrol Project (FSP) in 1993. We posed a continuous sliding window query with a COUNT aggregate, i.e., we computed the number of vehicles that passed a loop detector during the past 10 minutes. As the original granularity of the timestamps delivered from the sensor is relatively fine ( $\frac{1}{60}$  seconds), almost every input element of the aggregation results in two output elements. During hour 4 to hour 7 of the experiment, we set the time granularity to 10 seconds. Since the average lag between the events that two successive vehicles pass a sensor is lower than that granularity, we expected a significant decrease in the resource usage, which agrees with our observations shown in Figure 24.2. Moreover, our cost model gives a good approximation to the measured costs. Note that the estimates were updated hourly and thus this experiment also outlines the adaptivity potential of our approach. Having in mind that this real-world data set inherently contains correlation between tuple values and timestamps, the experiment also confirms that re-estimations compensate any existing correlation effects.

## 24.3 Window Reduction vs. Load Shedding

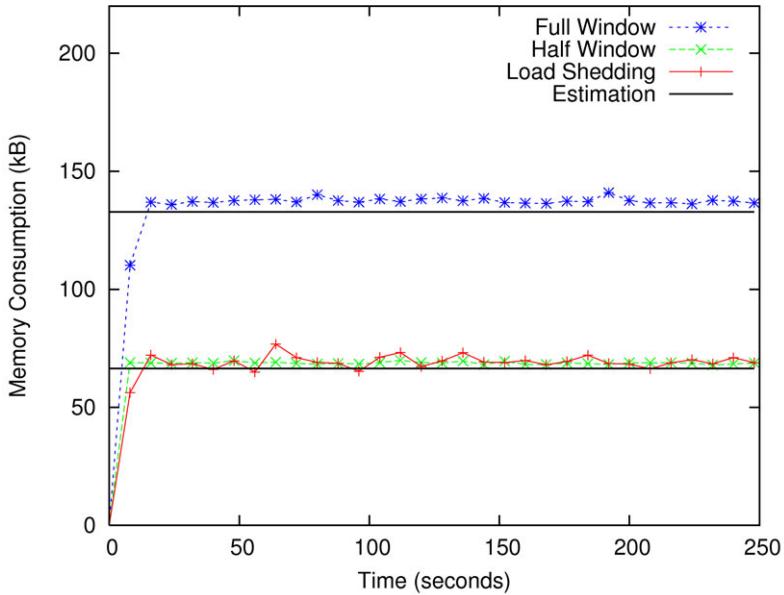


Figure 24.3: Savings in memory usage

The third experiment compares window reduction with random load shedding [TCZ<sup>+</sup>03]. We computed an equi-join of two streams, feeding each stream with random integers between 0 and 49 at a rate of 100 elements per second. By applying a time-based sliding

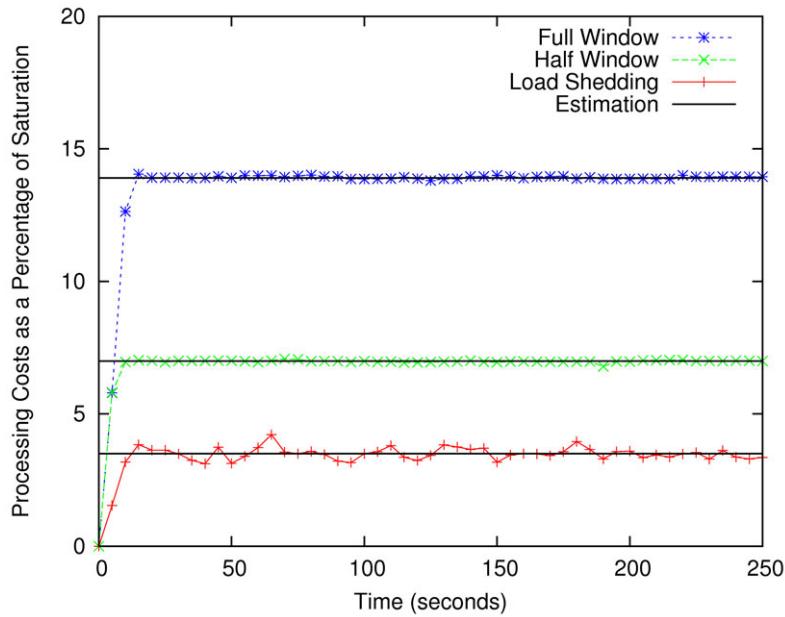


Figure 24.4: Savings in processing costs

window of 10 seconds, the join needed about 140 kB memory as shown in Figure 24.3. We compared both techniques with the aim to achieve a memory saving of 50 per cent. For load shedding, this means to drop an element from the input stream with a probability of

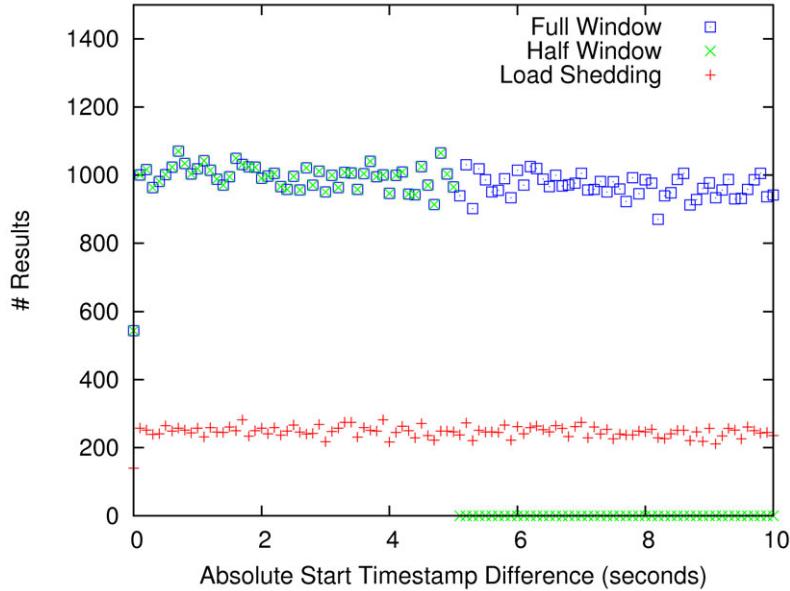


Figure 24.5: Output quality

$\frac{1}{2}$ . For window reduction, the window sizes have to be halved which means assigning validities of 5 seconds. As Figure 24.3 illustrates, both methods achieved their goal. The reason why the memory estimation closely underestimates the measured values is that our measurements include the memory allocation of the priority queue used in the join. However, this experiment confirms that these costs arising from application time skew between the input streams are negligible. With regard to processing costs, load shedding is more effective than window reduction – see Figure 24.4. While half a window size causes only half of the processing costs, load shedding reduces the processing costs by a factor of 4 because elements of both input streams are dropped with a probability of  $\frac{1}{2}$ .

Much more interesting is the number of results produced. In general, results of time-based sliding window joins are considered to be more important if the start timestamps of their constituents are temporally close to each other [GÖ03b]. We plotted the effect of both techniques on the number of results in Figure 24.5. For two joining elements, the x-axis shows the absolute value of the temporal difference of their start timestamps. For the sake of clarity, only each fifth value was plotted. In the case of load shedding, only about  $\frac{1}{2} \cdot \frac{1}{2} = \frac{1}{4}$  of the original join results were produced because half of each input is dropped. In contrast to load shedding, window reduction produced the exact join result for halved windows. This means, window reduction preserves the results computed from elements whose start timestamps are temporally close to each other. Recall that those are considered to have a higher utility to the application [GÖ03b]. On the contrary, load shedding is likely to drop some of them. As a consequence, window reduction should be preferred to load shedding in those applications where exact results are important and resources need to be restricted, in particular memory usage.

## 24.4 Scalability and Adaptivity

Our last experiment confirms the scalability of our techniques and our cost model. Furthermore, it proves that the techniques are suitable for adaptive resource management. We generated 5000 continuous queries with time-based sliding windows. The query workload was randomly composed of the plans shown in Table 24.1 including time-based windows ( $\omega$ ), scalar aggregations ( $\alpha$ ), granularity conversions ( $\Gamma$ ), and joins ( $\bowtie$ ) where 33% of all joins were nested-loops joins and the remaining 67% were hash joins. Join selectivities were chosen between 0.005 and 0.02. In addition, we randomly set the QoS constraints for the window sizes with range from 1 to 30 minutes, and for the time granularities with range from 1 to 10000 milliseconds. We used 50 synthetic data streams with Poisson-distributed arrival rates between 2 and 600 elements per minute. This implies that during our experiment 50 threads ran concurrently.

The resource manager applied our adaptation strategy explained in Section 23.3 with the objective function to adapt system memory usage to 100 MB. In order to demonstrate adaptivity as well as scalability, we steadily increased query workload. We started with 5000 queries. Since the operator states were empty at the beginning, it took about 30 minutes until the states became steady in size. Note that this time period corresponds to the maximum window size. After 1 hour, we started to pose 10 new queries every 72 seconds throughout a duration of 30 minutes. Finally, we created a burst of 500

Plan Types	Percentage
$\omega(S_1) \bowtie \omega(S_2)$	30 %
$(\omega(S_1) \bowtie \omega(S_2)) \bowtie \omega(S_3)$	20 %
$((\omega(S_1) \bowtie \omega(S_2)) \bowtie \omega(S_3)) \bowtie \omega(S_4)$	10 %
$(\omega(S_1) \bowtie \omega(S_2)) \bowtie (\omega(S_3) \bowtie \omega(S_4))$	10 %
$\alpha(\Gamma(\omega(S)))$	10 %
$(\alpha(\Gamma(\omega(S_1)))) \bowtie \omega(S_2)$	10 %
$(\alpha(\Gamma(\omega(S_1)))) \bowtie (\alpha(\Gamma(\omega(S_2))))$	10 %

Table 24.1: Plan generation

new queries after 2 hours of total runtime. As shown in Figure 24.6, our resource manager succeeded in controlling the resource utilization dynamically. The periods of underestimation result from adaptation delays (see Section 23.2). While the operators of a new query allocate resources immediately when the query is installed, the reaction of the resource manager in the form of window reductions requires a certain time to become fully effective. This can be avoided if the resource manager adapts resource utilization in advance, either by preventive strategies or by delaying the start of new queries. In our DSMS we prefer to keep a portion of the system resources as reserve for that purpose, which is also reasonable for the case of unpredictable stream behavior such as bursts.

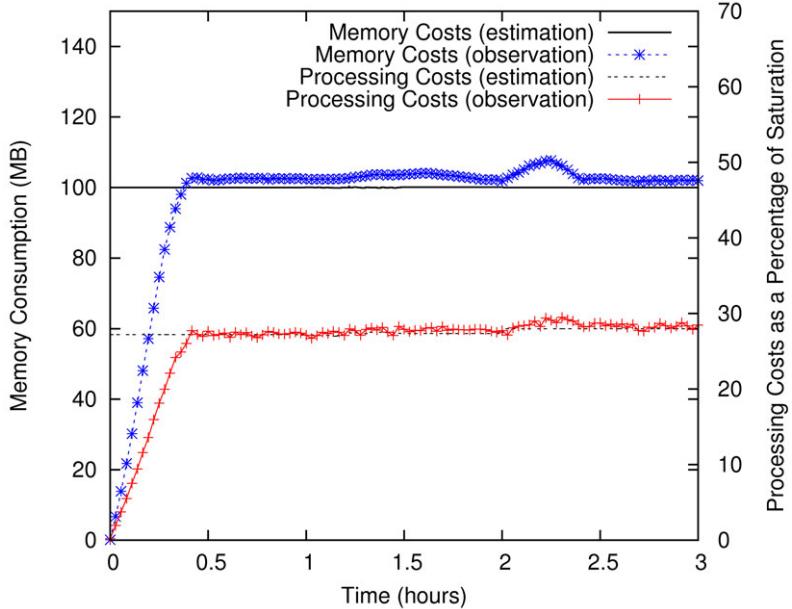


Figure 24.6: Scalability of adaptation techniques and cost model

We want to highlight the following points with regard to scalability:

- The experiment ran for several hours with an initial workload of 5000 continuous queries executed concurrently. Moreover, we steadily increased the workload

by initiating bursts of 10 up to 500 new queries entering the system to enforce adaptations.

- Our adaptation strategy succeeded in keeping resource requirements within bounds.
- We had a passable memory usage around 100 MB.
- The processing costs were about 30% of the system saturation capacity. We chose this extent of resources because we did not want to have side effects caused by our profiling overhead. Be aware that we monitored all our operator costs exactly, as specified in our cost model, which includes counting the function and predicate calls for example.
- As our cost model only gives approximations to the actual costs, and cost estimation runs bottom up in a query plan, the estimates get worse as the plan increases in height. For this reason, we measured the average relative error in dependence of the plan height and observed a slight increase by at most 1% per level.

To summarize, both techniques turn out to be appropriate for adaptive resource management. The observed behavior is close to the one predicted by our cost model, which consequently provides a sufficient degree of accuracy.



# 25 Related Work

Time plays an important role in data stream processing because the majority of applications provide stream elements with a timestamp or time interval. As a consequence, nearly every stream algebra relies on temporal information [ACC<sup>+</sup>03, ABW03, KS05, CJSS03]. In addition, finite windows are a common technique to avoid blocking and to restrict the resource requirements of stateful operations [GÖ03b]. Changing the window size and time granularity are consequently two widely applicable techniques. Although the basic idea of changing the window size has been mentioned in [MWA<sup>+</sup>03], a detailed analysis of the semantics and impact on resource usage has not been published so far. In addition, QoS constraints for window sizes and time granularities have not been considered in the literature before.

In prior work [CKSV06], we defined the semantics of our adaptation techniques and explained their usage with regard to adaptive memory management. We presented the impact of those techniques on a query plan neither formally nor experimentally. This part overcomes these deficiencies while it seamlessly enriches our previous work with novel insights. The main new contributions are summarized as follows. (i) We not only consider memory usage but also take processing costs into account. (ii) We develop an appropriate cost model. (iii) We analyze the runtime behavior of both techniques. (iv) We introduce our resource management architecture and present a suitable adaptation strategy. (v) Finally, experimental studies validate the potential of our approach involving its scalability. Our cost model is general enough to serve as a basis for cost-based query optimization as well. In the following, we concentrate on cost models for continuous queries to estimate resource requirements and methods for adaptive resource management. The interested reader is referred to [BB05] for a survey of work on adaptive query processing.

An overview on related cost models is given in Section 25.1, while other approaches endeavoring to estimate the resource consumption are outlined in Section 25.2. Sections 25.3 and 25.4 present alternative techniques for reducing the resource usage by load shedding or approximate query processing and scheduling. Finally, Section 25.5 discusses the applicability of our adaptation techniques and cost formulas for the positive-negative approach.

## 25.1 Costs Models

Adaptive query processing often relies on a cost model to pick out the best query execution plan according to a heuristic method [SLMK01]. For data streams, the initial work on rate-based query optimization [VN02] introduces a cost model for SPJ queries with the main goal of maximizing the output rate of join trees. A different objective for sliding window multi-way joins is to minimize the number of intermediate tuples [GÖ03a].

The cost model in [AN04] deals with the optimum placement of random load shedding operators to find an execution plan that minimizes resource usage when computational resources are insufficient. This problem is similar to ours but the techniques applied are entirely different. Furthermore, our cost model is more extensive than those in the papers mentioned, as it considers not only the memory usage of SPJ queries but also processing costs, and contains further important operators such as scalar aggregation. It is also more detailed because it shows the impact of time granularity on resource consumption as well as cost of temporal expiration. The cost model in [BSLH05] is used to answer in advance whether the QoS requirements of a query can be met or not. The QoS of running queries is guaranteed at operating system level. In contrast to [BSLH05], our approach is dynamic because it adapts to changing stream characteristics and query workload, but at the expense of weaker QoS guarantees. Moreover, the underlying operator semantics is different.

## 25.2 Estimation of Resource Usage

*k-Mon* [BSW04] is a query processing architecture to detect automatically useful stream characteristics such as certain data or arrival patterns and exploits these so-called *k-constraints* to reduce the runtime state for continuous queries. In [ABB<sup>+</sup>04], a broad range of continuous queries are categorized into two classes based on their asymptotic memory requirements. An algorithm is presented which determines whether or not a given query can be evaluated with bounded memory. Both papers neither address our techniques and QoS constraints nor give a cost model for estimating the resource usage of a query plan.

Arasu and Widom consider the problem of resource sharing for sliding window aggregates in [AW04b] and also prefer the computation of exact aggregates over approximations on the grounds that many applications such as applications over financial data require exact answers. Furthermore, [DGIM02] reveals that aggregation functions like MIN and MAX are inherently difficult to approximate. [KWF06] investigates and exploits similarities in streaming aggregate queries with varying selections or varying windows to share resources. The proposed dynamic sharing approach does not require a static multi-query optimization prior to query execution.

## 25.3 Load Shedding and Approximate Queries

There has also been considerable research on approximate query processing over data streams in recent years. *Load shedding* [TCZ<sup>+</sup>03, BDM04, GWYL05] belongs to this class. Aurora [ACC<sup>+</sup>03] uses QoS information provided by external applications to control the degree of load to shed, usually CPU load. While standard load shedding randomly drops elements from a stream, semantic load shedding aims at maximizing QoS by dropping those elements that have a lower utility to the application. Compared to load shedding, our adaptation techniques produce exact query results for the windows sizes and time granularities selected by the resource manager. This consequently enables query re-optimizations at runtime [ZRH04, KYC<sup>+</sup>06]. Note that query optimization becomes

quite limited for random load shedding as, for example, it is known from joins that the join over samples of its inputs does not return a sample of the join’s output [CMN99].

Another advantage of window reduction is that, even for small windows and high system load, it preserves those results computed from elements that are temporally close to each other, which is just the basic idea of sliding windows. In contrast, random load shedding is very likely to drop some of those results. The adaptive load shedding approach for windowed stream joins being applied in [GWYL05] employs a selective processing concept by joining only a subset of highly beneficial tuples. Different runtime adaptations to (i) input stream rates, (ii) time-correlation between streams, (iii) and join directions effectively shed CPU load for costly stream joins over tuples with set-valued or weighted set-valued attributes. Although this approach assumes memory resources to be sufficient and stores all elements in the windows, a combination with our window reduction method could be used to handle the memory limitations. Work on approximate query processing using synopses, e. g., [DGR03, DGM05, AN04, ANWS06], is altogether complementary to our work which provides exact query results with regard to the selected QoS.

## 25.4 Scheduling

The impact of operator scheduling on system resources at runtime is studied in [CCZ<sup>+</sup>03, BBDM03]. While the first paper [CCZ<sup>+</sup>03] proposes scheduling strategies to meet QoS specifications, *Chain* scheduling [BBDM03, BBD<sup>+</sup>04] adaptively minimizes inter-operator queues in query graphs. These works are complementary to ours since they neither address the memory allocated for operator states, which is under normal system load the dominating part [BSW04], nor do they analyze operator costs in detail.

## 25.5 Applicability

It should be emphasized that our adaptation techniques and estimations for stream characteristics are equally applicable to the *positive-negative approach* implemented by STREAM [MWA<sup>+</sup>03, ABW06] and Nile [HMA<sup>+</sup>04, GHM<sup>+</sup>07] (see also Section 14.1). In this case,  $d$  corresponds to the average time distance between timestamps of successive positive stream elements, and  $l$  is average time distance between timestamps of a positive and its corresponding negative element in a stream. The window operator ( $\text{W-Expire}$ ) can easily supply the required information to initialize parameter  $l$ . Implementing the time granularity is straightforward. Rounding the timestamps is similar to the time-interval approach, timestamps of positive elements are rounded like start timestamps and timestamps of negative elements like end timestamps (see Chapter 20). Finally, we want to point out that our cost model is general enough to serve as an appropriate basis for cost-based query optimization in these systems as well.



## 26 Conclusions

This part introduced a novel approach for cost-based resource management of continuous sliding window queries in DSMSs. We investigated two adaptation techniques that rely on adjustments of window sizes and time granularities at runtime. The techniques are compatible with the generally accepted stream semantics and do not collide with query optimizations at runtime, as query results are exact for the selected QoS settings – an advantage compared to load shedding. Practicable QoS specifications enable the resource manager to dynamically control the resource usage within user-defined QoS bounds. A solid cost model was developed that allows estimation of the impact of both techniques on the average resource utilization of a query plan. This cost model serves not only as basis for our resource manager but also for our query optimizer. The simplicity of this model is a desired feature as the efficient evaluation of cost formulas guarantees scalability in terms of query workload. The results obtained from our extensive experimental studies prove the potential and scalability of both adaptation methods and validate the accuracy of our cost model. Overall, our approach gracefully combines cost-based resource management with query optimization at runtime because it gives the DSMS the flexibility to control resource allocation while it retains the ability to reoptimize continuous queries.



## **Part IV**

# **Dynamic Plan Migration**



## 27 Introduction

Query optimization at runtime is important for data stream management systems because subscribed queries are long-running and underlying stream characteristics such as arrival rates and data distributions may vary over time. In addition, the query workload may gradually change. For the latter case, a DSMS can, for instance, gain from re-optimization at runtime by exploiting subquery sharing to save system resources.

Two major tasks arise from the optimization of continuous queries at runtime:

1. The query optimizer needs to identify a suitable plan with optimization potential. For this purpose, a DSMS keeps a plethora of metadata involving *runtime statistics*, e. g., stream rates and selectivities.
2. The query optimizer has to replace the currently running, yet inefficient plan with a semantically equivalent but more efficient new plan.

The latter transition at runtime is called *dynamic plan migration* [ZRH04]. Dynamic plan migration is easy as long as query plans only consist of *stateless* operators, such as filter and map, and inter-operator queues. In order to perform the migration, it is sufficient to (i) pause the execution of the old plan, (ii) drain out all existing elements in the old plan, (iii) substitute the new plan for the old plan, and finally (iv) resume the execution. In contrast to *stateless* operators, *stateful* operators like join and aggregation must maintain information derived from previously received elements as state information to produce correct results. Migration strategies for query plans with stateful operators are complex because it is non-trivial to appropriately transfer the state information from the old to the new plan.

Besides the essential requirement of correctness, a migration strategy should take the following performance objectives into account:

- The migration strategy should *not stall query execution* for a significant timespan as catching up with processing could cause system overload afterwards.
- The migration strategy should *produce results continuously* during migration – the smoother the output rate the better.
- The migration strategy should *minimize the migration duration and migration overhead* in terms of system resources like memory and CPU costs.

To the best of our knowledge, a general approach to dynamic plan migration has only been addressed in [ZRH04] so far. The authors proposed two different migration strategies, *moving states* (MS) and *parallel track* (PT). MS computes the state of the new plan instantly from the state of the old plan at migration start. Afterwards the old plan is discarded and the execution of the new plan is started. In order to apply MS, the query

optimizer requires a detailed knowledge about the operator implementations because it needs to access and modify state information. Despite the fact that it may be possible to define those transitions correctly for arbitrary transformation rules involving joins, aggregation, duplicate elimination etc., the implementation will be very complex and inflexible. For this reason, we prefer the second strategy proposed in [ZRH04], namely PT, as starting point for our black box migration approach. In contrast to MS, PT runs both plans in parallel for a certain timespan to initialize the new plan gradually with the required state information. Although [ZRH04] claims that PT is generally applicable to continuous queries over data streams, we identified problems if stateful operators other than joins occur in a plan, e. g., duplicate elimination and scalar aggregation. In this paper we develop a general migration strategy which overcomes these deficiencies.

Our continuous query semantics presented in [KS05] and the resultant, powerful set of transformation rules serve as the basis for our optimization approach (see also Part II). Based on this solid semantic foundation, our migration strategy requires for correctness that both plans – the old and new one – produce snapshot-equivalent results. Recall that the transformation rules presented in Chapter 10 guarantee this property.

As our approach generalizes PT, we treat the old and new plan as black boxes, i. e., a plan represents a composition of arbitrary operators from our algebra over the same set of input streams. At any time instant, both boxes must produce snapshot-equivalent output streams. Be aware that the term *plan* here usually refers to a subplan, namely, to the portion of a executed query plan which is to be optimized. The core idea of our approach is to define an appropriate *split time* representing an explicit point in application time that separates the processing of the old and new plan. Stream elements delivered by the input streams are assigned to the plans as follows. For all tuples valid at time instants prior to the split time results are computed by the old plan, otherwise by the new plan. The migration is finished as soon as the time progression in any input stream reached the split time.

The contributions of this part can be summarized as follows:

- We show that PT fails to cope with plans involving stateful operators other than joins.
- We propose our general solution for the dynamic plan migration of continuous queries, called *GenMig*, prove its correctness, and discuss implementation issues. In addition to the direct algorithm, we suggest two optimizations.
- We analyze the performance of GenMig with regard to the above objectives for a migration strategy, and compare GenMig with PT.

The rest of this part is organized as follows. In Chapter 28, we demonstrate that PT fails for plans with other stateful operators than joins. Our general plan migration strategy overcoming this imperfection is presented in Chapter 29. Chapter 30 shows the results of our experimental studies. Related work is discussed in Chapter 31. Finally, Chapter 32 concludes this part.

# 28 Problems of the Parallel Track Strategy

This chapter consists of two sections. Section 28.1 elaborates on the parallel track strategy. Section 28.2 points out that PT produces incorrect results if applied to plans involving stateful operators other than joins, although the authors claim in [ZRH04] that their migration strategies are generally applicable. In the remainder of this part, we assume plans to be composed of standard operators adhering to a global time-based window of size  $w$  as in [ZRH04]. We use the term *box* to refer to the implementation of a plan, i. e., the physical query plan actually executed.

## 28.1 Parallel Track Strategy

At migration start PT pauses the processing only briefly to plug in the new box. Thereafter it resumes the old box and runs both boxes in parallel while merging their results. Finally, when the states in the old box merely consist of elements that arrived after migration start, the migration is over and the old box can be removed safely. This implies that all elements stored in the state prior to the migration start have been purged from the state due to temporal expiration.

The following aspects guarantee the correctness of the PT strategy according to [ZRH04]:

1. Although all elements arriving during migration are processed by both plans, the combined output must not contain duplicates, i. e., results produced by both plans for the same snapshot. This is achieved by marking the elements with flags, *old* and *new*, which indicate whether an element arrived before or after the migration start time respectively. For combined results, e. g., join results, a *new* flag is assigned if all contributing elements were tagged with a *new* flag. To ensure correctness, PT discards all results of the old box tagged with a *new* flag as these are additionally produced by the new box.
2. In order to meet the ordering requirement for physical streams, the output of the two boxes has to be synchronized. Therefore, PT simply buffers the output of the new box during migration.

## 28.2 Problem Definition

The PT strategy proposed in [ZRH04] works well for join reordering but fails if other stateful operators are contained in query plans. Let us illustrate this deficiency with an example.

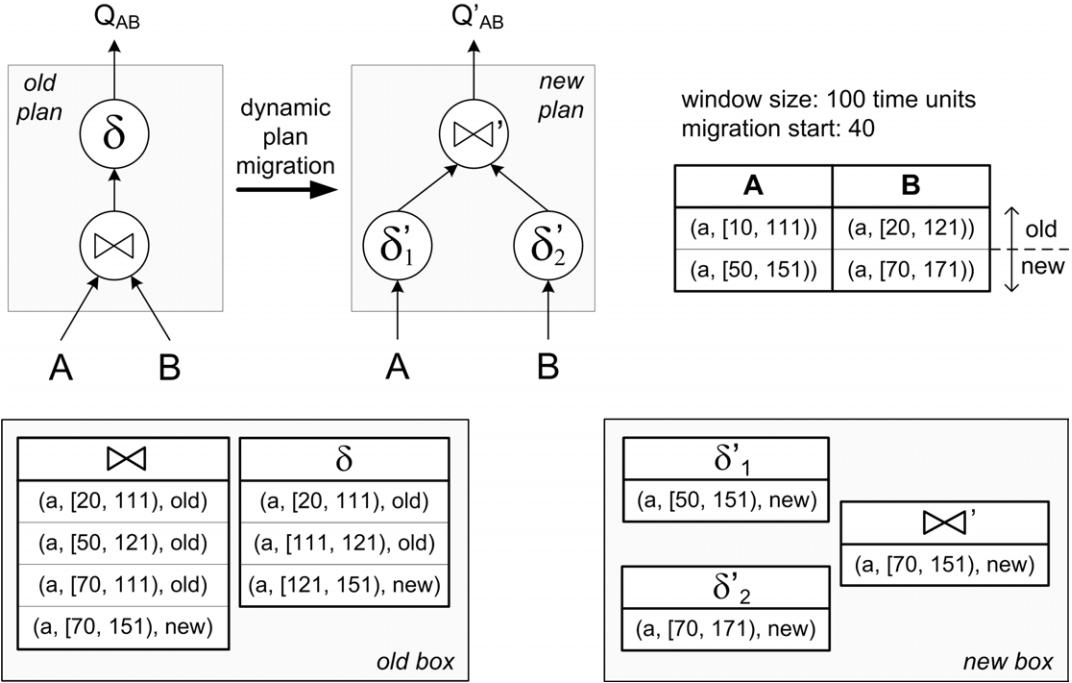


Figure 28.1: Plan migration with join and duplicate elimination

**Example 28.1.** Consider the plan migration shown in Figure 28.1. The old plan is composed of an equi-join ( $\bowtie$ ) and a duplicate elimination ( $\delta$ ). The new plan results from the old one by pushing down the duplicate elimination below the join. Recall that this is a standard transformation rule which holds in our stream algebra due to snapshot-equivalence. Besides the plans, Figure 28.1 shows the operator inputs and outputs.

We have two windowed input streams  $A$  and  $B$  delivering the elements listed in the upper right table. The table labeled with  $\bowtie$  depicts the correct results of the join inside the old box. The table labeled with  $\delta$  contains the correct results of the duplicate elimination of the old box. The tables  $\delta'_1$ ,  $\delta'_2$ , and  $\bowtie'$  correspond to the operator results of the new box. Correctness here refers to our continuous query semantics (see Chapter 8). Despite the use of half-open time intervals for denoting the validity of tuples, the example is implementation-independent because it only requires the join and duplicate elimination to be snapshot-reducible. A time interval just describes a contiguous set of time instants.

All input elements are considered to be valid for 100 time units due to the global window size. The migration start is at time instant 40. The element  $(a, [20, 121])$  from input  $B$ , which is marked as *old*, joins with the element  $(a, [50, 151])$  from input  $A$ . Hence, the join result  $(a, [50, 121])$  is marked as *old*. Since  $[50, 121]$  overlaps with the time interval of the duplicate elimination's result  $(a, [111, 121])$ , this is also marked as *old*. Therefore, it definitely belongs to the output of the old box. Unfortunately,  $(a, [111, 121])$  has a temporal overlap with  $(a, [70, 151])$  which is a result of the new box. Consequently, the complete output contains tuple  $a$  for the snapshots 111 to 120 twice. Thus, PT does not

produce a correct query answer.

The reason for this problem is that the validity of a tuple obtained from the output of the old box can refer to points in time that are beyond the plan migration start time. Tuples valid at these time instants may also be required by the new box. Since the output of both boxes is merged and the new box has no information about the old box, incorrect duplicates may occur in the output at those time instants. Our GenMig strategy overcomes this problem by defining a split time which is greater than any time instant referenced by the old box.

Notice that the problem of PT is not specific to a single transformation rule. We can therefore construct similar examples with plans containing other stateful operators, e. g., scalar aggregation or grouping with aggregation. See, for instance, the optimization illustrated in Figure 10.2. The PT strategy as proposed in [ZRH04] only generates correct results for SPJ queries. Although join reordering is an important transformation rule which is handled correctly by PT, PT fails for a variety of other transformation rules (see Chapter 10 and [SJS00, GJ01]).



# 29 A General Strategy for Plan Migration

In this chapter, we propose *GenMig*, our plan migration strategy that overcomes the imperfections of PT while maintaining its merits, namely, (i) a gradual migration from the old to the new plan, and (ii) a continual production of results during migration. While Section 29.1 explains GenMig from the logical perspective, Section 29.2 presents a plan migration algorithm for our time-interval approach. A performance analysis is given in Section 29.3, followed by two optimizations in Section 29.4. Section 29.5 outlines GenMig for the positive-negative approach and, thus, broadens the applicability of our migration strategy. Finally, Section 29.6 discusses how GenMig copes with plans over multiple time-based windows.

**Remark 29.1** (Moving States Strategy). We choose PT instead of MS as starting point for the development of our general migration strategy for the following reasons. MS requires a specific algorithm for each transformation rule to determine the state information of the new box. Therefore, it is not suited as a basis for a general migration strategy. Moreover, it is only efficient for small windows [ZRH04]. Another drawback is that MS generates no output during migration, an aspect colliding with our objectives for a good migration strategy formulated in Chapter 27.

## 29.1 Logical View

We first present the basic idea of GenMig from a purely logical and semantic perspective.

### 29.1.1 GenMig Strategy

Given two snapshot-equivalent query plans, the old and new one. The query optimizer determines a time instant  $T_{split} \in \mathbb{T}$  at which the time domain is divided into two partitions.

- For all time instants  $t < T_{split}$ , the query results are produced by the old plan.
- For all time instants  $t \geq T_{split}$ , the query results are produced by the new plan.

The union of the results from both plans produces the total query answer.  $T_{split}$  refers to the plan *migration end* because from this time instant onwards the new plan produces the output on its own. At this point in time, the old plan can be discarded. The *migration duration* is the period from migration start to  $T_{split}$ .

### 29.1.2 Correctness

Under the assumption that the old and new plan produce snapshot-equivalent outputs, the merged output of GenMig is *complete* because the output is computed for every single

time instant. Due to snapshot-equivalence it does not matter if the results for a snapshot are produced by either the old or new plan. Because we consider the entire time domain, no snapshot is lost. The problem with incorrect duplicates does not occur, because the results of both plans are disjoint in terms of snapshots.

## 29.2 Physical View

GenMig is clear and comprehensible at the logical level. However, at the physical level it is impractical to treat the query results for every single snapshot separately. For this reason, we propose the following algorithm implementing GenMig for our time-interval approach introduced in Chapter 11. Algorithm 25 is based on the same assumptions as PT. Hence, plans are restricted to having a global time-based window and to consisting of standard operators. Section 29.6 clarifies how we extend GenMig to support multiple windows with different sizes and further window types.

### 29.2.1 Implementation of GenMig for the Time-Interval Approach

---

#### Algorithm 25: GenMig Strategy

---

**Input** : physical streams  $I_1, \dots, I_n$ ; old plan with state information; new plan without state information; global window constraint  $w$   
**Output** : physical streams  $O_1, \dots, O_m$ ; new plan with state information

- 1 **foreach** input stream  $I_i, i \in \{1, \dots, n\}$  **do**
- 2   | Start monitoring the start timestamps;
- 3   | Keep the most recent start timestamps of  $I_i$  as  $t_{S_i}$ ;
- 4 Pause the execution of the old plan as soon as  $t_{S_i}$  has been set for each input;
- 5  $T_{split} \leftarrow \max\{t_{S_i} \mid i \in \{1, \dots, n\}\} + w + \varepsilon$ ;
- 6 Insert a *fork* operator downstream of each source in the old plan;
- 7 Insert a *fuse* operator at the top of both plans for each output stream;
- 8 Resume the execution of the old plan and start the execution of the new plan;
- 9 **while**  $\min\{t_{S_i} \mid i \in \{1, \dots, n\}\} < T_{split}$  **do**
- 10   | Continue the execution of both plans;
- 11 Signal the end of all input streams to the old plan;
- 12 Stop the execution of the old plan;
- 13 Pause the execution of the new plan;
- 14 Remove the old plan and also any *fuse* and *fork* operators;
- 15 Connect the inputs and outputs directly with the new plan;
- 16 Resume the execution of the new plan;

---

Algorithm 25 takes both boxes as inputs, the old and new one. The old box contains the currently executed physical plan connected with the corresponding input streams. The new box consists of the new, optimized query plan which is neither connected to the inputs nor initialized with regard to the operator states. In contrast to PT, GenMig does not start with the plan migration instantly. Instead, it starts monitoring the start timestamps provided by incoming stream elements.

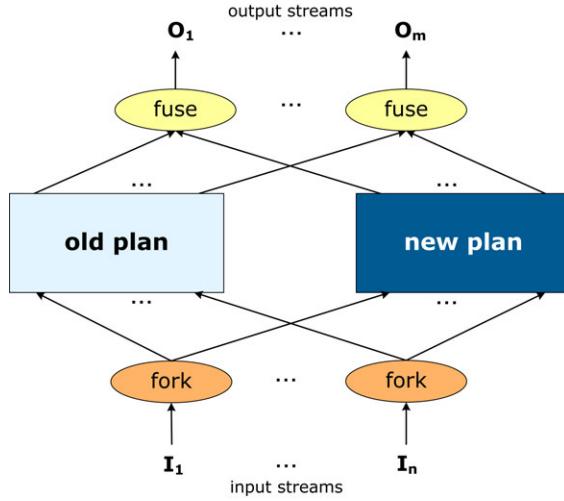


Figure 29.1: GenMig strategy

**Remark 29.2.** In contrast to [ZRH04] where a single migration start time is used, our approach maintains a migration start time for each input. This has the advantage that GenMig does not require the scheduling to adhere to the global temporal ordering by start timestamps, which conflicts with sophisticated scheduling strategies, e.g., [BBDM03, CCZ<sup>+</sup>03, JC04], in general.

After initialization, i.e., as soon as for each input stream a timestamp  $t_{S_i}$  has been determined, the old box is paused. The split time  $T_{split}$  is set to the maximum  $t_{S_i}$  plus the window size  $w$  plus  $\varepsilon$ . This choice of  $T_{split}$  ensures that the old box does not contain an element in any operator state that is valid at a time instant equal to or greater than  $T_{split}$ .

**Remark 29.3.** Without loss of generality, we assume  $\varepsilon$  to be chosen so that  $T_{split}$  neither occurs as start nor end timestamp in any input stream. This can for instance be achieved if  $T_{split}$  is expressed at a finer time granularity and  $\varepsilon$  refers to a granule associated with this granularity. From the implementation side, this assumption can be assured easily but details are omitted in the algorithms for clarity reasons.

In addition to setting the split time, two specific operators – *fork* and *fuse* – are inserted at the inputs and outputs respectively. Figure 29.1 illustrates this placement, while Algorithms 26 and 27 show the respective implementation. We reuse our notation defined in Section 11.4 for the specification of both algorithms. Recall that a plan can have multiple output streams due to subquery sharing. The stateless *fork* splits the time interval of an incoming element at  $T_{split}$  into two disjoint intervals. Tuple  $e$  tagged with the first interval is sent to the old box,  $e$  tagged with the second interval is sent to the new box.

*Fuse* inverts the effects of *fork*. It merges two equivalent tuples, each from one input, with consecutive time intervals. Contrary to *fork*, *fuse* is a stateful operator because it has to maintain internal data structures, e.g., hash maps, for the detection of equivalent tuples. Probing a hash map means traversing the elements in the corresponding bucket in start timestamp order  $\leq_{ts}$ . Furthermore, a priority queue ordered by start timestamps is

**Algorithm 26:** Fork

---

```

Input : physical stream  $I$ ; split time  $T_{split}$ 
Output : physical streams  $O_{old}, O_{new}$ 

1 foreach  $(e, [t_S, t_E]) \leftarrow I$  do
2   if  $t_S < T_{split}$  then
3     if  $t_E \leq T_{split}$  then  $(e, [t_S, t_E]) \hookrightarrow O_{old};$ 
4     else
5        $(e, [t_S, T_{split}]) \hookrightarrow O_{old};$ 
6        $(e, [T_{split}, t_E]) \hookrightarrow O_{new};$ 
7   else
8      $(e, [t_S, t_E]) \hookrightarrow O_{new};$ 

```

---

required to ensure the ordering property of the output stream. The *fork* and *fuse* operators differ from split and coalesce specified in Section 11.7 by having multiple output and input streams respectively. Split and coalesce are unary operators with a single output stream. Here, *fork* takes a single stream as input, splits incoming elements, but transfers the results to two different output streams. In contrast, *fuse* merges two input streams into a single output stream by coalescing results obtained from elements originally split by the *fork* operator.

The migration end is defined as the point in application time when the monitored start timestamps of all input streams are equal to or greater than  $T_{split}$ . At migration end the optimizer signals the end of all input streams to the old box in order to drain out intermediate elements. Thereafter the priority queue inside the *fuse* operator containing the new results can be flushed. Finally, the optimizer interrupts the processing briefly to discard the old box as well as the *fork* and *fuse* operators before it resumes the execution of the new plan.

### 29.2.2 Correctness

**Lemma 29.1.** *GenMig produces correct results and satisfies the temporal order requirement.*

*Proof.* (sketch)

1. No elements are lost during the insertion and removal of the fork and fuse operators because the processing of the boxes is suspended during these actions.
2. The fork operator guarantees that all elements valid at snapshots less than  $T_{split}$  are transferred to the old box, whereas elements with snapshots equal to or greater than  $T_{split}$  are delivered to the new box. This matches with the logical view of GenMig (see Section 29.1). Since no snapshot is lost under this partitioning and each box produces snapshot-equivalent results, the union of both plans provides the complete result.
3. PT uses a marking mechanism to detect duplicates, i. e., results at the same snapshot produced by both plans. We showed in Chapter 28 that this marking fails for

**Algorithm 27:** Fuse

---

**Input** : physical streams  $I_1$  (from old plan),  $I_2$  (from new plan); split time  $T_{split}$   
**Output** : physical stream  $O$

- 1 Let  $M_1, M_2$  be two empty hash maps that organize elements in their buckets according to  $\leq_{ts}$ ;
- 2 Let  $Q$  be an empty min-priority queue with priority  $\leq_{ts}$ ;
- 3  $i \in \{1, 2\}$ ;
- 4  $t_{out} \in T$ ;
- 5 **foreach**  $(e, [t_S, t_E]) \leftarrow I_i$  **do**
- 6      $t_{out} \leftarrow 0$ ;
- 7     **if**  $t_E < T_{split} \vee t_S > T_{split}$  **then**
- 8          $(e, [t_S, t_E]) \hookrightarrow Q$ ;
- 9     **else**
- 10         **if**  $i = 1$  **then**
- 11             **if**  $\exists(\hat{e}, [\hat{t}_S, \hat{t}_E]) \in M_2$  where  $e = \hat{e}$  **then**
- 12                  $Q.insert((e, [t_S, t_E]))$ ;
- 13                  $M_2.remove((\hat{e}, [\hat{t}_S, \hat{t}_E]))$ ;
- 14             **else**  $M_1.insert((e, [t_S, t_E]))$ ;
- 15         **if**  $i = 2$  **then**
- 16             **if**  $\exists(\hat{e}, [\hat{t}_S, \hat{t}_E]) \in M_1$  where  $e = \hat{e}$  **then**
- 17                  $Q.insert((e, [t_S, t_E]))$ ;
- 18                  $M_1.remove((\hat{e}, [\hat{t}_S, \hat{t}_E]))$ ;
- 19              $t_{out} \leftarrow \hat{t}_S$ ;
- 20         **else**  $M_2.insert((e, [t_S, t_E]))$ ;
- 21     **while**  $\neg Q.isEmpty()$  **do**
- 22         Element  $(\tilde{e}, [\tilde{t}_S, \tilde{t}_E]) \leftarrow Q.min()$ ;
- 23         **if**  $\tilde{t}_S \leq t_{out}$  **then**
- 24              $Q.extractMin() \hookrightarrow O$ ;
- 25         **else break**;
- 26 **if** migration is finished **then**
- 27     **while**  $\neg Q.isEmpty()$  **do**
- 28          $Q.extractMin() \hookrightarrow O$ ;

---

stateful operators other than joins. GenMig overcomes this problem because the fork operator ensures that the results of both boxes are disjoint in terms of snapshots due to the choice of  $T_{split}$ . As  $T_{split}$  is greater than any snapshot referenced in the old box, the corresponding snapshot-reducible operators in that box will never produce a result with a snapshot equal to or greater than  $T_{split}$ . Moreover, the new box will never generate results for snapshots less than  $T_{split}$ . As a result, GenMig inherently avoids the problem with duplicates addressed by PT.

4. The temporal stream order is preserved as (i) all operators inside a box produce a correctly ordered output stream, (ii) the fork operators are stateless and do not affect the ordering, and (iii) the fuse operator explicitly synchronizes the results provided by the old and new box using a priority queue.
5. The fuse operator combines the results of both plans. It does not have any semantic effects as coalescing preserves snapshot-equivalence [SJS00]. Because it merges stream elements with identical tuples and consecutive time intervals, it simply inverts the effects of the fork operator.
6. Due to the global window constraint  $w$ , the maximum time interval length for stream elements is limited to  $w$  time units (see window operator in Section 11.6.1). Snapshot-reducibility implies that the time intervals in the output stream of a standard operator can only have shorter time intervals. Hence, setting  $T_{split}$  to  $\max\{t_{S_i} \mid i \in \{1, \dots, n\}\} + w + \varepsilon$  ensures that  $T_{split}$  is greater than any time instant occurring in the old box.

□

### 29.3 Performance Analysis

Given the sufficient-system-resources assumption as in [ZRH04], the difference between system and application time becomes negligible. We can thus identify durations in application time with those in system time. The migration duration of GenMig is determined by  $T_{split} - \min\{t_{S_i}\}$  where  $t_{S_i}$  denotes the migration start time of input  $i$ . For negligible application time skew between streams and negligible latency in streams, the migration duration is approximately  $w$  time units due to the choice of  $T_{split}$ .

Compared to PT, GenMig has the following advantages:

- For join trees with more than one join, the required migration time is only  $w$  time units instead of  $2w$ . GenMig requires at most  $w$  time units because all elements in the old plan have become outdated at  $T_{split}$ . GenMig does not need to wait until all *old* elements were purged from operator states in the old box as done for PT. Consequently, the allocated memory for the old box can be released earlier.
- GenMig profits from reduced processing costs because it does not require any mechanisms to detect duplicates at the output of the old box.

- GenMig does not need to buffer the entire output of the new box during migration for ordering purposes. All results produced by the new box during migration are coalesced and emitted. The size of the priority queue and hash maps inside the fuse operator is dominated by the application time skew between the input streams. Heartbeats [SW04] and sophisticated scheduling strategies can be used to minimize these effects and, thus, the memory allocation of the fuse operator.
- According to [ZRH04], the migration for PT is finished if all *old* elements have been removed from the old box. For join trees with more than one join, this happens after  $2w$  time units. Interestingly, the old box only produces output during the first  $w$  time units. The other  $w$  time units are used to purge all *old* elements from the operator states. For any snapshot-reducible query plan, there will be no output during this timespan. As a consequence, PT has the following output rate characteristics. For the first  $w$  time units, the output rate corresponds to the output rate of the old box. The next  $w$  time units there is no output at all. At the migration end there is a burst because the buffer on top of the new box is flushed. In contrast, GenMig switches smoothly from the output rate of the old box to the one of the new box at migration end ( $T_{split}$ ).

## 29.4 Optimizations

In the following, we propose two optimizations for improving our GenMig strategy.

### 29.4.1 Reference Point Method

The *reference point method* [See91, BS96] is a common technique for index structures to prevent duplicates in the output. We can use this method as an optimization of GenMig if the following modifications are performed. The fork operator has to be modified to send elements to the old plan without splitting, i. e., with the full time intervals. The fuse operator is removed and replaced by a simple filter followed by a union. While the filter is placed on top of the new box and drops all elements with a start timestamp equal to  $T_{split}$ , the union merges the output of the old box and the filter. We treat the start timestamp attached to results of the new box as *reference point*. This reference point is compared with  $T_{split}$ . If it is greater than  $T_{split}$ , the element is no duplicate and send to the output.

The reference point method substitutes the stateful fuse operator with stateless operators. Therefore, it saves memory as well as processing costs. With regard to correctness, both boxes ensure the output stream order requirement. Because we use the start timestamp as reference point, all results from the old box have a smaller start timestamp than those from the new box. Hence, it is sufficient to first output the results of the old box and afterwards those of the new box. Under the assumptions of a global temporal scheduling as in [ZRH04], no buffer is needed to synchronize the output of both boxes. Notice that all additional operators required for GenMig with reference point optimization (fork, union, and filter) have constant processing costs per element.

### 29.4.2 Shortening Migration Duration

The migration duration could be shortened if  $T_{split}$  is set to the maximum end timestamp occurring inside the old box plus  $\varepsilon$ . While this setting still satisfies the correctness condition that  $T_{split}$  has to be greater than any time instant occurring in the old box, such an optimization requires the monitoring of the end timestamps as well. If a DSMS provides this information as some kind of metadata [CKS07], it could be used effectively to shorten the migration duration and consequently gain from savings in system resource consumption. This optimization is particularly effective if the plan, which is optimized, is not located closely downstream of the window operators. In this case, it is likely that the time intervals are significantly shorter than the window size.

## 29.5 Implementation of GenMig for the Positive-Negative Approach

The GenMig algorithm can easily be transferred to PNA [GHM<sup>+</sup>07, ABW06] (see Section 14.1). Instead of monitoring the start timestamps, the timestamps attached to positive tuples are monitored.  $T_{split}$  is set as proposed in Algorithm 25. The fork operator sends all incoming positive elements along with their corresponding negative elements to the new box and additionally to the old box if their timestamp is less than  $T_{split}$ . Using the timestamp of an element, independent of its sign, as reference point, we accept results from the old box if their timestamp is less than  $T_{split}$ , and from the new box if it is greater than  $T_{split}$ . Since the results generated by both plans are ordered properly, it is sufficient to first output the results of the old box and afterwards those from the new box. The migration end is reached if the timestamps of all input streams are greater than  $T_{split}$ .

## 29.6 Extensions

In compliance with [ZRH04], we assume that plans consist of standard operators to which a global time-based window is applied. The correctness proof for GenMig is based on the invariant that  $T_{split}$  is defined greater than any time instant occurring in the old box, i. e., that no element exists in any operator state of the old box being valid at  $T_{split}$  or later. If plans are restricted to snapshot-reducible operators and a global window size, we can satisfy this property by setting  $T_{split}$  to  $\max\{t_{S_i} \mid i \in \{1, \dots, n\}\} + w + \varepsilon$ . However, the appropriate definition of the split time is more complicated whenever plans refer to multiple window operators, which can be of different types and sizes. For instance, the validity of tuples entering the old box may depend on upstream count-based or partitioned windows. In these cases, it is usually not possible to specify an appropriate substitute for the global window size  $w$ , i. e., an upper bound for the tuple validity.

We determine the split time for the general case as follows. Prior to the start of the migration task, we not only monitor the start timestamp but also the end timestamp of elements entering the old box. The maximum end timestamp for each stream is stored as metadata and updated continuously (see [CKS07] for details on metadata management). Let  $t_{E_i}$ ,  $i \in \{1, \dots, n\}$ , be the observed maximum end timestamp for each input stream

of the old box respectively. Setting  $T_{split}$  to  $\max\{t_{E_i} \mid i \in \{1, \dots, n\}\} + \varepsilon$  is sufficient for any algebraic optimization in which the old box is composed of standard operators, independent of upstream operators. In this case, the plan associated with the old box is snapshot-reducible. If the old box corresponds to a non-snapshot-reducible subplan, only the window transformation rules are applicable for optimization (see Section 10.4). As window operators do not commute with stateful operators, plan migration reduces to the following actions: (i) pausing the execution of the subplan, (ii) processing all intermediate elements (in the case of inter-operator queues), (iii) swapping the operators according to the transformation rule, and (iv) resuming the execution. In other words, we do not use GenMig for these specific cases as it is more efficient to perform the aforementioned steps instead.

In the majority of cases, the query optimizer will pick out a snapshot-reducible subplan as those offer the most optimization potential. Furthermore, it is important that the optimizer aims at minimizing the selected plans so that these plans only consists of the relevant operators to which the transformation rules refer.



## 30 Experimental Evaluation

Although the primary focus of this work is on the semantics, generality, and correctness of GenMig, we conducted a set of experiments that compare GenMig with PT for join reordering. We observed that even in this case, the only transformation in which PT is applicable to stateful operators, GenMig is at least as efficient as PT. In addition, we validated GenMig for a variety of transformation rules beyond join reordering. However, those experiments only show the correctness of GenMig and point out the potential of dynamic query optimization. As these do not contribute further insights into our approach, we omitted them.

We implemented PT and GenMig in our PIPES framework [KS04, CHK<sup>+</sup>06]. Our hardware was a PC with an Intel Pentium 4 processor (2.8 GHz) and 1 GB of RAM running Windows XP Professional. To guarantee a fair comparison with the original PT implementation in [ZRH04], scheduling was based on a single thread following a strategy that caused elements to be processed according to the global temporal order defined by their timestamps. Since a comparison with PT is only possible for joins, we run 4-way nested-loops joins as done in [ZRH04].

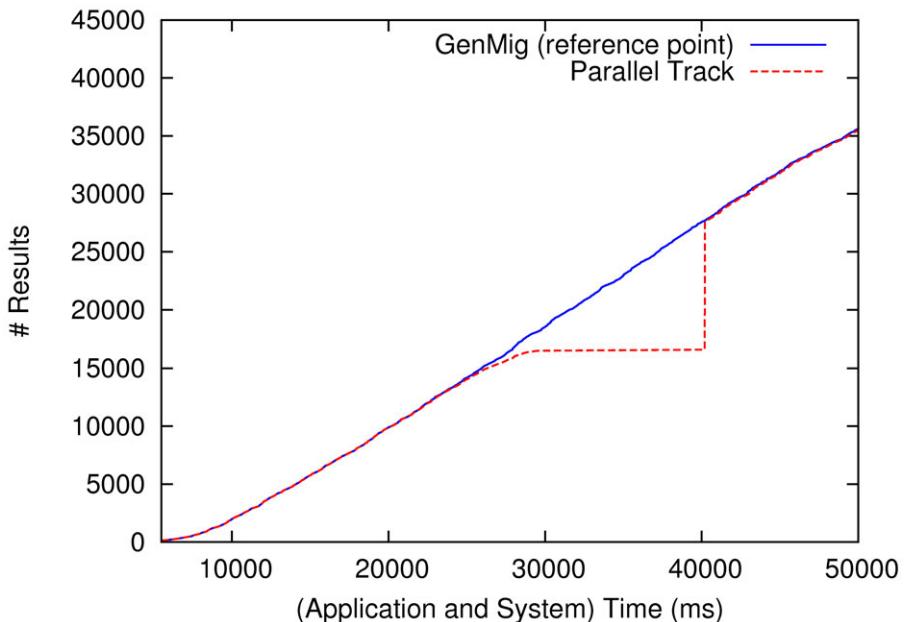


Figure 30.1: Output stream characteristics of Parallel Track and GenMig

In our first experiment, we ensured the sufficient-system-resources assumption which means that query execution can keep up with stream rates. Each input stream delivered

5000 random numbers at a rate of 100 elements per second. The random numbers were distributed uniformly between 0 and 500 for streams  $A$  and  $B$ , and between 0 and 1000 for streams  $C$  and  $D$ . We performed time-based sliding window equi-joins with a global window size of 10 seconds. The old plan was set to the left-deep join tree  $((A \bowtie B) \bowtie C) \bowtie D$  which was rather inefficient due to the huge intermediate result produced by  $A \bowtie B$ . The goal of the dynamic plan migration was to switch to the more efficient right-deep join tree  $A \bowtie (B \bowtie (C \bowtie D))$ . The migration started after 20 seconds.

GenMig finished the migration  $w$  time units (10 seconds) after migration start as expected. In contrast, PT required  $2w$  time units due to the purging of all *old* elements from the old box. This complies with the analysis given in [ZRH04]. Figure 30.1 shows that during migration the output rate of PT decreased because the results of the new box were buffered. After 30 seconds the output rate reached zero for a duration of 10 seconds. During this period the eviction of *old* elements took place. At the migration end for PT after 40 seconds (migration start +  $2 \cdot$  window size =  $20 + 2 \cdot 10 = 40$ ), the results of the new box which had been buffered during migration were immediately released. This caused the significant burst in the output rate. Such a burst may lead to a temporary system overload and should be avoided whenever possible. In contrast to PT, GenMig produced results with a smooth output rate during migration (see Figure 30.1).

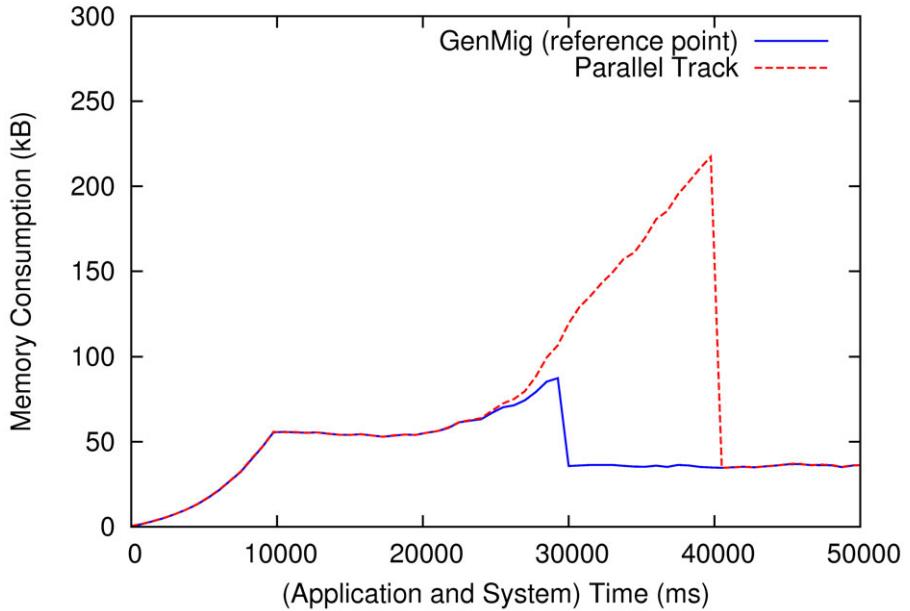


Figure 30.2: Memory usage of Parallel Track and GenMig

Figure 30.2 shows the memory usage of GenMig and PT during the experiment. For the sake of better comparability, we measured the memory allocated for the tuples and omitted the overhead of timestamps because GenMig requires two timestamps per element (time interval), whereas PT needs up to four. Note that the memory consumption can only be different during migration. Figure 30.2 demonstrates that PT permanently requires more memory than GenMig during this period. Overall, the system has an increased

---

memory usage during plan migration but profits from the reduced memory usage of the new plan afterwards. However, the temporary increase in memory consumption is less for GenMig.

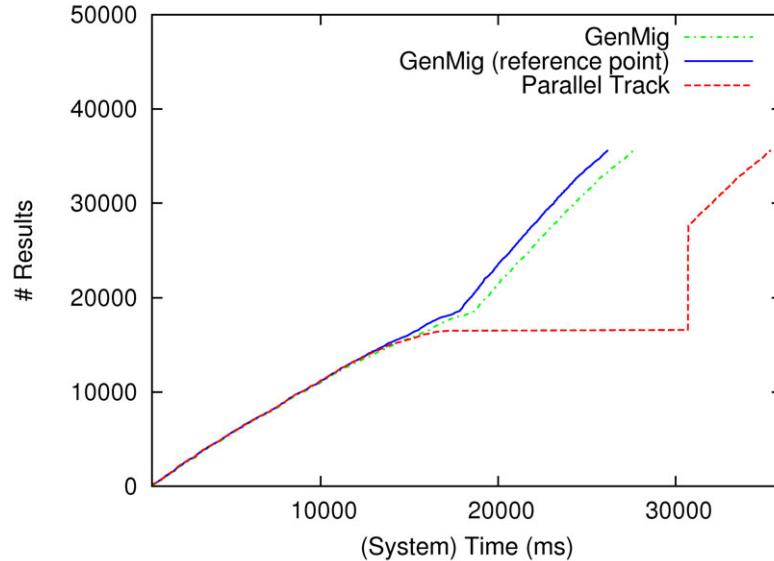


Figure 30.3: Performance comparison of Parallel Track, GenMig, and GenMig with reference point optimization

Our second experiment was aimed at comparing the total system load of PT, standard GenMig, and GenMig with reference point optimization. While using the same input streams as in our first experiment, we processed the streams in our second experiment as fast as possible, i. e., we no longer synchronized application and system time. Measuring the runtime under system saturation is a widely accepted approach to determine the efficiency of stream join algorithms [GÖ03a] (see also Section 14.3). Furthermore, we simulated a more expensive join predicate to emphasize the effects of complex join computations. Figure 30.3 depicts our performance measurements. At the beginning, the slope of the curves, which corresponds to the output rate, is less steep than at the end, because the left-deep join plan is not as efficient as the right-deep plan. During migration the slope reaches its minimum as both plans run in parallel. The total runtimes demonstrate that GenMig is superior to PT. Moreover, the reference point optimization improves the standard variant of GenMig slightly as it avoids the CPU costs caused by the fuse operator.

To sum up the lessons learned from our experiments, GenMig is not only more general than PT but also more efficient for the case of join reordering where both strategies are applicable. GenMig allocates less additional memory than PT during plan migration. Moreover, GenMig satisfies any of the objectives for a suitable plan migration strategy formulated in Chapter 27.



## 31 Related Work

In recent years adaptivity issues for query processing and optimization have attracted a lot of research attention. The following discussion is limited to work related to stream processing. The interested reader is referred to [BB05] for a survey beyond streams.

Our work directly refers to the dynamic plan migration strategies proposed in [ZRH04] as it generalizes PT towards arbitrary continuous query plans. To the best of our knowledge, the strategies published in [ZRH04] are the only methods for dynamic plan migration in DSMSSs.

There are several papers on different topics of runtime optimizations for DSMSSs but these do not tackle plan migration issues explicitly. In [BMWM05] the problem of executing continuous multiway join queries is addressed for streaming environments with changing data characteristics. GenMig does not aim at optimizing multiway join performance by materializing intermediate join views. In contrast, it is designed as a more general solution and treats join reordering as one possible transformation rule. Proactive re-optimization [BBD05] is a novel optimization technique used in stream processing whereby the optimizer selects the initial query plan, while being aware that this plan is likely to see further re-optimizations due to the uncertainty in estimating the underlying statistics. However, the choice of a suitable plan is not the focus of GenMig. It rather describes how to migrate from one plan to another snapshot-equivalent plan.

Eddies [AH00, DH04] perform a very flexible kind of adaptive runtime optimization for continuous queries. Unlike traditional approaches in which elements are processed according to a fixed query plan, eddies are central system components determining an individual query plan for every element by routing it through the connected operators. PT and GenMig are by far not as flexible as eddies, but the per-element routing is expensive and only profitable in highly dynamic environments. Moreover, eddies are limited to queries with selection, projection, and join, whereas GenMig can handle a broader set of operators including stream variants for all operators of the extended relational algebra.

Cross-network optimization issues for continuous queries are discussed in [AAB<sup>+</sup>05] for the Borealis stream processing engine, but not at a semantic level with concrete techniques for plan migration as presented in this paper. How GenMig can be adapted to a distributed environment remains as an open issue for future work.



## 32 Conclusions

In this part we first identified some shortcomings of the parallel track strategy, an existing solution for the dynamic plan migration problem in DSMSs. We showed that this strategy fails to cope with plans involving stateful operators other than joins. Thereafter we proposed a novel strategy for dynamic plan migration, called *GenMig*, which enables a DSMS to optimize arbitrary continuous queries at runtime as long as snapshot-equivalence is preserved. Our analysis and performance comparison showed that GenMig with its optimizations is at least as efficient as the parallel track strategy, while GenMig overcomes the limitations of PT. Due to the underlying semantics, the whole set of conventional transformation rules can be applied for optimization purposes. Moreover, GenMig does not require any specific knowledge about operator states and implementations because it treats the old and new plans as black boxes. Due to its generality and ease of use, GenMig is likely to be integrated into existing and future DSMSs as a promising strategy for the dynamic optimization of continuous queries.



## **Part V**

# **Summary of Conclusions and Future Research Directions**



## 33 Summary of Conclusions

The main theme of the dissertation is to study data management issues for an emerging class of applications that require the processing of *continuous queries* over a set of *data streams*. We focus on the important and common type of *sliding window queries* to restrict resource requirements of operations over potentially infinite streams.

In Part I, we introduced the data stream model and explained why traditional DBMSs are deemed inadequate to support high-volume, low-latency stream processing applications. Thereafter, we outlined several research challenges crucial to building a fully fledged DSMS and pointed out the specific contributions of this thesis.

In Part II, we developed a formal foundation with clear semantics for specifying continuous queries over data streams. At first, we introduced our *declarative query language* that stays close to SQL:2003 standards and allows users to express complex application logic conveniently, without requiring the use of custom code. Afterwards, we precisely defined query semantics by means of our *logical stream algebra*. Although our algebra incorporates the notion of time and supports windowing functionality, it preserves the well understood semantics of the relational algebra to the extent possible due to the snapshot-reducibility of standard operators. In comparison to the abstract semantics behind CQL, our algebra reflects the temporal properties of stream operators directly. Multiset-snapshot-equivalence enabled us to carry over the extensive set of conventional and temporal transformation rules defined in [SJS01] from temporal databases to stream processing. As a consequence, our approach provides a solid and powerful basis for *query optimization*. Next, we introduced our *physical algebra* modeling data streams as sequences of tuples tagged with time intervals. To the best of our knowledge, our time-interval approach is unique in the stream community. Our window operators set the validity of tuples according to the type and size of the window. Placing window operators upstream of stateful operators in a query plan avoids blocking and restricts resource requirements because tuples expire from the state. For every operator in our logical algebra, we presented a physical analog in the form of an online algorithm. We discussed the instantiation and parameterization of the abstract data type SweepArea, our generic sweepline data structure for efficient operator state maintenance. We defined a novel split operator to handle windowed subqueries and reported on several *physical optimizations* for which we exploited expiration patterns, coalescing, and priority queue placement. After that, we outlined the architecture of PIPES and explained how PIPES copes with stream imperfections. The comparison between our time-interval approach and the popular positive-negative approach revealed that TIA is superior to PNA for the majority of time-based sliding window queries. Because PNA doubles stream rates, it suffers from increased processing costs and higher memory consumption. Finally, we proved that our query language has at least the same *expressive power* as CQL.

In Part III, we introduced two novel *adaptation techniques* to control the resource

consumption of continuous queries during execution based on adjustments to window and granule sizes. Extensions to our query language allow the user to specify QoS ranges for window and granule sizes when issuing a continuous query and open up effective degrees of freedom for adaptive resource management. In contrast to random load shedding, the main advantage of these methods is that the system can give *reliable guarantees* on query answers. Because query results are exact with regard to the selected QoS settings, adaptations do not collide with query optimization at runtime and consequently increase the flexibility and scalability of a DSMS. Furthermore, we introduced an *extensive cost model* to quantify the resource usage of continuous queries. Adaptive runtime components such as the resource manager or query optimizer utilize such a cost model to estimate the impact of their actions on resource consumption. Our extensive experimental studies on real-world and synthetic data streams prove the accuracy of our cost model and the effectiveness of our adaptation techniques, even for larger settings where thousands of continuous queries were processed concurrently over several hours.

In Part IV, we studied the problem of *dynamic query optimization*. We proposed *GenMig*, a novel *plan migration strategy*, that performs a gradual transition from the old to the new plan at runtime. Because GenMig treats the old and new plans as snapshot-equivalent black boxes, the complete set of transformation rules holding in our stream algebra becomes applicable for optimization purposes. This is a major advance over the existing PT technique, which is limited in its scope to plans with stateless operators and joins. Our performance analysis and experimental results show that GenMig even outperforms PT in the case of join reordering. GenMig not only reduces the migration duration and the memory overhead, but also exhibits a smoother output rate by avoiding bursts at migration end. Because of its merits, in particular the ability to perform complex transformations in a convenient fashion, GenMig serves as a vital component for query optimizers in DSMSs.

Finally, it is worth noting that we invested significant time and effort in the development of our stream processing infrastructure PIPES, which has been released as open-source software. This thesis describes the core principles and sophisticated technology behind PIPES. We have successfully applied PIPES to various application domains including traffic management, online auctions, sleep medicine, and factory automation. Because of the flexible and extensible library design and component frameworks, developers can enhance existing functionality easily or tailor abstract pre-implementations to their needs. As a consequence, our software infrastructure represents an excellent platform for research that enables and facilitates the exchange and fair comparison of query processing functionality inside a uniform runtime environment.

Overall, this thesis tackles one of the most challenging issues in continuous query processing, namely, the development of a coherent and efficient stream processing engine whose powerful processing primitives derive from suitable extensions of the relational algebra. As the data stream model finds widespread use in modern applications, such stream processing engines are expected to gain considerable popularity.

# 34 Future Research Directions

We already devoted sections to opening up interesting avenues for future work in the respective parts of this thesis. Section 15.2 discusses possible extensions of our query language and stream algebra, while Section 23.4 points out improvements to our cost model and resource adaptation strategies. In the following, we suggest some broader research directions for future work that would enhance our approach with additional functionality to cope with the diverse requirements of real-world streaming applications. We are convinced that future approaches can reap the benefits of the semantic foundations and core functionality developed in this thesis.

## **Application-Specific Extensions**

The heterogeneity of the application landscape for data stream management makes it impractical to develop a general-purpose DSMS providing an optimal solution for the various applications. For this reason, we designed PIPES as a software library whose functionality can be tailored to the particular needs of an application. In order to facilitate usage and improve efficiency, we suggest developing pre-configured software packages by augmenting the core of PIPES with application-specific functionality. For example, consider a class of applications that monitor moving objects. Using the standard functionality of PIPES may be suboptimal because the algorithms and also the query language are not aware of the spatial properties of stream elements. Adding spatial functionality such as spatial predicates and functions to the query language would facilitate the specification of application logic. At implementation level, algorithms may exploit the spatial information of stream elements to optimize purging and probing. Besides spatial support, which is required in many streaming applications [MA05], software packages could provide advanced functionality for sensor data cleaning [JAF<sup>+</sup>06a, JAF<sup>+</sup>06b] and data mining [LTWZ05, BTW<sup>+</sup>06].

## **Historical Queries**

While this thesis focuses on queries being evaluated continuously over newly arrived data, some streaming applications require ad-hoc queries referencing the past. Since the data stream computation model dictates that past stream elements cannot be revisited without explicit storage, efficient data storage and access methods, in particular for external memory, have to be developed. These archives should typically maintain a sliding window over a data stream, which can be the output of a continuous query. In addition to historical queries, these archives can also be exploited for optimization purposes by materializing shared sub-expressions of similar queries. Initial work on materializing query results employs B-trees [ACC<sup>+</sup>03] or doubly partitioned indices [GPÖ06] to organize historical stream elements. To reduce costs for index maintenance and ensure efficient probing, index structures should organize stream elements not only by search key but also according to time, similar to our SweepAreas. For this reason, we suggest investigating

the suitability of multidimensional index structures [BKSS90] and multiversion index structures [BGO<sup>+</sup>96] in this context.

### Adaptive Runtime Environment

The runtime components of PIPES can be extended in various directions. With regard to *scheduling*, novel strategies should be examined that reduce application-time skew between the streams in a plan. Balancing the time advancement between input streams can decrease resource usage of operators with multiple inputs such as the union or join in general, because the size of the internal priority queue and SweepAreas is likely to become smaller. For DSMSs performing pull-based processing, such a strategy can shrink the sizes of inter-operator queues. In contrast to many DSMS prototypes, PIPES already supports multi-threaded query execution. However, we need more sophisticated strategies that automatically control the degree of concurrency. This problem will increase in importance because of the upcoming multi-core computer architectures. An orthogonal problem is the *allocation of memory* to query plans with the aim to maximize overall quality of service, especially under changing system conditions. Moreover, continuous queries present better opportunities for *resource sharing* than traditional one-time queries because they hold on to resources longer. Thus, the query optimizer should identify and exploit resource sharing opportunities [AW04b, DGH<sup>+</sup>06]. In addition, dynamic query optimization necessitates strategies based on runtime statistics that determine (i) which subplans to pick out for re-optimization and (ii) when to initiate re-optimization. Rather than using a cost model to estimate resource allocation of a new plan, it may be beneficial to pursue a *simulation-based* approach instead. The latter will measure the resource consumption for the plan in a secondary system in which the input streams are simulated by means of a statistical model. As the simulation can run at much faster rates, the optimizer will be able to examine and assess new query plans in a shorter period of time, without running the risk of adversely affecting the efficiency of the original plan by continuous updates to runtime statistics.

### High-Availability

Most stream processing applications, e.g., medical monitoring or financial services, require DSMSs to be highly available. Therefore, a DSMS should be resilient against failures. Recovery mechanisms based on a single server and log files would incur unacceptable overhead due to the huge amount of transient data being processed as well as the significant disruption time. As a result, a backup server is required that takes over the operation of the failed server. To ensure consistent query results, both systems must be synchronized. Different recovery techniques using redundant processing, checkpointing, and remote logging, have been proposed in [HBR<sup>+</sup>05, BSS05], each with its own recovery guarantees. It needs to be identified how these recovery techniques can be adapted to our query semantics and processing methodology. For planned transitions from one server to another, our plan migration strategy might turn out to be useful.

### Distributed Data Stream Management

Because active data sources are often located at remote sites, for instance in sensor networks, many stream-oriented applications are inherently distributed. As a consequence, extending DSMSs to distributed environments has attracted recent research

---

attention [AAB<sup>+</sup>05, MFHH05, KSKR05]. The underlying hardware ranges from resource-constrained devices such as tiny sensors to clusters of resource-rich servers. *Distributed* DSMSs can gain in both scalability and efficiency if (i) nodes collectively process data streams and (ii) computation takes place at nodes located closer to the data sources, in the case of geographically distributed nodes. The latter would reduce communication costs significantly. Distributed data stream management opens up several research challenges, including the questions of how to distribute query plans and how to balance load. Cost models have to be extended to incorporate new metrics such as network bandwidth and sensor power consumption. Furthermore, a distributed DSMS should support optimizations at different levels, ranging from single-node optimizations to global optimizations. Because queries are evaluated continuously and nodes interact with each other, the distribution and processing models have to dynamically reconfigure as network conditions change. In addition to these optimization capabilities, distributed DSMSs require fault tolerance against node and network failures. Ideally, the distribution should be hidden from users and applications.



# Bibliography

- [AAB<sup>+</sup>05] Daniel J. Abadi, Yanif Ahmad, Magdalena Balazinska, Ugur Cetintemel, Mitch Cherniack, Jeong-Hyon Hwang, Wolfgang Lindner, Anurag Maskey, Alex Rasin, Esther Ryvkina, Nesime Tatbul, Ying Xing, and Stanley B. Zdonik. The Design of the Borealis Stream Processing Engine. In *Proc. of the Conf. on Innovative Data Systems Research (CIDR)*, pages 277–289, 2005.
- [ABB<sup>+</sup>02] Arvind Arasu, Brian Babcock, Shivnath Babu, Jon McAlister, and Jennifer Widom. Characterizing Memory Requirements for Queries over Continuous Data Streams. In *Proc. of the ACM SIGMOD Int. Conf. on Management of Data*, pages 221–232, 2002.
- [ABB<sup>+</sup>04] Arvind Arasu, Brian Babcock, Shivnath Babu, Jon McAlister, and Jennifer Widom. Characterizing memory requirements for queries over continuous data streams. *ACM Transactions on Database Systems (TODS)*, 29(1):162–194, 2004.
- [ABW03] Arvind Arasu, Shivnath Babu, and Jennifer Widom. An Abstract Semantics and Concrete Language for Continuous Queries over Streams and Relations. In *Proc. of the Intl. Conf. on Data Base Programming Languages (DBPL)*, pages 1–19, 2003.
- [ABW06] Arvind Arasu, Shivnath Babu, and Jennifer Widom. The CQL Continuous Query Language: Semantic Foundations and Query Execution. *VLDB Journal*, 15(2):121–142, 2006.
- [ACC<sup>+</sup>03] Daniel J. Abadi, Don Carney, Ugur Cetintemel, Mitch Cherniack, Christian Convey, Sangdon Lee, Michael Stonebraker, Nesime Tatbul, and Stan Zdonik. Aurora: A New Model and Architecture for Data Stream Management. *VLDB Journal*, 12(2):120–139, 2003.
- [ACG<sup>+</sup>04] Arvind Arasu, Mitch Cherniack, Eduardo F. Galvez, David Maier, Anurag Maskey, Esther Ryvkina, Michael Stonebraker, and Richard Tibbetts. Linear Road: A Stream Data Management Benchmark. In *Proc. of the Int. Conf. on Very Large Databases (VLDB)*, pages 480–491, 2004.
- [AF00] Mehmet Altinel and Michael J. Franklin. Efficient Filtering of XML Documents for Selective Dissemination of Information. In *Proc. of the Int. Conf. on Very Large Databases (VLDB)*, pages 53–64, 2000.
- [AGPR99] Swarup Acharya, Phillip B. Gibbons, Viswanath Poosala, and Sridhar Ramaswamy. Join Synopses for Approximate Query Answering. In *Proc. of the ACM SIGMOD Int. Conf. on Management of Data*, pages 275–286, 1999.

## Bibliography

---

- [AH00] Ron Avnur and Joseph M. Hellerstein. Eddies: Continuously Adaptive Query Processing. In *Proc. of the ACM SIGMOD Int. Conf. on Management of Data*, pages 261–272, 2000.
- [AHS06] Karl Aberer, Manfred Hauswirth, and Ali Salehi. A Middleware for Fast and Flexible Sensor Network Deployment. In *Proc. of the Int. Conf. on Very Large Databases (VLDB)*, pages 1199–1202, 2006.
- [Alb91] Joseph Albert. Algebraic Properties of Bag Data Types. In *Proc. of the Int. Conf. on Very Large Databases (VLDB)*, pages 211–219, 1991.
- [AMS96] Noga Alon, Yossi Matias, and Mario Szegedy. The Space Complexity of Approximating the Frequency Moments. In *Proc. of the ACM Symposium on Theory of Computing*, pages 20–29, 1996.
- [AN04] Ahmed Ayad and Jeffrey F. Naughton. Static Optimization of Conjunctive Queries with Sliding Windows Over Infinite Streams. In *Proc. of the ACM SIGMOD Int. Conf. on Management of Data*, pages 419–430, 2004.
- [ANWS06] Ahmed Ayad, Jeffrey F. Naughton, Stephen Wright, and Utkarsh Srivastava. Approximating Streaming Window Joins Under CPU Limitations. In *Proc. of the IEEE Int. Conf. on Data Engineering (ICDE)*, page 142, 2006.
- [APR<sup>+</sup>98] Lars Arge, Octavian Procopiuc, Sridhar Ramaswamy, Torsten Suel, and Jeffrey Scott Vitter. Scalable Sweeping-Based Spatial Join. In *Proc. of the Int. Conf. on Very Large Databases (VLDB)*, pages 570–581, 1998.
- [Ara06] Arvind Arasu. *Continuous Queries over Data Streams*. PhD thesis, Stanford University, 2006.
- [AW04a] Arvind Arasu and Jennifer Widom. A Denotational Semantics for Continuous Queries over Streams and Relations. *SIGMOD Record*, 33(3):6–12, 2004.
- [AW04b] Arvind Arasu and Jennifer Widom. Resource Sharing in Continuous Sliding-Window Aggregates. In *Proc. of the Int. Conf. on Very Large Databases (VLDB)*, pages 336–347, 2004.
- [BB05] Shivnath Babu and Pedro Bizarro. Adaptive Query Processing in the Looking Glass. In *Proc. of the Conf. on Innovative Data Systems Research (CIDR)*, pages 238 – 249, 2005.
- [BBD<sup>+</sup>01] Jochen van den Bercken, Björn Blohsfeld, Jens-Peter Dittrich, Jürgen Krämer, Tobias Schäfer, Martin Schneider, and Bernhard Seeger. XXL - A Library Approach to Supporting Efficient Implementations of Advanced Database Queries. In *Proc. of the Int. Conf. on Very Large Databases (VLDB)*, pages 39–48, 2001.
- [BBD<sup>+</sup>02] Brian Babcock, Shivnath Babu, Mayur Datar, Rajeev Motwani, and Jennifer Widom. Models and Issues in Data Stream Systems. In *Symp. on Principles of Database Systems (PODS)*, pages 1–16, 2002.

- [BBD<sup>+</sup>04] Brian Babcock, Shivnath Babu, Mayur Datar, Rajeev Motwani, and Dilys Thomas. Operator Scheduling in Data Stream Systems. *VLDB Journal*, 13(4):333–353, 2004.
- [BBD05] Shivnath Babu, Pedro Bizarro, and David DeWitt. Proactive Re-Optimization. In *Proc. of the ACM SIGMOD Int. Conf. on Management of Data*, pages 107–118, 2005.
- [BBDM03] Brian Babcock, Shivnath Babu, Mayur Datar, and Rajeev Motwani. Chain: Operator Scheduling for Memory Minimization in Data Stream Systems. In *Proc. of the ACM SIGMOD Int. Conf. on Management of Data*, pages 253–264, 2003.
- [BBJ98] Michael H. Böhlen, Renato Busatto, and Christian S. Jensen. Point-Versus Interval-Based Temporal Data Models. In *Proc. of the IEEE Int. Conf. on Data Engineering (ICDE)*, pages 192–200, 1998.
- [BDE<sup>+</sup>97] Claudio Bettini, Curtis E. Dyreson, William S. Evans, Richard T. Snodgrass, and Xiaoyang Sean Wang. A Glossary of Time Granularity Concepts. In *Temporal Databases: Research and Practice*, pages 406–413. Lecture Notes in Computer Science, 1997.
- [BDM04] Brian Babcock, Mayur Datar, and Rajeev Motwani. Load shedding for Aggregation Queries over Data Streams. In *Proc. of the IEEE Int. Conf. on Data Engineering (ICDE)*, pages 350–361, 2004.
- [BGO<sup>+</sup>96] Bruno Becker, Stephan Gschwind, Thomas Ohler, Bernhard Seeger, and Peter Widmayer. An Asymptotically Optimal Multiversion B-Tree. *VLDB Journal*, 5(4):264–275, 1996.
- [BGS01] Philippe Bonnet, Johannes Gehrke, and Praveen Seshadri. Towards Sensor Database Systems. In *Proc. of the Int. Conf. on Mobile Data Management (MDM)*, pages 3–14, 2001.
- [BHS05] Björn Blohsfeld, Christoph Heinz, and Bernhard Seeger. Maintaining Non-parametric Estimators over Data Streams. In *Proc. of the Conf. on Database Systems for Business, Technology, and the Web (BTW)*, pages 385–404, 2005.
- [BKSS90] Norbert Beckmann, Hans-Peter Kriegel, Ralf Schneider, and Bernhard Seeger. The R\*-Tree: An Efficient and Robust Access Method for Points and Rectangles. In *Proc. of the ACM SIGMOD Int. Conf. on Management of Data*, pages 322–331, 1990.
- [BLT86] José A. Blakeley, Per-Åke Larson, and Frank Wm. Tompa. Efficiently Updating Materialized Views. In *Proc. of the ACM SIGMOD Int. Conf. on Management of Data*, pages 61–71, 1986.
- [BMWM05] Shivnath Babu, Kamesh Munagala, Jennifer Widom, and Rajeev Motwani. Adaptive Caching for Continuous Queries. In *Proc. of the IEEE Int. Conf. on Data Engineering (ICDE)*, pages 118–129, 2005.

## Bibliography

---

- [BS96] Jochen van den Bercken and Bernhard Seeger. Query Processing Techniques for Multiversion Access Methods. In *Proc. of the Int. Conf. on Very Large Databases (VLDB)*, pages 168–179, 1996.
- [BSLH05] Henrike Berthold, Sven Schmidt, Wolfgang Lehner, and Claude-Joachim Hamann. Integrated Resource Management for Data Stream Systems. In *Proc. of the ACM Symposium on Applied Computing (SAC)*, pages 555–562, 2005.
- [BSS05] Gert Brettlecker, Heiko Schuldt, and Hans-Jörg Schek. Towards Reliable Data Stream Processing with OSIRIS-SE. In *Proc. of the Conf. on Database Systems for Business, Technology, and the Web (BTW)*, pages 405–414, 2005.
- [BSW04] Shivnath Babu, Utkarsh Srivastava, and Jennifer Widom. Exploiting k-Constraints to Reduce Memory Overhead in Continuous Queries over Data Streams. *ACM Transactions on Database Systems (TODS)*, 29(3):545–580, 2004.
- [BTW<sup>+</sup>06] Yijian Bai, Hetal Thakkar, Haixun Wang, Chang Luo, and Carlo Zaniolo. A Data Stream Language and System Designed for Power and Extensibility. In *Proc. of the Int. Conf. on Information and Knowledge Management (CIKM)*, pages 337–346, 2006.
- [Car75] A. Cardenas. Analysis and Performance of Inverted Database Structures. *Communications of the ACM*, 18(5):253–263, 1975.
- [CCC<sup>+</sup>02] Donald Carney, Ugur Cetintemel, Mitch Cherniack, Christian Convey, Sangdon Lee, Greg Seidman, Michael Stonebraker, Nesime Tatbul, and Stanley B. Zdonik. Monitoring Streams: A New Class of Data Management Applications. In *Proc. of the Int. Conf. on Very Large Databases (VLDB)*, pages 215–226, 2002.
- [CCD<sup>+</sup>03] Sirish Chandrasekaran, Owen Cooper, Amol Deshpande, Michael J. Franklin, Joseph M. Hellerstein, Wei Hong, Sailesh Krishnamurthy, Samuel Madden, Vijayshankar Raman, Frederick Reiss, and Mehul A. Shah. TelegraphCQ: Continuous Dataflow Processing for an Uncertain World. In *Proc. of the Conf. on Innovative Data Systems Research (CIDR)*, 2003.
- [CCZ<sup>+</sup>03] Donald Carney, Ugur Cetintemel, Stan Zdonik, Alex Rasin, Mitch Cherniak, and Michael Stonebraker. Operator Scheduling in a Data Stream Manager. In *Proc. of the Int. Conf. on Very Large Databases (VLDB)*, pages 838–849, 2003.
- [CDTW00] Jianjun Chen, David J. DeWitt, Feng Tian, and Yuan Wang. NiagaraCQ: A Scalable Continuous Query System for Internet Databases. In *Proc. of the ACM SIGMOD Int. Conf. on Management of Data*, pages 379–390, 2000.
- [CF02] Sirish Chandrasekaran and Michael J. Franklin. Streaming Queries over Streaming Data. In *Proc. of the Int. Conf. on Very Large Databases (VLDB)*, pages 203–214, 2002.

- [CFPR00] Corinna Cortes, Kathleen Fisher, Daryl Pregibon, and Anne Rogers. Hancock: A Language for Extracting Signatures from Data Streams. In *Proc. of the ACM SIGKDD Int. Conf. on Knowledge Discovery and Data Mining (KDD)*, pages 9–17, 2000.
- [CGRS00] Kaushik Chakrabarti, Minos N. Garofalakis, Rajeev Rastogi, and Kyuseok Shim. Approximate Query Processing Using Wavelets. In *Proc. of the Int. Conf. on Very Large Databases (VLDB)*, pages 111–122, 2000.
- [Cha96] Christopher Chatfield. *The Analysis of Time Series: An Introduction*. Chapman & Hall, 1996.
- [CHK<sup>+</sup>03] Michael Cammert, Christoph Heinz, Jürgen Krämer, Martin Schneider, and Bernhard Seeger. A Status Report on XXL – a Software Infrastructure for Efficient Query Processing. *IEEE Data Engineering Bulletin*, 26(2):12–18, 2003.
- [CHK<sup>+</sup>06] Michael Cammert, Christoph Heinz, Jürgen Krämer, Tobias Riemenschneider, Maxim Schwarzkopf, Bernhard Seeger, and Alexander Zeiss. Stream Processing in Production-to-Business Software. In *Proc. of the IEEE Int. Conf. on Data Engineering (ICDE)*, pages 168–169, 2006.
- [CHK<sup>+</sup>07] Michael Cammert, Christoph Heinz, Jürgen Krämer, Bernhard Seeger, Sonny Vaupel, and Udo Wolske. Flexible Multi-Threaded Scheduling for Continuous Queries over Data Streams. In *First Int. Workshop on Scalable Stream Processing Systems (SSPS)*, 2007. (Co-located with ICDE).
- [CHKS03] Michael Cammert, Christoph Heinz, Jürgen Krämer, and Bernhard Seeger. Datenströme im Kontext des Verkehrsmanagements. In *Mobilität und Informationssysteme, Workshop des GI-Arbeitskreises „Mobile Datenbanken und Informationssysteme“*, 2003.
- [CHKS05] Michael Cammert, Christoph Heinz, Jürgen Krämer, and Bernhard Seeger. Sortierbasierte Joins über Datenströmen. In *Proc. of the Conf. on Database Systems for Business, Technology, and the Web (BTW)*, pages 365–384, 2005.
- [CJSS03] Charles D. Cranor, Theodore Johnson, Oliver Spatscheck, and Vladislav Shkapenyuk. Gigascope: A Stream Database for Network Applications. In *Proc. of the ACM SIGMOD Int. Conf. on Management of Data*, pages 647–651, 2003.
- [CKS07] Michael Cammert, Jürgen Krämer, and Bernhard Seeger. Dynamic Metadata Management for Scalable Stream Processing Systems. In *First Int. Workshop on Scalable Stream Processing Systems (SSPS)*, 2007. (Co-located with ICDE).
- [CKSV06] Michael Cammert, Jürgen Krämer, Bernhard Seeger, and Sonny Vaupel. An Approach to Adaptive Memory Management in Data Stream Systems. In *Proc. of the IEEE Int. Conf. on Data Engineering (ICDE)*, pages 137–139, 2006.

## Bibliography

---

- [CM99] Surajit Chaudhuri and Rajeev Motwani. On Sampling and Relational Operators. *IEEE Data Engineering Bulletin*, 22(4):41–46, 1999.
- [CMN99] Surajit Chaudhuri, Rajeev Motwani, and Vivek Narasayya. On Random Sampling over Joins. In *Proc. of the ACM SIGMOD Int. Conf. on Management of Data*, pages 263–274, 1999.
- [CW00] Surajit Chaudhuri and Gerhard Weikum. Rethinking Database System Architecture: Towards a Self-Tuning RISC-Style Database System. In *Proc. of the Int. Conf. on Very Large Databases (VLDB)*, pages 1–10, 2000.
- [Day87] Umeshwar Dayal. Of Nests and Trees: A Unified Approach to Processing Queries That Contain Nested Subqueries, Aggregates, and Quantifiers. In *Proc. of the Int. Conf. on Very Large Databases (VLDB)*, pages 197–208, 1987.
- [Des04] Amol Deshpande. An Initial Study of Overheads of Eddies. *SIGMOD Record*, 33(1):44–49, 2004.
- [DGGR02] Alin Dobro, Minos Garofalakis, Johannes Gehrke, and Rajeev Rastogi. Processing Complex Aggregate Queries over Data Streams. In *Proc. of the ACM SIGMOD Int. Conf. on Management of Data*, pages 61–72, 2002.
- [DGH<sup>+</sup>06] Alan J. Demers, Johannes Gehrke, Mingsheng Hong, Mirek Riedewald, and Walker M. White. Towards Expressive Publish/Subscribe Systems. In *Proc. of the Int. Conf. on Extending Data Base Technology (EDBT)*, pages 627–644, 2006.
- [DGIM02] Mayur Datar, Aristides Gionis, Piotr Indyk, and Rajeev Motwani. Maintaining Stream Statistics over Sliding Windows. In *Proc. of the ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pages 635–644, 2002.
- [DGK82] Umeshwar Dayal, Nathan Goodman, and Randy H. Katz. An Extended Relational Algebra with Control Over Duplicate Elimination. In *Proc. of the ACM SIGMOD Int. Conf. on Management of Data*, pages 117–123, 1982.
- [DGM05] Amol Deshpande, Carlos Guestrin, and Samuel Madden. Using Probabilistic Models for Data Management in Acquisitional Environments. In *Proc. of the Conf. on Innovative Data Systems Research (CIDR)*, pages 317–328, 2005.
- [DGP<sup>+</sup>07] Alan J. Demers, Johannes Gehrke, Biswanath Panda, Mirek Riedewald, Varun Sharma, and Walker M. White. Cayuga: A General Purpose Event Monitoring System. In *Proc. of the Conf. on Innovative Data Systems Research (CIDR)*, pages 412–422, 2007.
- [DGR03] Abhinandan Das, Johannes Gehrke, and Mirek Riedewald. Approximate Join Processing over Data Streams. In *Proc. of the ACM SIGMOD Int. Conf. on Management of Data*, pages 40–51, 2003.
- [DH01] Pedro Domingos and Geoff Hulten. Catching up with the Data: Research Issues in Mining Data Streams. In *ACM SIGMOD Workshop on Research Issues in Data Mining and Knowledge Discovery*, 2001.

- [DH04] Amol Deshpande and Joseph M. Hellerstein. Lifting the Burden of History from Adaptive Query Processing. In *Proc. of the Int. Conf. on Very Large Databases (VLDB)*, pages 948–959, 2004.
- [DMRH04] Luping Ding, Nishant Mehta, Elke A. Rundensteiner, and George T. Heineman. Joining Punctuated Streams. In *Proc. of the Int. Conf. on Extending Data Base Technology (EDBT)*, pages 587–604, 2004.
- [DS00] Jens-Peter Dittrich and Bernhard Seeger. Data Redundancy and Duplicate Detection in Spatial Join Processing. In *Proc. of the IEEE Int. Conf. on Data Engineering (ICDE)*, pages 535–546, 2000.
- [DSTW02] Jens-Peter Dittrich, Bernhard Seeger, David Scot Taylor, and Peter Widmayer. Progressive Merge Join: A Generic and Non-blocking Sort-based Join Algorithm. In *Proc. of the Int. Conf. on Very Large Databases (VLDB)*, pages 299–310, 2002.
- [Exo99] ExoLab Group, Intalio Inc., and Contributors. The Castor Project. <http://www.castor.org>, 1999. Open Source Data Binding Framework for Java.
- [FRM94] Christos Faloutsos, M. Ranganathan, and Yannis Manolopoulos. Fast Subsequence Matching in Time-Series Databases. In *Proc. of the ACM SIGMOD Int. Conf. on Management of Data*, pages 419–429, 1994.
- [GAE06] Thanaa M. Ghanem, Walid G. Aref, and Ahmed K. Elmagarmid. Exploiting Predicate-window Semantics over Data Streams. *SIGMOD Record*, 35(1):3–8, 2006.
- [Gar85] Everette S. Gardner, Jr. Exponential smoothing: The state of the art. *Journal of Forecasting*, 4(1):1–28, 1985.
- [GCB<sup>+</sup>97] Jim Gray, Surajit Chaudhuri, Adam Bosworth, Andrew Layman, Don Reichtart, Murali Venkatrao, Frank Pellow, and Hamid Pirahesh. Data Cube: A Relational Aggregation Operator Generalizing Group-By, Cross-Tab, and Sub-Totals. *Data Mining and Knowledge Discovery*, 1(1):29–53, 1997.
- [GHJV95] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley Professional Computing Series, 1995.
- [GHLK06] Hector Gonzalez, Jiawei Han, Xiaolei Li, and Diego Klabjan. Warehousing and Analyzing Massive RFID Data Sets. In *Proc. of the IEEE Int. Conf. on Data Engineering (ICDE)*, page 83, 2006.
- [GHM<sup>+</sup>07] Thanaa M. Ghanem, Moustafa A. Hammad, Mohamed F. Mokbel, Walid G. Aref, and Ahmed K. Elmagarmid. Incremental Evaluation of Sliding-Window Queries over Data Streams. *IEEE Transactions on Knowledge and Data Engineering (TKDE)*, 19(1):57–72, 2007.

## Bibliography

---

- [GJ01] César A. Galindo-Legaria and Milind Joshi. Orthogonal Optimization of Subqueries and Aggregation. In *Proc. of the ACM SIGMOD Int. Conf. on Management of Data*, pages 571–581, 2001.
- [GJSS05] Dengfeng Gao, Christian S. Jensen, Richard T. Snodgrass, and Michael D. Soo. Join Operations in Temporal Databases. *VLDB Journal*, 14(1):2–29, 2005.
- [GKMS01] Anna C. Gilbert, Yannis Kotidis, S. Muthukrishnan, and Martin Strauss. Surfing Wavelets on Streams: One-Pass Summaries for Approximate Aggregate Queries. In *Proc. of the Int. Conf. on Very Large Databases (VLDB)*, pages 79–88, 2001.
- [GM95] Ashish Gupta and Inderpal Singh Mumick. Maintenance of Materialized Views: Problems, Techniques, and Applications. *IEEE Data Engineering Bulletin*, 18(2):3–18, 1995.
- [GÖ03a] Lukasz Golab and M. Tamer Özsu. Processing Sliding Window Multi-Joins in Continuous Queries over Data Streams. In *Proc. of the Int. Conf. on Very Large Databases (VLDB)*, pages 500–511, 2003.
- [GÖ03b] Lukasz Golab and M. Tamer Özsu. Issues in Data Stream Management. *SIGMOD Record*, 32(2):5–14, 2003.
- [GÖ05] Lukasz Golab and M. Tamer Özsu. Update-Pattern-Aware Modeling and Processing of Continuous Queries. In *Proc. of the ACM SIGMOD Int. Conf. on Management of Data*, pages 658–669, 2005.
- [GPÖ06] Lukasz Golab, Piyush Prahladka, and M. Tamer Özsu. Indexing Time-Evolving Data With Variable Lifetimes. In *Proc. of the Int. Conf. on Scientific and Statistical Database Management (SSDM)*, pages 265–274, 2006.
- [Gra93] Goetz Graefe. Query Evaluation Techniques for Large Databases. *ACM Computing Surveys*, 25(2):73–170, 1993.
- [GS90] Himawan Gunadhi and Arie Segev. A Framework for Query Optimization in Temporal Databases. In *Proc. of the Int. Conf. on Scientific and Statistical Database Management (SSDM)*, pages 131–147, 1990.
- [GS92] Hector Garcia-Molina and Kenneth Salem. Main Memory Database Systems: An Overview. *IEEE Transactions on Knowledge and Data Engineering (TKDE)*, 4(6):509–516, 1992.
- [GUW00] Hector Garcia-Molina, Jeffrey D. Ullman, and Jennifer Widom. *Database System Implementation*. Prentice Hall, 2000.
- [GWYL05] Bugra Gedik, Kun-Lung Wu, Philip S. Yu, and Ling Liu. Adaptive Load Shedding for Windowed Stream Joins. In *Proc. of the Int. Conf. on Information and Knowledge Management (CIKM)*, pages 171–178, 2005.

- [HAE03] Moustafa A. Hammad, Walid G. Aref, and Ahmed K. Elmagarmid. Stream Window Join: Tracking Moving Objects in Sensor-Network Databases. In *Proc. of the Int. Conf. on Scientific and Statistical Database Management (SSDBM)*, pages 75–84, 2003.
- [HBR<sup>+</sup>05] Jeong-Hyon Hwang, Magdalena Balazinska, Alexander Rasin, Ugur Cetintemel, Michael Stonebraker, and Stan Zdonik. High-Availability Algorithms for Distributed Stream Processing. In *Proc. of the IEEE Int. Conf. on Data Engineering (ICDE)*, pages 779–790, 2005.
- [HCH<sup>+</sup>99] Eric N. Hanson, Chris Carnes, Lan Huang, Mohan Konyala, Lloyd Noronha, Sashi Parthasarathy, J. B. Park, and Albert Vernon. Scalable Trigger Processing. In *Proc. of the IEEE Int. Conf. on Data Engineering (ICDE)*, pages 266–275, 1999.
- [HH99] Peter J. Haas and Joseph M. Hellerstein. Ripple Joins for Online Aggregation. In *Proc. of the ACM SIGMOD Int. Conf. on Management of Data*, pages 287–298, 1999.
- [HHW97] Joseph M. Hellerstein, Peter J. Haas, and Helen Wang. Online Aggregation. In *Proc. of the ACM SIGMOD Int. Conf. on Management of Data*, pages 171–182, 1997.
- [HMA<sup>+</sup>04] Moustafa A. Hammad, Mohamed F. Mokbel, Mohamed H. Ali, Walid G. Aref, Ann Christine Catlin, Ahmed K. Elmagarmid, Mohamed Y. Eltabakh, Mohamed G. Elfeky, Thanaa M. Ghanem, R. Gwadera, Ihab F. Ilyas, Mirette S. Marzouk, and Xiaopeng Xiong. Nile: A Query Processing Engine for Data Streams. In *Proc. of the IEEE Int. Conf. on Data Engineering (ICDE)*, page 851, 2004.
- [HS05] Christoph Heinz and Bernhard Seeger. Wavelet Density Estimators over Data Streams. In *Proc. of the ACM Symposium on Applied Computing (SAC)*, pages 578–579, 2005.
- [HS06a] Christoph Heinz and Bernhard Seeger. Resource-aware Kernel Density Estimators over Streaming Data. In *Proc. of the Int. Conf. on Information and Knowledge Management (CIKM)*, pages 870–871, 2006.
- [HS06b] Christoph Heinz and Bernhard Seeger. Stream Mining via Density Estimators: A concrete Application. In *Proc. of the Int. Conf. on Management of Data (COMAD)*, 2006.
- [IP99] Yannis E. Ioannidis and Viswanath Poosala. Histogram-Based Approximation of Set-Valued Query-Answers. In *Proc. of the Int. Conf. on Very Large Databases (VLDB)*, pages 174–185, 1999.
- [JAF<sup>+</sup>06a] Shawn R. Jeffery, Gustavo Alonso, Michael J. Franklin, Wei Hong, and Jennifer Widom. A Pipelined Framework for Online Cleaning of Sensor Data Streams. In *Proc. of the IEEE Int. Conf. on Data Engineering (ICDE)*, page 140, 2006.

## Bibliography

---

- [JAF<sup>+</sup>06b] Shawn R. Jeffery, Gustavo Alonso, Michael J. Franklin, Wei Hong, and Jennifer Widom. Declarative Support for Sensor Data Cleaning. In *Proc. of the Int. Conf. on Pervasive Computing*, pages 83–100, 2006.
- [JC04] Qingchun Jiang and Sharma Chakravarthy. Scheduling strategies for processing continuous queries over streams. In *Proc. of the British National Conf. on Databases (BNCOD)*, pages 16–30, 2004.
- [JCE<sup>+</sup>94] Christian S. Jensen, James Clifford, Ramez Elmasri, Shashi K. Gadia, Patrick J. Hayes, and Sushil Jajodia. A Consensus Glossary of Temporal Database Concepts. *SIGMOD Record*, 23(1):52–64, 1994.
- [JMS95] H. V. Jagadish, Inderpal Singh Mumick, and Abraham Silberschatz. View Maintenance Issues for the Chronicle Data Model. In *Proc. of the ACM SIGMOD Int. Conf. on Management of Data*, pages 113–124, 1995.
- [KG95] Ben Kao and Hector Garcia-Molina. An Overview of Real-time Database Systems. *Advances in Real-time Systems*, pages 463–486, 1995.
- [KK06a] Richard Kuntschke and Alfons Kemper. Data Stream Sharing. In *Proc. of the Int. Conf. on Extending Data Base Technology (EDBT) Workshops*, pages 769–788, 2006.
- [KK06b] Richard Kuntschke and Alfons Kemper. Matching and Evaluation of Disjunctive Predicates for Data Stream Sharing. In *Proc. of the Int. Conf. on Information and Knowledge Management (CIKM)*, pages 832–833, 2006.
- [KNV03] J. Kang, J. Naughton, and S. Viglas. Evaluating Window Joins over Unbounded Streams. In *Proc. of the IEEE Int. Conf. on Data Engineering (ICDE)*, pages 341–352, 2003.
- [KS04] Jürgen Krämer and Bernhard Seeger. PIPES - A Public Infrastructure for Processing and Exploring Streams. In *Proc. of the ACM SIGMOD Int. Conf. on Management of Data*, pages 925–926, 2004.
- [KS05] Jürgen Krämer and Bernhard Seeger. A Temporal Foundation for Continuous Queries over Data Streams. In *Proc. of the Int. Conf. on Management of Data (COMAD)*, pages 70–82, 2005.
- [KSKR05] Richard Kuntschke, Bernhard Stegmaier, Alfons Kemper, and Angelika Reiser. StreamGlobe: Processing and Sharing Data Streams in Grid-Based P2P Infrastructures. In *Proc. of the Int. Conf. on Very Large Databases (VLDB)*, pages 1259–1262, 2005.
- [KSPL06] Jürgen Krämer, Bernhard Seeger, Thomas Penzel, and Richard Lenz. *PIPES<sub>med</sub>*: Ein flexibles Werkzeug zur Verarbeitung kontinuierlicher Datenströme in der Medizin. In *51. Jahrestagung der Deutschen Gesellschaft für Medizinische Informatik, Biometrie und Epidemiologie (GMDS)*, 2006.

- [KSSS04] Christoph Koch, Stefanie Scherzinger, Nicole Schweikardt, and Bernhard Stegmaier. FluXQuery: An Optimizing XQuery Processor for Streaming XML Data. In *Proc. of the Int. Conf. on Very Large Databases (VLDB)*, pages 1309–1312, 2004.
- [KWF06] Sailesh Krishnamurthy, Chung Wu, and Michael Franklin. On-the-fly Sharing for Streamed Aggregation. In *Proc. of the ACM SIGMOD Int. Conf. on Management of Data*, pages 623–634, 2006.
- [KYC<sup>+</sup>06] Jürgen Krämer, Yin Yang, Michael Cammert, Bernhard Seeger, and Dimitris Papadias. Dynamic Plan Migration for Snapshot-Equivalent Continuous Queries in Data Stream Systems. In *Proc. of the Int. Conf. on Extending Data Base Technology (EDBT) Workshops*, pages 497–516, 2006.
- [LM93] T. Y. Cliff Leung and Richard R. Muntz. Stream Processing: Temporal Query Processing and Optimization. In *Temporal Databases: Theory, Design, and Implementation*, pages 329–355. Benjamin/Cummings, 1993.
- [LMT<sup>+</sup>05] Jin Li, David Maier, Kristin Tufte, Vassilis Papadimos, and Peter A. Tucker. Semantics and Evaluation Techniques for Window Aggregates in Data Streams. In *Proc. of the ACM SIGMOD Int. Conf. on Management of Data*, pages 311–322, 2005.
- [LPT99] Ling Liu, Calton Pu, and Wei Tang. Continual Queries for Internet Scale Event-Driven Information Delivery. *IEEE Transactions on Knowledge and Data Engineering (TKDE)*, 11(4):610–628, 1999.
- [LTWZ05] Chang Luo, Hetal Thakkar, Haixun Wang, and Carlo Zaniolo. A Native Extension of SQL for Mining Data Streams. In *Proc. of the ACM SIGMOD Int. Conf. on Management of Data*, pages 873–875, 2005.
- [Luc02] David Luckham. *An Introduction to Complex Event Processing In Distributed Enterprise Systems*. Addison-Wesley Longman, 2002. ISBN 0-201-72789-7.
- [LWZ04] Yan-Nei Law, Haixun Wang, and Carlo Zaniolo. Query Languages and Data Models for Database Sequences and Data Streams. In *Proc. of the Int. Conf. on Very Large Databases (VLDB)*, pages 492–503, 2004.
- [MA05] Mohamed F. Mokbel and Walid G. Aref. PLACE: A Scalable Location-aware Database Server for Spatio-temporal Data Streams. *IEEE Data Engineering Bulletin*, 28(3):3–10, 2005.
- [McC85] Edward M. McCreight. Priority Search Trees. *SIAM Journal on Computing*, 14(2):257–276, 1985.
- [MF02] Samuel Madden and Michael J. Franklin. Fjording the Stream: An Architecture for Queries Over Streaming Sensor Data. In *Proc. of the IEEE Int. Conf. on Data Engineering (ICDE)*, pages 555–566, 2002.

## Bibliography

---

- [MFHH02] Samuel Madden, Michael J. Franklin, Joseph M. Hellerstein, and Wei Hong. TAG: A Tiny AGgregation Service for Ad-Hoc Sensor Networks. In *Symposium on Operating System Design and Implementation (OSDI)*, 2002.
- [MFHH05] Samuel Madden, Michael J. Franklin, Joseph M. Hellerstein, and Wei Hong. TinyDB: An Acquisitional Query Processing System for Sensor Networks. *ACM Transactions on Database Systems (TODS)*, 30(1):122–173, 2005.
- [MLA04] Mohamed F. Mokbel, Ming Lu, and Walid G. Aref. Hash-Merge Join: A Non-blocking Join Algorithm for Producing Fast and Early Join Results. In *Proc. of the IEEE Int. Conf. on Data Engineering (ICDE)*, pages 251–263, 2004.
- [MSHR02] Samuel Madden, Mehul A. Shah, Joseph M. Hellerstein, and Vijayshankar Raman. Continuously Adaptive Continuous Queries over Streams. In *Proc. of the ACM SIGMOD Int. Conf. on Management of Data*, pages 49–60, 2002.
- [MWA<sup>+</sup>03] Rajeev Motwani, Jennifer Widom, Arvind Arasu, Brian Babcock, Shivnath Babu, Mayur Datar, Gurmeet Manku, Chris Olsten, Justin Rosenstein, and Rohit Varma. Query Processing, Resource Management, and Approximation in a Data Stream Management System. In *Proc. of the Conf. on Innovative Data Systems Research (CIDR)*, 2003.
- [NACP01] Benjamin Nguyen, Serge Abiteboul, Gregory Cobena, and Mihai Preda. Monitoring XML Data on the Web. In *Proc. of the ACM SIGMOD Int. Conf. on Management of Data*, pages 437–448, 2001.
- [NDM<sup>+</sup>01] Jeffrey F. Naughton, David J. DeWitt, David Maier, Ashraf Aboulnaga, Jianjun Chen, Leonidas Galanis, Jaewoo Kang, Rajasekar Krishnamurthy, Qiong Luo, Naveen Prakash, Ravishankar Ramamurthy, Jayavel Shanmugasundaram, Feng Tian, Kristin Tufte, Stratis Viglas, Yuan Wang, Chun Zhang, Bruce Jackson, Anurag Gupta, and Rushan Chen. The Niagara Internet Query System. *IEEE Data Engineering Bulletin*, 24(2):27–33, 2001.
- [NP82] J. Nievergelt and F. P. Preparata. Plane-Sweep Algorithms for Intersecting Geometric Figures. *Communications of the ACM*, 25(10):739–747, 1982.
- [ÖS95] Gultekin Özsoyoglu and Richard T. Snodgrass. Temporal and Real-Time Databases: A Survey. *IEEE Transactions on Knowledge and Data Engineering (TKDE)*, 7(4):513–532, 1995.
- [PD99] Norman W. Paton and Oscar Díaz. Active Database Systems. *ACM Computing Surveys*, 31(1):63–103, 1999.
- [PS06] Kostas Patroumpas and Timos K. Sellis. Window Specification over Data Streams. In *Proc. of the Int. Conf. on Extending Data Base Technology (EDBT) Workshops*, pages 445–464, 2006.
- [Pug90] William Pugh. Skip Lists: A Probabilistic Alternative to Balanced Trees. *Communications of the ACM*, 33(6):668–676, 1990.

- [RDH03] Vijayshankar Raman, Amol Deshpande, and Joseph M. Hellerstein. Using State Modules for Adaptive Query Processing. In *Proc. of the IEEE Int. Conf. on Data Engineering (ICDE)*, page 353, 2003.
- [RDS<sup>+</sup>04] Elke A. Rundensteiner, Luping Ding, Timothy M. Sutherland, Yali Zhu, Bradford Pielech, and Nishant Mehta. CAPE: Continuous Query Engine with Heterogeneous-Grained Adaptivity. In *Proc. of the Int. Conf. on Very Large Databases (VLDB)*, pages 1353–1356, 2004.
- [RRH99] Vijayshankar Raman, Bhaskaran Raman, and Joseph M. Hellerstein. Online Dynamic Reordering for Interactive Data Processing. In Malcolm P. Atkinson, Maria E. Orlowska, Patrick Valduriez, Stanley B. Zdonik, and Michael L. Brodie, editors, *Proc. of the Int. Conf. on Very Large Databases (VLDB)*, pages 709–720, 1999.
- [RSSB00] Prasan Roy, S. Seshadri, S. Sudarshan, and Siddhesh Bhobe. Efficient and Extensible Algorithms for Multi Query Optimization. In *Proc. of the ACM SIGMOD Int. Conf. on Management of Data*, pages 249–260, 2000.
- [SA85] Richard T. Snodgrass and Ilsoo Ahn. A Taxonomy of Time in Databases. In *Proc. of the ACM SIGMOD Int. Conf. on Management of Data*, pages 236–246, 1985.
- [SCZ05] Michael Stonebraker, Ugur Cetintemel, and Stan Zdonik. The 8 Requirements of Real-time Stream Processing. *SIGMOD Record*, 34(4):42–47, 2005.
- [See91] Bernhard Seeger. Performance Comparison of Segment Access Methods Implemented on Top of the Buddy-Tree. In *Advances in Spatial Databases*, volume 525 of *Lecture Notes in Computer Science*, pages 277–296. Springer, 1991.
- [Sel88] Timos K. Sellis. Multiple-Query Optimization. *ACM Transactions on Database Systems (TODS)*, 13(1):23–52, 1988.
- [SG97] Narayanan Shivakumar and Hector Garcia-Molina. Wave-Indices: Indexing Evolving Databases. In *Proc. of the ACM SIGMOD Int. Conf. on Management of Data*, pages 381–392, 1997.
- [SH98] Mark Sullivan and Andrew Heybey. Tribeca: A System for Managing Large Databases of Network Traffic. In *In Proc. of the USENIX Annual Technical Conference*, pages 13–24, 1998.
- [SHCF03] Mehul A. Shah, Joseph M. Hellerstein, Sirish Chandrasekaran, and Michael J. Franklin. Flux: An Adaptive Partitioning Operator for Continuous Query Systems. In *Proc. of the IEEE Int. Conf. on Data Engineering (ICDE)*, pages 25–36, 2003.
- [SJS00] Giedrius Slivinskas, Christian S. Jensen, and Richard T. Snodgrass. Query Plans for Conventional and Temporal Queries Involving Duplicates and

## Bibliography

---

- Ordering. In *Proc. of the IEEE Int. Conf. on Data Engineering (ICDE)*, pages 547–558, 2000.
- [SJS01] Giedrius Slivinskas, Christian S. Jensen, and Richard T. Snodgrass. A Foundation for Conventional and Temporal Query Optimization Addressing Duplicates and Ordering. *IEEE Transactions on Knowledge and Data Engineering (TKDE)*, 13(1):21–49, 2001.
- [SJS06] Albrecht Schmidt, Christian S. Jensen, and Simona Saltenis. Expiration Times for Data Management. In *Proc. of the IEEE Int. Conf. on Data Engineering (ICDE)*, page 36, 2006.
- [SLJR05] Timothy M. Sutherland, Bin Liu, Mariana Jbantova, and Elke A. Rundensteiner. D-CAPE: Distributed and Self-tuned Continuous Query Processing. In *Proc. of the Int. Conf. on Information and Knowledge Management (CIKM)*, pages 217–218, 2005.
- [SLMK01] Michael Stillger, Guy M. Lohman, Volker Markl, and Mokhtar Kandil. LEO - DB2’s LEarning Optimizer. In *Proc. of the Int. Conf. on Very Large Databases (VLDB)*, pages 19–28, 2001.
- [SLR94] Praveen Seshadri, Miron Livny, and Raghu Ramakrishnan. Sequence Query Processing. In *Proc. of the ACM SIGMOD Int. Conf. on Management of Data*, pages 430–441, 1994.
- [SLR95] Praveen Seshadri, Miron Livny, and Raghu Ramakrishnan. SEQ: A Model for Sequence Databases. In *Proc. of the IEEE Int. Conf. on Data Engineering (ICDE)*, pages 232–239, 1995.
- [SLR96] Praveen Seshadri, Miron Livny, and Raghu Ramakrishnan. The Design and Implementation of a Sequence Database System. In *Proc. of the Int. Conf. on Very Large Databases (VLDB)*, pages 99–110, 1996.
- [SMFH01] Mehul A. Shah, Sam Madden, Michael J. Franklin, and Joseph M. Hellerstein. Java Support for Data-Intensive Systems: Experiences Building the Telegraph Dataflow System. *SIGMOD Record*, 30(4):103–114, 2001.
- [SPAM91] Ulf Schreier, Hamid Pirahesh, Rakesh Agrawal, and C. Mohan. Alert: An Architecture for Transforming a Passive DBMS into an Active DBMS. In *Proc. of the Int. Conf. on Very Large Databases (VLDB)*, pages 469–478, 1991.
- [SQR03] SQR – A Stream Query Repository. <http://www.db.stanford.edu/stream/sqr>, 2003. Joint Effort of Several Data Stream Research Groups.
- [STD<sup>+</sup>00] Jayavel Shanmugasundaram, Kristin Tufte, David J. DeWitt, David Maier, and Jeffrey F. Naughton. Architecting a Network Query Engine for Producing Partial Results. In *Proc. of the Int. Workshop on the World Wide Web and Databases (WebDB)*, pages 58–77, 2000.

- [Str07] StreamBase Systems Inc. <http://www.streambase.com>, 2007.
- [Sul96] Mark Sullivan. Tribeca: A stream database manager for network traffic analysis. In *Proc. of the Int. Conf. on Very Large Databases (VLDB)*, page 594, 1996.
- [SW04] Utkarsh Srivastava and Jennifer Widom. Flexible Time Management in Data Stream Systems. In *Symp. on Principles of Database Systems (PODS)*, pages 263–274, 2004.
- [TCG<sup>+</sup>93] A. Tansel, J. Clifford, S. Gadia, S. Jajodia, A. Segev, and R. T. Snodgrass. *Temporal Databases: Theory, Design, and Implementation*. Benjamin/Cummings, 1993.
- [TCZ<sup>+</sup>03] Nesime Tatbul, Ugur Cetintemel, Stanley B. Zdonik, Mitch Cherniack, and Michael Stonebraker. Load Shedding in a Data Stream Manager. In *Proc. of the Int. Conf. on Very Large Databases (VLDB)*, pages 309–320, 2003.
- [TGNO92] Douglas B. Terry, David Goldberg, David Nichols, and Brian M. Oki. Continuous Queries over Append-Only Databases. In *Proc. of the ACM SIGMOD Int. Conf. on Management of Data*, pages 321–330, 1992.
- [TMSF03] Peter A. Tucker, David Maier, Tim Sheard, and Leonidas Fegaras. Exploiting Punctuation Semantics in Continuous Data Streams. *IEEE Transactions on Knowledge and Data Engineering (TKDE)*, 15(3):555–568, 2003.
- [Tom03] David Toman. Logical Data Expiration. In *Logics for Emerging Applications of Databases*, pages 203–238, 2003.
- [TTPM02] Peter A. Tucker, Kristin Tufte, Vassilis Papadimos, and David Maier. NEX-Mark – A Benchmark for Queries over Data Streams, 2002. Manuscript available at <http://www.cse.ogi.edu/dot/niagara/NEXMark>.
- [UF01] Tolga Urhan and Michael J. Franklin. Dynamic Pipeline Scheduling for Improving Interactive Query Performance. In *Proc. of the Int. Conf. on Very Large Databases (VLDB)*, pages 501–510, 2001.
- [Vit85] Jeffrey Scott Vitter. Random Sampling with a Reservoir. *ACM Transactions on Mathematical Software*, 11(1):37–57, 1985.
- [Vit01] Jeffrey Scott Vitter. External memory algorithms and data structures: Dealing with massive data. *ACM Computing Surveys*, 33(2):209–271, 2001.
- [VN02] Stratis D. Viglas and Jeffrey F. Naughton. Rate-based Query Optimization for Streaming Information Sources. In *Proc. of the ACM SIGMOD Int. Conf. on Management of Data*, pages 37–48, 2002.
- [VNB03] Stratis D. Viglas, Jeffrey F. Naughton, and Josef Burger. Maximizing the Output Rate of Multi-Join Queries over Streaming Information Sources. In *Proc. of the Int. Conf. on Very Large Databases (VLDB)*, pages 285–296, 2003.

## Bibliography

---

- [WA91] Annita N. Wilschut and Peter M. G. Apers. Dataflow Query Execution in a Parallel Main-Memory Environment. In *Proceedings of the First International Conference on Parallel and Distributed Information Systems (PDIS 1991)*, pages 68–77. IEEE Computer Society, 1991.
- [WC96] Jennifer Widom and Stefano Ceri. *Active Database Systems: Triggers and Rules For Advanced Database Processing*. Morgan Kaufmann, 1996.
- [WDR06] Eugene Wu, Yanlei Diao, and Shariq Rizvi. High-Performance Complex Event Processing over Streams. In *Proc. of the ACM SIGMOD Int. Conf. on Management of Data*, pages 407–418, 2006.
- [WZL03] Haixun Wang, Carlo Zaniolo, and Chang Luo. ATLaS: A Small but Complete SQL Extension for Data Mining and Data Streams. In *Proc. of the Int. Conf. on Very Large Databases (VLDB)*, pages 1113–1116, 2003.
- [XXL07] XXL – eXtensible and fleXible Library. <http://www.xxl-library.de>, 2007. University of Marburg: The Database Research Group.
- [YG02] Yong Yao and Johannes Gehrke. The Cougar Approach to In-Network Query Processing in Sensor Networks. *SIGMOD Record*, 31(3):9–18, 2002.
- [YKPS07] Yin Yang, Jürgen Krämer, Dimitris Papadias, and Bernhard Seeger. HybMig: A Hybrid Approach to Dynamic Plan Migration for Continuous Queries. *IEEE Transactions on Knowledge and Data Engineering (TKDE)*, 19(3):398–411, 2007.
- [YW01] Jun Yang and Jennifer Widom. Incremental Computation and Maintenance of Temporal Aggregates. In *Proc. of the IEEE Int. Conf. on Data Engineering (ICDE)*, pages 51–60, 2001.
- [ZGTS02] Donghui Zhang, Dimitrios Gunopoulos, Vassilis J. Tsotras, and Bernhard Seeger. Temporal Aggregation over Data Streams Using Multiple Granularities. In *Proc. of the Int. Conf. on Extending Data Base Technology (EDBT)*, pages 646–663, 2002.
- [ZRH04] Yali Zhu, Elke A. Rundensteiner, and George T. Heineman. Dynamic Plan Migration for Continuous Queries Over Data Streams. In *Proc. of the ACM SIGMOD Int. Conf. on Management of Data*, pages 431–442, 2004.
- [ZS02] Yunyue Zhu and Dennis Shasha. StatStream: Statistical Monitoring of Thousands of Data Streams in Real Time. In *Proc. of the Int. Conf. on Very Large Databases (VLDB)*, pages 358–369, 2002.
- [ZTZ06] Xin Zhou, Hetal Thakkar, and Carlo Zaniolo. Unifying the Processing of XML Streams and Relational Data Streams. In *Proc. of the IEEE Int. Conf. on Data Engineering (ICDE)*, page 50, 2006.

# **Curriculum Vitae**

## **Contact Information**

Jürgen Krämer  
Am Heiligen Garten 10  
36304 Alsfeld-Leusel, Germany  
Email: kraemerj@mathematik.uni-marburg.de

## **Personal Information**

Date of Birth: 28.06.1977  
Place of Birth: Alsfeld, Germany  
Citizenship: German  
Marital status: Married, with one son

## **Education**

- 1997 – 2003      *Studies of Computer Science (Informatik)*  
                        University of Marburg, Germany
- 1996                *Abitur (higher education entrance qualification)*  
                        Albert-Schweitzer-Schule, Alsfeld, Germany

## **Employment History**

- 2003 – 2007      *Research Assistant*  
                        Project “Anfrageverarbeitung aktiver Datenströme” with Prof. Dr. Bernhard Seeger funded by the German Research Foundation (DFG), University of Marburg, Germany
- 2001                *Internship*  
                        Project “Lightweight Extensible Agent Platform” funded by the European Union, SIEMENS Corporate Technology, Information and Communication, Munich, Germany
- 2000 – 2001      *Software Engineer*  
                        KAMAX-Werke Rudolf Kellermann GmbH & Co. KG, Research and Development, Homberg/Ohm, Germany
- 2000 – 2001      *Student Assistant*  
                        Project “Verarbeitung von Massenanfragen in Geo- und Zeitdatenbanken” with Prof. Dr. Bernhard Seeger funded by the German Research Foundation (DFG), University of Marburg, Germany

## **National Service**

- 1996 – 1997      *Civilian Service* at Brüder-Grimm-Schule Alsfeld, Germany

Marburg, May 15, 2007