Lazy Evaluation Methods for Complex Event Processing

Ilya Kolchinsky
Lazy Evaluation Methods
for Complex Event Processing

Research Thesis

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

Ilya Kolchinsky

Submitted to the Senate
of the Technion — Israel Institute of Technology
Adar bet 5779 Haifa March 2019
This research was carried out under the supervision of Prof. Assaf Schuster, in the Faculty of Computer Science.

Some results in this thesis have been published as articles by the author and research collaborators in conferences and journals during the course of the author’s doctoral research period, the most up-to-date versions of which being:


**Acknowledgements**

I would like to express my sincere gratitude to my advisor, Prof. Assaf Schuster, for being an excellent supervisor and a mentor during this research.

I am enormously thankful to my loving family: Rita, Lisik, and Diana. This work would not have been possible without their endless care, love, encouragement, inspiration, and support, that contributed and helped me along the way.

The generous financial help of the Technion is gratefully acknowledged.
# Contents

## List of Figures

<table>
<thead>
<tr>
<th>Figure</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
</tr>
</tbody>
</table>

## Abstract

1 Introduction 3

2 Lazy Evaluation Model for Nondeterministic Finite Automata 7

2.1 Introduction 7

2.2 Eager NFA Evaluation 8

2.2.1 Specification Language 8

2.2.2 The Eager Evaluation Mechanism 10

2.3 Lazy NFA Evaluation 12

2.4 Lazy Chain NFA 14

2.4.1 Sequences 16

2.4.2 Conjunctions 17

2.4.3 Partial Sequences 18

2.4.4 Negations 19

2.4.5 Iterations 23

2.5 Lazy Multi-Chain NFA 26

2.5.1 Disjunctions 26

2.5.2 General Composite Patterns 28

2.6 Lazy Tree NFA 29

2.7 Experimental Evaluation 34

2.7.1 Time Window Size 36

2.7.2 Rarest-to-most-frequent Event Ratio 40

2.7.3 Chain NFA vs. Tree NFA on Dynamic Data 42

2.7.4 State-of-the-art Comparison 42

3 Join Query Optimization Techniques for Complex Event Processing Applications 45

3.1 Introduction 45

3.2 Complex Event Patterns and Evaluation Mechanisms 48

3.2.1 CEP Patterns 49
3.2.2 Order-based Evaluation Mechanisms ........................................ 50
3.2.3 Tree-based Evaluation Mechanisms ........................................ 51
3.3 Plan Generation Problems ....................................................... 52
3.3.1 CEP Plan Generation ....................................................... 52
3.3.2 Join Query Plan Generation ................................................ 53
3.4 The Equivalence of CPG and JQPG for Pure Conjunctive Patterns .... 54
3.4.1 Order-Based Evaluation .................................................... 54
3.4.2 Tree-Based Evaluation ...................................................... 57
3.4.3 Join Query Types .......................................................... 60
3.5 JQPG for General Pattern Types .............................................. 61
3.5.1 Sequence patterns .......................................................... 61
3.5.2 Kleene closure patterns .................................................... 62
3.5.3 Negation patterns .......................................................... 63
3.5.4 Nested patterns ............................................................. 64
3.6 Adapting JQPG Algorithms to Complex Event Processing ............ 64
3.6.1 Pattern Detection Latency .................................................. 65
3.6.2 Event Selection Strategies ............................................... 66
3.6.3 Adaptive Complex Event Processing .................................... 68
3.7 Experimental Evaluation ....................................................... 68
3.7.1 CPG and JQPG Algorithms ............................................... 69
3.7.2 Experimental Setup ........................................................ 69
3.7.3 Experimental Results ...................................................... 71

4 Efficient Adaptive Detection of Complex Event Patterns .................. 79
4.1 Introduction ........................................................................... 79
4.2 Preliminaries ........................................................................ 83
4.2.1 Notations and Terminology ............................................... 83
4.2.2 Detection-Adaptation Loop ............................................... 85
4.2.3 Reoptimizing Decision Problem ........................................ 86
4.3 Invariant-Based Method for the Reoptimizing Decision Problem ...... 87
4.3.1 Invariant Creation .......................................................... 87
4.3.2 Invariant Verification and Adaptation ................................... 90
4.3.3 Correctness Guarantees and the K-invariant Method ............... 91
4.3.4 Distance-Based Invariants ............................................... 92
4.3.5 Tightest Conditions Selection Strategy ................................ 93
4.4 Applications of the Invariant-Based Method ................................ 94
4.4.1 Greedy Algorithm for Order-Based Plans ............................... 94
4.4.2 Dynamic Programming Algorithm for Tree-Based Plans .......... 96
4.4.3 General Applicability of the Invariant-Based Method ............... 97
4.5 Experimental Evaluation ....................................................... 98
4.5.1 Experimental Setup .............................................. 98
4.5.2 Experimental Results ............................................ 99

5 Real-Time Multi-Pattern Detection over Event Streams .......... 107
  5.1 Introduction ....................................................... 107
  5.2 Background and Terminology ..................................... 111
  5.3 Multi-Pattern CEP with Prefix Sharing ................................. 113
    5.3.1 Multi-Pattern NFA Evaluation ................................ 114
    5.3.2 Multi-Pattern Tree ........................................... 115
    5.3.3 Runtime Complexity and Multi-Pattern Cost Model .............. 116
    5.3.4 MCEP Optimization Problem .................................. 118
  5.4 Optimization Framework for T-MCEP ................................ 119
    5.4.1 Multi-Pattern Graph .......................................... 120
    5.4.2 Local Search Algorithms for T-MCEP .......................... 122
  5.5 MCEP with Arbitrary Subexpression Sharing ......................... 126
  5.6 Advanced Features of a MCEP System ............................... 129
    5.6.1 SLA Requirements ............................................. 131
    5.6.2 Dynamic MCEP .................................................. 131
    5.6.3 Adaptive MCEP ............................................... 132
    5.6.4 On-the-fly Addition and Removal of Patterns .................. 134
  5.7 Experimental Evaluation .......................................... 134
    5.7.1 Experimental Setup ......................................... 134
    5.7.2 Experimental Results ....................................... 137

6 Related Work .......................................................... 145
  6.1 Complex Event Processing Systems ................................. 145
  6.2 Single-Pattern Optimizations in CEP ................................ 146
  6.3 Lazy Evaluation ................................................... 146
  6.4 Join Query Plan Generation ....................................... 147
  6.5 Adaptive CEP ..................................................... 148
  6.6 Multi-Pattern CEP ................................................ 149

7 Conclusions and Future Work ........................................... 151

A Equivalence of the Eager and the Lazy Chain NFA ................. 155

B ASI Property of the Order-Based Cost Functions ................. 165

C Additional Experimental Results - Performance of the CPG and JQPG Methods by Pattern Type ................................. 169

D Additional Experimental Results - Performance of the Different Adaptation Methods by Pattern Type ......................... 175
List of Figures

1.1 A NFA for the sequence of events $a$, $b$, $c$. .......................... 4

2.1 Non-deterministic evaluation of NFA for Example 1.0.1. (a) The sole
NFA instance is currently at the second evaluation stage, with a single
event in its match buffer. (b) A new event $b$ arrives, and now the NFA
instance can either (1) accept the new event as a part of the potential
match and proceed to the next step, or (2) ignore it (by traversing a self
loop) and keep waiting for a future event of the same type. The problem
is solved by duplicating the instance and applying both moves. ....... 11

2.2 Example of a non-deterministic evaluation of the NFA from Figure 1.1.
The rejecting state is omitted for simplicity. At each step, newly created
instances are magnified. The current state of each instance is highlighted
in gray. ................................................................. 12

2.3 Lazy Chain NFA for Example 1.0.1 with frequency order $C,B,A$. .... 14

2.4 Example of a lazy evaluation of the NFA from Figure 2.3. The rejecting
state is omitted for simplicity. The current state of each instance is
highlighted in gray. .................................................... 15

2.5 Eager NFA for a conjunction pattern $AND(A,B,C)$. The rejecting state
is omitted for simplicity. .............................................. 18

2.6 Lazy Post-Processing Chain NFA for the example negation pattern. The
frequency order of $C,A,D$ is assumed. For simplicity, ordering filters are
omitted. ................................................................. 21

2.7 Lazy First-Chance Chain NFA for the example negation pattern. The
frequency order of $C,A,D$ is assumed. For simplicity, ordering filters are
omitted. ................................................................. 23

2.8 Lazy Multi-Chain NFA example (frequency order $C,B,A,D,E$). ....... 27

2.9 Tree NFA example. .................................................. 32

2.10 Throughput as a function of time window size (logarithmic scale): (a)
sequences; (b) negations; (c) disjunctions; (d) conjunctions and iteration
patterns. ............................................................... 35
2.11 Peak memory consumption as a function of time window size: (a) sequences; (b) negations; (c) disjunctions; (d) conjunctions and iteration patterns. .......................................................... 36

2.12 Peak number of NFA instances as a function of time window size (logarithmic scale): (a) sequences; (b) negations; (c) disjunctions; (d) conjunctions and iteration patterns. .......................................................... 37

2.13 Number of computations per event as a function of time window size (logarithmic scale): (a) sequences; (b) negations; (c) disjunctions; (d) conjunctions and iteration patterns. ......................................................... 38

2.14 Number of memory operations per event as a function of time window size: (a) sequences; (b) negations (logarithmic scale); (c) disjunctions; (d) conjunctions and iteration patterns (logarithmic scale). .................. 39

2.15 Number of computations per pattern match as a function of time window size (logarithmic scale): (a) sequences; (b) negations; (c) disjunctions; (d) conjunctions and iteration patterns. ......................................................... 39

2.16 Number of memory operations per pattern match as a function of time window size (logarithmic scale): (a) sequences; (b) negations; (c) disjunctions; (d) conjunctions and iteration patterns. ......................................................... 40

2.17 Eager vs. Lazy evaluation performance for patterns with varied rarest-to-most-common event ratio: (a) throughput; (b) peak memory consumption; (c) number of predicate evaluations per event; (d) number of memory modifications per event; (e) peak number of simultaneously active instances. 41

2.18 Number of operations as a function of the number of received events on highly dynamic input (pattern SEQ(A,B,C), logarithmic scale). ........ 42

2.19 Throughput of SASE+ and the lazy Chain NFA for sequence, negation and iteration patterns (logarithmic scale). .......................... 43

3.1 Evaluation structures for a CEP pattern SEQ(A,B,C,D): (a) NFA with no reordering; (b) NFA with reordering; (c) evaluation tree. ............... 46

3.2 Execution plans for a join of four relations R₁, R₂, R₃, R₄: (a) a left-deep tree; (b) a bushy tree. .......................................................... 47

3.3 Evaluation trees for a pattern SEQ(A,B,C): (a) a left-deep tree produced by ZStream; (b) a right-deep tree produced by ZStream; (c) an optimal evaluation tree, which cannot be produced by ZStream. ............... 52

3.4 Throughput for different pattern types: (a) stock dataset, order-based methods; (b) stock dataset, tree-based methods; (c) traffic dataset, order-based methods; (d) traffic dataset, tree-based methods. ............... 72

3.5 Memory consumption for different pattern types: (a) stock dataset, order-based methods; (b) stock dataset, tree-based methods; (c) traffic dataset, order-based methods; (d) traffic dataset, tree-based methods. ............... 72
3.6 Performance metrics as a function of the cost for order-based and tree-based patterns: (a) throughput; (b) memory consumption.

3.7 Generation of large plans (selected algorithms): (a) average normalized plan cost; (b) average plan generation time (logarithmic scale). The results are presented as a function of pattern size.

3.8 Throughput vs. latency using different values for the alpha parameter of the cost model.

3.9 Throughput for different event selection strategies (logarithmic scale): (a) order-based methods; (b) tree-based methods.

4.1 Evaluation structures for a sequence of events from streams \( A,B,C \): (a) NFA without reordering; (b) Lazy NFA with reordering.

4.2 General structure of an adaptive CEP system.

4.3 Evaluation trees produced by ZStream for the sequence of events from streams \( A,B,C \): (a) a left-deep tree; (b) a right-deep tree.

4.4 Invariant creation for pattern \( SEQ(A,B,C) \) from Example 4.1.1: (a) selecting the first event type in the detection order; (b) \( C \) is set as the first event type, and selection of the second event type is in process; (c) \( B \) is set as the second type, and only a single event type remains for the third position; (d) the evaluation plan and the invariant set are finalized.

4.5 Throughput of the invariant-based method for different dataset-algorithm pairs as a function of the pattern size and the invariant distance \( d \): (a) traffic dataset / greedy algorithm; (b) traffic dataset / ZStream algorithm; (c) stocks dataset / greedy algorithm; (d) stocks dataset / ZStream algorithm.

4.6 Comparison of the adaptation methods applied on the traffic dataset in conjunction with the greedy algorithm: (a) throughput; (b) relative throughput gain over the non-adaptive method; (c) total number of plan reoptimizations; (d) computational overhead.

4.7 Comparison of the adaptation methods applied on the traffic dataset in conjunction with ZStream algorithm: (a) throughput; (b) relative throughput gain over the non-adaptive method; (c) total number of plan reoptimizations; (d) computational overhead.

4.8 Comparison of the adaptation methods applied on the stocks dataset in conjunction with the greedy algorithm: (a) throughput; (b) relative throughput gain over the non-adaptive method; (c) total number of plan reoptimizations; (d) computational overhead.

4.9 Comparison of the adaptation methods applied on the stocks dataset in conjunction with ZStream algorithm: (a) throughput; (b) relative throughput gain over the non-adaptive method; (c) total number of plan reoptimizations; (d) computational overhead.
5.1 Evaluation mechanisms for an event sequence $A, B, C$: (a) NFA without reordering; (b) NFA with reordering. ................................. 108
5.2 NFA sharing example for event sequences $A, B, C$ and $A, B, D$: (a) no sharing; (b) prefix sharing is applied. ................................. 108
5.3 NFA optimization example for event sequences $A, B, C, D$ and $A, E, C, F$: (a) no sharing or reordering; (b) reordering without sharing; (c) sharing without reordering; (d) a combination of reordering and sharing. 109
5.4 General structure of a MCEP system. ................................. 112
5.5 Multi-pattern trees for a workload consisting of $SEQ(A, B, C)$ and $SEQ(A, B, D)$: (a) evaluation orders $A, B, C$ and $A, B, D$ (maximal sharing); (b) evaluation orders $B, C, A$ and $B, A, D$; (c) evaluation orders $C, B, A$ and $A, D, B$ (minimal sharing). ................................. 114
5.6 MPT modification example: (a) a MPT from Figure 5.5(a) and a local plan for a pattern $SEQ(A, C, E)$; (b) the MPT following the addition of the new evaluation plan (the path corresponding to the newly added plan is highlighted); (c) the MPT after the local evaluation plan for $SEQ(A, B, C)$ is removed. ................................. 116
5.7 A multi-pattern graph for a workload of 6 patterns. Edges corresponding to maximal common subpatterns of size 1 are not shown. The triplet $P_1$, $P_2$ and $P_3$ shares a maximal common pattern $SEQ(A, C)$. $P_3$ and $P_4$ have two distinct maximal common subpatterns. $P_5$ is fully contained in $P_3$. ................................. 121
5.8 Three possible evaluation trees for a pattern $SEQ(A, B, C)$. ................................. 126
5.10 Examples of possible modifications of a MPM in Figure 5.9 as a result of: (a) a SLA enforced on the local evaluation plan of $P_1$ (highlighted); (b) a rise in the arrival rate of the event type $A$ (highlighted); (c) an addition of a new pattern $P_4$ (highlighted) at runtime. ................................. 130
5.11 Colored pattern search strategy for adapting the global evaluation plan for a workload from Figure 5.7 following a change in the arrival rate of the event type $G$. ................................. 133
5.12 Throughput gain as a function of the workload size for different configurations: (a) stocks dataset, simulated annealing; (b) stocks dataset, Tabu search; (c) traffic dataset, simulated annealing; (d) traffic dataset, Tabu search. ................................. 136
5.13 Throughput gain as a function of the workload size for different configurations (prefix-only version): (a) stocks dataset, simulated annealing; (b) stocks dataset, Tabu search; (c) traffic dataset, simulated annealing; (d) traffic dataset, Tabu search. ................................. 136
5.14 Throughput gain of the local search algorithms applied on the stock dataset as a function of: (a) average pattern size; (b) local search running time; (c) pattern timestamp-based window; (d) pattern count-based window. ........................................................................................................... 138

5.15 Throughput gain of the local search algorithms applied on the traffic dataset as a function of: (a) average pattern size; (b) local search running time; (c) pattern timestamp-based window; (d) pattern count-based window. ........................................................................................................... 138

5.16 Throughput gain of the local search algorithms as a function of workload statistical properties: (a) stock dataset, sharing sensitivity; (b) traffic dataset, sharing sensitivity; (c) stock dataset, reordering sensitivity; (d) traffic dataset, reordering sensitivity. ........................................................................................................... 140

5.17 Throughput gain of the state-of-the-art and the local search algorithms applied on the stock dataset as a function of: (a) workload size; (b) average pattern size; (c) pattern time window; (d) pattern count-based window. ........................................................................................................... 142

5.18 Throughput gain of the state-of-the-art and the local search algorithms applied on the traffic dataset as a function of: (a) workload size; (b) average pattern size; (c) pattern time window; (d) pattern count-based window. ........................................................................................................... 142

5.19 Throughput gain as a function of the percentage of the patterns with SLAs requiring optimal local plans. ........................................................................................................... 143

5.20 Throughput gain as a function of the number of input stream event reshufflings per 1000 incoming events. ........................................................................................................... 144

5.21 Throughput gain as a function of the number of dynamic additions and/or removals of patterns to/from the workload per 1000 incoming events. .................. 144

C.1 Throughput as a function of the sequence pattern size: (a) stock dataset, order-based methods; (b) stock dataset, tree-based methods; (c) traffic dataset, order-based methods; (d) traffic dataset, tree-based methods. ................................................... 170

C.2 Memory consumption as a function of the sequence pattern size: (a) stock dataset, order-based methods; (b) stock dataset, tree-based methods; (c) traffic dataset, order-based methods; (d) traffic dataset, tree-based methods. ........................................................................................................... 170

C.3 Throughput as a function of the negation pattern size: (a) stock dataset, order-based methods; (b) stock dataset, tree-based methods; (c) traffic dataset, order-based methods; (d) traffic dataset, tree-based methods. 171

C.4 Memory consumption as a function of the negation pattern size: (a) stock dataset, order-based methods; (b) stock dataset, tree-based methods; (c) traffic dataset, order-based methods; (d) traffic dataset, tree-based methods. ........................................................................................................... 171
C.5 Throughput as a function of the conjunction pattern size: (a) stock dataset, order-based methods; (b) stock dataset, tree-based methods; (c) traffic dataset, order-based methods; (d) traffic dataset, tree-based methods. ................................................................. 172
C.6 Memory consumption as a function of the conjunction pattern size: (a) stock dataset, order-based methods; (b) stock dataset, tree-based methods; (c) traffic dataset, order-based methods; (d) traffic dataset, tree-based methods. ................................................................. 172
C.7 Throughput as a function of the Kleene closure pattern size: (a) stock dataset, order-based methods; (b) stock dataset, tree-based methods; (c) traffic dataset, order-based methods; (d) traffic dataset, tree-based methods. ................................................................. 173
C.8 Memory consumption as a function of the Kleene closure pattern size: (a) stock dataset, order-based methods; (b) stock dataset, tree-based methods; (c) traffic dataset, order-based methods; (d) traffic dataset, tree-based methods. ................................................................. 173
C.9 Throughput as a function of the disjunction pattern size: (a) stock dataset, order-based methods; (b) stock dataset, tree-based methods; (c) traffic dataset, order-based methods; (d) traffic dataset, tree-based methods. ................................................................. 174
C.10 Memory consumption as a function of the disjunction pattern size: (a) stock dataset, order-based methods; (b) stock dataset, tree-based methods; (c) traffic dataset, order-based methods; (d) traffic dataset, tree-based methods. ................................................................. 174
D.1 Comparison of the adaptation methods applied on the traffic dataset in conjunction with the greedy algorithm (sequence patterns): (a) throughput; (b) relative throughput gain over the non-adaptive method; (c) total number of plan reoptimizations; (d) computational overhead. .......... 175
D.2 Comparison of the adaptation methods applied on the traffic dataset in conjunction with ZStream algorithm (sequence patterns): (a) throughput; (b) relative throughput gain over the non-adaptive method; (c) total number of plan reoptimizations; (d) computational overhead. .......... 176
D.3 Comparison of the adaptation methods applied on the stocks dataset in conjunction with the greedy algorithm (sequence patterns): (a) throughput; (b) relative throughput gain over the non-adaptive method; (c) total number of plan reoptimizations; (d) computational overhead. .......... 176
D.4 Comparison of the adaptation methods applied on the stocks dataset in conjunction with ZStream algorithm (sequence patterns): (a) throughput; (b) relative throughput gain over the non-adaptive method; (c) total number of plan reoptimizations; (d) computational overhead. .......... 176
D.5 Comparison of the adaptation methods applied on the traffic dataset in conjunction with the greedy algorithm (conjunction patterns): (a) throughput; (b) relative throughput gain over the non-adaptive method; (c) total number of plan reoptimizations; (d) computational overhead. . 177

D.6 Comparison of the adaptation methods applied on the traffic dataset in conjunction with ZStream algorithm (conjunction patterns): (a) throughput; (b) relative throughput gain over the non-adaptive method; (c) total number of plan reoptimizations; (d) computational overhead. . . . . . . 177

D.7 Comparison of the adaptation methods applied on the stocks dataset in conjunction with the greedy algorithm (conjunction patterns): (a) throughput; (b) relative throughput gain over the non-adaptive method; (c) total number of plan reoptimizations; (d) computational overhead. . . . . . . 177

D.8 Comparison of the adaptation methods applied on the stocks dataset in conjunction with ZStream algorithm (conjunction patterns): (a) throughput; (b) relative throughput gain over the non-adaptive method; (c) total number of plan reoptimizations; (d) computational overhead. . . . . . . 178

D.9 Comparison of the adaptation methods applied on the traffic dataset in conjunction with the greedy algorithm (negation patterns): (a) throughput; (b) relative throughput gain over the non-adaptive method; (c) total number of plan reoptimizations; (d) computational overhead. . . . . . . 178

D.10 Comparison of the adaptation methods applied on the traffic dataset in conjunction with ZStream algorithm (negation patterns): (a) throughput; (b) relative throughput gain over the non-adaptive method; (c) total number of plan reoptimizations; (d) computational overhead. . . . . . . 178

D.11 Comparison of the adaptation methods applied on the stocks dataset in conjunction with the greedy algorithm (negation patterns): (a) throughput; (b) relative throughput gain over the non-adaptive method; (c) total number of plan reoptimizations; (d) computational overhead. . . . . . . 179

D.12 Comparison of the adaptation methods applied on the stocks dataset in conjunction with ZStream algorithm (negation patterns): (a) throughput; (b) relative throughput gain over the non-adaptive method; (c) total number of plan reoptimizations; (d) computational overhead. . . . . . . 179

D.13 Comparison of the adaptation methods applied on the traffic dataset in conjunction with the greedy algorithm (Kleene closure patterns): (a) throughput; (b) relative throughput gain over the non-adaptive method; (c) total number of plan reoptimizations; (d) computational overhead. . 179

D.14 Comparison of the adaptation methods applied on the traffic dataset in conjunction with ZStream algorithm (Kleene closure patterns): (a) throughput; (b) relative throughput gain over the non-adaptive method; (c) total number of plan reoptimizations; (d) computational overhead. . 180
D.15 Comparison of the adaptation methods applied on the stocks dataset in conjunction with the greedy algorithm (Kleene closure patterns): (a) throughput; (b) relative throughput gain over the non-adaptive method; (c) total number of plan reoptimizations; (d) computational overhead. . 180

D.16 Comparison of the adaptation methods applied on the stocks dataset in conjunction with ZStream algorithm (Kleene closure patterns): (a) throughput; (b) relative throughput gain over the non-adaptive method; (c) total number of plan reoptimizations; (d) computational overhead. . 180

D.17 Comparison of the adaptation methods applied on the traffic dataset in conjunction with the greedy algorithm (composite patterns): (a) throughput; (b) relative throughput gain over the non-adaptive method; (c) total number of plan reoptimizations; (d) computational overhead. . 181

D.18 Comparison of the adaptation methods applied on the traffic dataset in conjunction with ZStream algorithm (composite patterns): (a) throughput; (b) relative throughput gain over the non-adaptive method; (c) total number of plan reoptimizations; (d) computational overhead. . . . . . . 181

D.19 Comparison of the adaptation methods applied on the stocks dataset in conjunction with the greedy algorithm (composite patterns): (a) throughput; (b) relative throughput gain over the non-adaptive method; (c) total number of plan reoptimizations; (d) computational overhead. . 181

D.20 Comparison of the adaptation methods applied on the stocks dataset in conjunction with ZStream algorithm (composite patterns): (a) throughput; (b) relative throughput gain over the non-adaptive method; (c) total number of plan reoptimizations; (d) computational overhead. . . . . . . 182
Abstract

Rapid advances in data-driven applications over recent years have intensified the need for efficient real-time detection of arbitrarily complex patterns in massive data streams. Complex event processing (CEP) is a prominent technology widely employed for performing this task in many areas, including online finance, security monitoring, credit card fraud detection, and IoT (Internet of Things) technologies. An increasingly active and rapidly developing area of academic research, CEP functionality is also provided by multiple open source libraries and commercial data analysis platforms.

CEP engines operate by collecting basic data items arriving from input data streams and using them to infer occurrences of complex events according to the user-defined patterns. To that end, data items are combined into higher-level entities matching the pattern structure. To guarantee detection correctness, a CEP system is required to actively maintain all subsets of data items that might eventually become a part of a successful pattern match. As a result, the overall number of such potential matches grows exponentially with the size and the sophistication of the pattern being detected. Considering that real-life patterns typically incorporate a highly convoluted structure and may consist of 10 or more events connected by increasingly complex operators, this situation makes large-scale complex event processing virtually infeasible even for businesses capable of acquiring extensive computation power. In addition, modern CEP applications are often required to process hundreds or even thousands of patterns and streams in parallel under tight real-time constraints, which increases the magnitude of the problem.

In this thesis, we present a novel method for overcoming the exponential resource requirements of complex event processing. Our solution is based on the principle of 'statistic-based lazy evaluation'. Under this paradigm, incoming data items are allowed to be processed in an order different from their natural order of appearance in an input stream. As a result, statistical properties of the underlying data, such as the arrival frequencies of different types of items and the selectivities (probabilities of success) of the pattern-defined constraints, can be utilized to construct evaluation plans providing close-to-optimal detection performance.

We describe an efficient lazy evaluation mechanism for complex event processing based on nondeterministic finite automata. By exploiting the similarity of our problem to the well-known problem of join query plan generation, we develop a procedure
for adapting the existing join-based algorithms to the CEP domain, thus creating a family of algorithms for generating practically effective pattern detection plans. As the statistical data properties required for plan generation are rarely known in advance and may change dynamically, we devise an efficient and precise adaptive method that continuously estimates the current statistic values of the required data characteristics on-the-fly and, if and whenever necessary, modifies the evaluation plan accordingly. Finally, we extend our methods to multi-pattern CEP environment, showing how the lazy evaluation approach can facilitate common subexpression sharing between different patterns in a workload. An extensive theoretical and empirical analysis of our innovations demonstrates their superiority over state-of-the-art approaches.
Chapter 1

Introduction

Real-time detection of complex data patterns is one of the fundamental tasks in stream processing. Many modern applications present a requirement for tracking data items arriving from multiple input streams and extracting occurrences of their predefined combinations. Examples of such areas include financial services [DGH+06], electronic health record systems [BEE+10], sensor networks [HCCI08], security monitoring [BSH+15], credit card fraud detection [SMMP09], and more recently IoT (Internet of Things) technologies [ZSP12].

Complex event processing (CEP) is a prominent technology for providing this functionality. CEP systems treat data items as primitive events arriving from event sources. As new primitive events are observed, they are assembled into higher-level complex events matching the specified patterns. The process of complex event detection generally consists of collecting primitive events and combining them into potential (partial) matches using some type of detection model. As more events are added to a partial match, a full pattern match is eventually formed and reported. An active area of academic research [AE04, ADGI08, CM12a, DGH+06, MM09, WDR06], CEP functionality is also provided by multiple commercial data analysis platforms [AAB+06, Bro13, GWA+09, RD15].

To illustrate the above, consider the following example.

Example 1.0.1. A stock market monitoring application is requested to detect non-standard behavior of stock prices. For each stock identifier, a primitive event is generated when its price exceeds some predefined value. We would like to detect a complex event in which an irregularly high price of stock $St1$ is detected, followed by a high price of stock $St2$, which is also followed by a high price of stock $St3$, within a time window of one hour. High price is defined as exceeding a given threshold $T$.

Now, consider an input stream $a_1, a_2, b_1, b_2, c$, where events $a$, $b$ and $c$ denote observations of high values of stocks $St1$, $St2$ and $St3$ respectively. Assuming that we are interested in detecting all combinations of the primitive events, the matches to be returned are $\{a_1b_1c\}$, $\{a_2b_1c\}$, $\{a_1b_2c\}$ and $\{a_2b_2c\}$.
Various methods have been discussed for evaluating complex event patterns. One popular approach is to employ Non-deterministic Finite Automata (NFAs) [ADGI08, CM10, DGH*06, GADI08, PSB03, WDR06]. A pattern is compiled into an NFA consisting of a set of states and conditional transitions between them. States are arranged in a pattern-specific topology. Transitions are triggered by the arrival of an appropriate event on the stream. More generally, we will refer to the structure used by a CEP engine as an internal representation for the pattern to be monitored as the evaluation mechanism. Other notable CEP evaluation mechanisms include finite state machines [AcT08, SMMP09], evaluation trees [MM09], and event processing networks [EN10, REG11].

A NFA for Example 1.0.1 is displayed in Figure 1.1. At each point in time, an instance of the automaton is maintained for every detected partial match of the pattern. This instance is kept until a full match is detected. As an example, consider again a stream $a_1, a_2, b_1, b_2, c$. After the first three events have arrived, the system will maintain match prefixes $\{a_1\}$, $\{a_2\}$, $\{a_1 b_1\}$ and $\{a_2 b_1\}$. After the fourth event, two more instances will be added. Finally, following the arrival of $c$, the NFA detects four sequences matching the pattern.

This approach, however, can prove inefficient when events arrive at highly varied rates or otherwise exhibit different statistical characteristics. Consider an input stream in which events of types $A$ and $B$ arrive at a rate of one per second, whereas an event of type $C$ arrives once every 12 hours. In this case, the system must maintain a large number of pairs of $A$’s and $B$’s that might not lead to any matches. In general, since a CEP system must create and maintain a separate instance of a NFA for each subset of events that might become a part of a successful match, the number of prefixes to be kept grows exponentially with the number of event types in a pattern. Consequently, this method becomes even more wasteful in terms of memory and computational resources for longer patterns.

The described situation could be avoided if the evaluation started from $C$, the rarest event type. This way, significantly fewer partial matches would be created. However, no current NFA-based CEP evaluation mechanism supports out-of-order processing of incoming events. Moreover, even if we assume the existence of such functionality, determining an optimal processing order for a general pattern based on the underlying

Figure 1.1: A NFA for the sequence of events $a$, $b$, $c$. 
data statistics is not a trivial task, especially considering the abundance of complex operators and predicates in real-life pattern specifications. A new methodology is needed for efficient generation of plans to be deployed by an out-of-order evaluation mechanism.

To apply this plan-based CEP paradigm in real-life systems, multiple practical considerations should be taken into account. Most importantly, the assumption of a priori knowledge of the required statistical properties is unacceptable. The data characteristics are rarely known in advance and are typically subject to rapid and frequent changes over time. Therefore, an adaptive mechanism is to be devised for extracting the required statistics from the input stream, monitoring them on-the-fly and deciding whether and how the currently employed evaluation plan is to be modified.

Another important aspect is the multi-pattern CEP support. Most large-scale commercial CEP engines are required to simultaneously monitor multiple distinct patterns. In this scenario, shared processing of common subexpressions of different patterns is beneficial for improving system performance. However, many sharing opportunities are lost when no out-of-order processing is possible. Even when such techniques are employed, these opportunities often cannot be facilitated due to each pattern following its own most efficient plan. Thus, a new approach is required to combine individual plan efficiency with high degree of inter-pattern sharing.

This thesis presents multiple innovations that address the concerns described above and provide tools for a CEP system to optimize the otherwise virtually infeasible process of event detection. In Chapter 2, we introduce the so-called 'lazy evaluation' model for nondeterministic finite automata. We formally define and describe a 'lazy NFA' that defers processing of frequent event types and stores them internally upon arrival. Events are then matched in ascending order of arrival frequency, thus minimizing potentially redundant computations. An algorithm for constructing a lazy NFA for common pattern types is presented, including conjunction, negation, iteration, disjunction, and composition. In addition, we propose a 'lazy tree-based NFA' that does not require the frequencies of the event types to be known in advance. Finally, we experimentally evaluate our mechanism on real-world stock trading data. The results demonstrate a performance gain of two orders of magnitude over traditional NFA-based approaches, with significantly reduced memory resource requirements.

Chapter 3 targets the problem of efficient calculation of a close-to-optimal plan for detecting a complex event using a lazy evaluation mechanism. We observe that the process of evaluating a CEP pattern is very similar to executing a multi-join query in traditional data management systems. Despite this similarity, surprisingly little work has been done on utilizing existing join optimization methods to improve the performance of CEP systems. We provide the first theoretical and experimental study of the relationship between these two research areas. We formally prove that the CEP plan generation problem is equivalent to the join query plan generation problem for a restricted class of patterns and can be reduced to it for a considerably wider range of classes. Based on this result, we show how join plan generation techniques developed
over the last decades can be adapted and utilized to provide practically efficient solutions for complex event detection. Our experiments demonstrate the superiority of these techniques over existing strategies for CEP optimization in terms of throughput, latency, and memory consumption.

Adapting lazy evaluation to the changes in the data statistics on-the-fly is discussed in Chapter 4. We present an efficient and precise method for dynamically deciding whether and how the evaluation mechanism should be reoptimized. This method is based on a small set of constraints to be satisfied by the monitored values, defined such that a better evaluation plan is guaranteed if any of the constraints is violated. To the best of our knowledge, the proposed mechanism is the first to provably avoid false positives on reoptimization decisions. We formally prove this claim and demonstrate how our approach can be applied on known algorithms for evaluation plan generation. An extensive experimental evaluation on two real-world datasets demonstrates that our method leads to better performance and accuracy as compared to the existing methods.

In Chapter 5, we utilize the lazy evaluation principle to devise a novel optimization framework for complex event processing in a multi-pattern setting. Our approach is based on formulating the task of multi-pattern detection as a global optimization problem and applying a combination of sharing and reordering techniques to construct the least expensive plan satisfying the problem constraints. To the best of our knowledge, no such fusion was previously attempted for multi-pattern CEP. To locate the best possible evaluation plan in the resulting hyperexponential solution space, we design efficient local search algorithms that exploit the unique problem structure. A comprehensive theoretical and empirical analysis of our system showcases its superiority over state-of-the-art solutions.

Chapter 6 surveys the extensive previous work in the area of complex event processing, as well as in the related fields of stream processing, database query optimization, adaptive query processing, multi-query optimization, and more.

Finally, Chapter 7 concludes the thesis with the discussion of open problems and potential extensions of the presented methods and approaches, which we recognize as challenging and interesting directions for future research.
Chapter 2

Lazy Evaluation Model for Nondeterministic Finite Automata

2.1 Introduction

As we discussed in Chapter 1, a CEP engine maintains a separate instance of an evaluation mechanism for each subset of events that might become a part of a successful match. A newly arriving event is independently matched against all existing instances and, if the corresponding conditions are satisfied, becomes a part of a potential match. As a result, the overall number of such potential (partial) matches grows exponentially with the size of the pattern being detected. While this problem can be partially addressed by altering the evaluation order according to the statistical properties of the underlying data, NFA-based CEP systems lack the functionality of out-of-order evaluation and implicitly require each incoming event to be processed immediately upon arrival. Thus, newly arrived events consume system resources by participating in some computations, even if the outcome of those computations will never be used.

In this chapter, we introduce a new NFA-based matching mechanism that overcomes the above drawback. The proposed mechanism constructs and extends partial matches by adding events in ascending order of frequency, rather than according to their original order in the pattern. This not only minimizes the number of partial matches held in memory, but also reduces computation time, since there are fewer partial matches to extend when processing a given event. Our proposed solution relies on a lazy evaluation mechanism that can either process an event upon arrival or store it in a buffer, referred to as the input buffer, to be processed at a later time if necessary.

In addition, we present two new NFA topologies that make use of the lazy evaluation model to detect patterns; we call these topologies a Chain NFA and a Tree NFA. A Chain NFA requires specifying the frequency order of the event types. For example, to...
construct an automaton for detecting the sequence \(a, b, c\), it is necessary to specify that 
\(b\) is expected to be the most frequent event, followed by \(a\), which is expected to be less 
frequent, followed by \(c\), which is expected to be the least frequent. A Tree NFA also 
employs lazy evaluation, but it does not require specifying the frequency order. Instead, 
it computes the actual order at each step in an ad hoc manner.

Our lazy evaluation model applies to all common types of patterns, including 
sequences, conjunctions, negations, and iterations. For some of them, we believe our 
work is the first to produce an efficient automata-based solution. The construction of 
a lazy NFA for each of the aforementioned pattern types is presented in detail. We 
also extend this topology for use with composite patterns. This extension, which we 
call a lazy Multi-Chain NFA, is capable of detecting an arbitrary composition of the 
operators above. The correctness of all construction algorithms is formally proven in 
the appendix.

We experimentally evaluate our mechanism on real-world stock trading data. The 
results demonstrate performance gain of two orders of magnitude over traditional NFA-
based approaches, with significantly reduced memory resource requirements. It is also 
shown that for every stream of events, the Tree NFA is at least as efficient as the best 
performing Chain NFA.

The remainder of this chapter is organized as follows. Section 2.2 provides the 
required background and briefly describes the NFA evaluation framework. In Section 
2.3, the concepts and ideas of lazy evaluation are presented. We proceed to describe 
how a lazy Chain NFA can be constructed using given frequencies of the participating 
events in Section 2.4. Lazy Multi-Chain NFA is discussed in Section 2.5. We extend 
the above topologies to introduce a Tree NFA in Section 2.6. Section 2.7 presents the 
experimental evaluation.

2.2 Eager NFA Evaluation

In this section, we discuss in detail the two main parts of a CEP system: the specification 
language and the evaluation mechanism. For the former, SASE+ \cite{GADI08} will be 
assumed for the rest of the chapter. For the latter, we first present the “eager” NFA 
evaluation framework over SASE+ patterns as described in \cite{ADGI08}, processing every 
incoming event upon arrival. Then, the modifications required to employ the lazy 
evaluation principle are described. The SASE+ framework was chosen without loss of 
generality, and the same ideas may be applied to any NFA-based method.

2.2.1 Specification Language

The patterns recognized by CEP systems are normally created using declarative specification languages. They are usually based on relational languages extended with additional operators, making it possible to define a wide range of pattern types, such as sequences,
disjunctions and iterations. Basic operators can be combined into arbitrarily composite expressions. Filters, constraints, time windows, and mutual conditions between events can be applied. Examples of complex event specification languages include SASE [WDR06], CQL [ABW06], CEL [BDG+07] and CEDR [BGAH07].

The SASE+ language, thoroughly described in [GADI08], combines a simple, SQL-like syntax with a high degree of expressiveness. The semantics and expressive power of the language are precisely defined in a formal model.

Each primitive event in SASE+ has an arrival timestamp, a type, and a set of attributes associated with the type. An attribute is a data item related to a given event type, represented by a name and a value. A pattern may contain multiple events belonging to the same event type.

In its most basic form, a complex event definition in SASE+ is composed of three building blocks: PATTERN, WHERE and WITHIN. The PATTERN clause defines the primitive events we would like to detect and the operators applied to combine them into a pattern match. In addition, this clause declares the event selection strategy of a pattern. An event selection strategy specifies how events are selected from an input stream for partial matches and whether an already selected event can be considered again for future matches. In this chapter we assume that, for any pattern, all possible matching combinations of events are requested to be detected. Consequently, a primitive event is allowed to participate in an unlimited number of matches. This selection strategy is called skip-till-any-match [ADGI08]. We address additional event selection strategies in Section 3.6.2.

The WHERE clause specifies constraints on the values of data attributes of the primitive events participating in the pattern. These constraints may be combined using Boolean expressions.

Finally, the WITHIN clause defines a time window over the entire pattern, specifying the maximal allowed time interval between the arrivals of the primitive events.

As an example, consider the pattern from Example 1.0.1. One possible representation of this complex event is:

```
PATTERN SEQ(A a, B b, C c)
WHERE skip_till_any_match {
    a.price > T
    AND b.price > T
    AND c.price > T}
WITHIN 1 hour.
```

Here, we define three event types for stocks with identifiers A, B and C. Every primitive event represents a value of a respective stock at some point in time. Since a fixed order on stock price reports is defined, the sequence (SEQ) operator is to be used. We assume that a numerical attribute price is assigned to each event. skip_till_any_match denotes the event selection strategy as presented in Section 2.1.
A wide variety of operators is supported by SASE+. The most commonly used operator types are sequence (SEQ), conjunction (AND), disjunction (OR), negation (marked as ‘˜’), and iteration (marked as ‘+’). We will describe and discuss each of the operators in detail in Section 2.4. Furthermore, as we demonstrate in Section 2.5.2, multiple operators can be combined to form arbitrarily complex nested pattern structures. In particular, negation and iteration operators can be applied on subexpressions as well as on primitive events.

2.2.2 The Eager Evaluation Mechanism

After a SASE+ pattern is created, it is compiled into an NFA, which is then employed on an input stream to detect pattern occurrences. In this section we define the structure of the eager NFA, which is a slightly modified version of the one described in [ADGI08].

Formally, an NFA is defined as follows:

\[ A = (Q, E, q_1, F, R), \]

where \( Q \) is a set of states; \( E \) is a set of directed edges; \( q_1 \) is an initial state; \( F \) is an accepting state; and \( R \) is a rejecting state.

Evaluation starts at the initial state. Transitions between states are triggered by event arrivals from the input stream. A transition may cause an NFA to be replicated, thus creating a new NFA instance. Upon startup, the system creates a single instance, whose current state pointer points to the initial state. As events arrive, multiple instances of an NFA are created and run in parallel, one for each partial match detected up to that point. Every event received on the input stream will be applied to all NFA instances.

Each NFA instance is associated with a match buffer. As we proceed through an automaton towards the final state, we use the match buffer to store the events that initiated the transitions. It is always empty at \( q_1 \) (that is, the initial NFA instance is created with an empty match buffer), and events are gradually added to it during evaluation. This is done by executing actions of the traversed edges, as will be described shortly. When a new NFA instance is created, its match buffer is initialized by replicating the match buffer of the original instance.

The match buffer should be thought of as a logical construct. As discussed in [ADGI08], there is no need to allocate dedicated memory for each match buffer, since multiple match buffers can be stored in a compact manner that takes into account that certain events may be included in many buffers.

If during the traversal of an NFA instance the accepting state is reached, the content of the associated match buffer is returned as a successful match for the pattern. If the rejecting state is reached, the NFA instance and its match buffer are discarded. If the time window specified in the WITHIN block is violated, a special timeout event is generated, resulting also in the rejecting state.
Figure 2.1: Non-deterministic evaluation of NFA for Example 1.0.1. (a) The sole NFA instance is currently at the second evaluation stage, with a single event in its match buffer. (b) A new event $b$ arrives, and now the NFA instance can either (1) accept the new event as a part of the potential match and proceed to the next step, or (2) ignore it (by traversing a self loop) and keep waiting for a future event of the same type. The problem is solved by duplicating the instance and applying both moves.

An edge is defined by the following tuple:

$$e = (q_s, q_d, action, types, condition),$$

where $q_s$ is the source state of an edge; $q_d$ is the destination state; $action$ is always one of those described below; $type$ may be one or more event types specified in the PATTERN block; and $condition$ is a Boolean predicate, reflecting the conditions in the WHERE block that have to be satisfied for the transition to occur.

The $action$ associated with an edge is performed when the edge is traversed. It can be one of the following (the actions listed below are simplified versions of the ones defined for SASE [ADGI08]):

- **take** - consumes the event from the input stream and adds it to the match buffer. A new instance is created.

- **ignore** - skips the event (consumes the event and discards it instead of storing). No new instance is created.

As an example, consider again Figure 1.1. Note that the accepting state can only be reached by executing three $take$ actions; hence, successful evaluation will produce a match buffer containing three primitive events which comprise the detected match.

Several edges may lead from the same state and specify the same event type. In this case, an event will non-deterministically cause multiple traversals from a given state. If an event triggered $n$ edges, the instance will be replicated $n$ times, for each of the possible traversals. As an example, consider the situation described in Figure 2.1 (for simplicity, the rejecting state is omitted). In 2.1(a), there is some instance of an NFA from Figure 1.1 with an event $a$ in its match buffer, currently in state $q_2$ (we mark the current state of an instance with bold border). In 2.1(b), an event $b$ has arrived. This event triggers the traversal of two edges, namely the outgoing $take$ edge.
Figure 2.2: Example of a non-deterministic evaluation of the NFA from Figure 1.1. The rejecting state is omitted for simplicity. At each step, newly created instances are magnified. The current state of each instance is highlighted in gray.

and the outgoing ignore edge. As a result, one new instance will be created to allow both traversals to occur.

Figure 2.2 illustrates the eager evaluation process of the NFA from Figure 1.1, applied on an input stream $a_1, a_2, b_1, b_2, c$.

### 2.3 Lazy NFA Evaluation

In this section, we describe the principles of the lazy evaluation method and the steps required to implement it over an eager NFA framework. Specific NFA topologies applying the lazy evaluation paradigm on different types of patterns will be discussed in the subsequent sections.

First, we will exemplify the need for such a mechanism. As demonstrated in Section 2.1, more frequent primitive events might trigger creation of instances that do not lead to any matches. Consider, for example, how the evaluation in Figure 2.2 might look if the input stream contained 100 events of type $A$, followed by 100 events of type $B$. Since no event of type $C$ is present, no pattern match can exist. However, $100^2$ instances would be spawned and redundant computations would take place. This could have been avoided simply by modifying the evaluation order to start from $C$.

There are two main reasons for the problem described above. First, an eager NFA is
constructed in a manner that preserves the structure of the underlying pattern. Second, any primitive event must be processed immediately upon arrival. Hence, e.g., in a sequence pattern, re-ordering events with mutual temporal constraints is not possible.

The lazy evaluation model takes advantage of the varying arrival rates of the events in the pattern to significantly reduce the use of computational and memory resources. It can be easily implemented by applying a number of modifications on the eager NFA model, which we will present shortly. We assume that all frequencies are known in advance (this assumption will be removed in Section 2.6).

The idea behind lazy evaluation is to enable instances to store incoming events, and if necessary, retrieve them later for processing. This functionality allows us to process incoming events in any arbitrary order, rather than in the order of their appearance in the input stream. Specifically, it can be utilized to arrange NFA states according to the order of event frequencies, as we will exemplify in Section 2.4.

To achieve out-of-order event processing, an additional buffer, referred to as the input buffer, is associated with each NFA instance, and an additional edge action, referred to as store, is defined. When an edge with a store action is traversed, the event causing the traversal is inserted into the input buffer. The input buffer stores events in chronological order.

Events located in the input buffer can later be fetched for processing. This functionality is implemented by extending the semantics of the take action. Whereas in the eager NFA model an event accepted by this type of edge is always taken from the input stream, here it also triggers a search for events of the required type inside the input buffer. The results of this search, as well as events from the input stream, are then evaluated non-deterministically by spawning new NFA instances. If the search yields no result, or if the retrieved events fail to trigger a take transition due to unsatisfied conditions, a special event called search_failed is generated for the current instance.

Consider again the pattern from Example 1.0.1. Assume that the event type C is the least frequent, and the event type A is the most common. Figure 2.3 shows a possible lazy NFA, which detects the pattern in order C,B,A. Upon arrival of A or B, an outgoing store edge of q1 inserts them into the input buffer. Following an arrival of type C, an outgoing take edge of q2 retrieves events belonging to type B from the input buffer and attempts to add each of them to a partial match, as if they arrived from the input stream. The same occurs for events of type A in q3.

Figure 2.4 presents an example of lazy evaluation of the pattern from Example 1.0.1 using the above NFA. The benefits of this mechanism can be observed by comparing it with detection of the same pattern in Figure 2.2. It can be seen that, since incoming events were not processed in order of their arrival, the number of spawned instances and respective calculations performed by them is drastically smaller.

Note that invoking a full scan of the entire input buffer on each take action would be inefficient and unnecessary. While searching for events of type A, we are only interested in those that arrived before the already accepted B. In general, only a certain, usually
very narrow range of events in the input buffer is relevant to a given take edge. Since the input buffer is arranged chronologically, these temporal constraints make it possible to efficiently limit the search to only a small portion of the storage.

To implement the desired functionality, we extend the NFA edge definition:

\[ e = (q_s, q_d, action, types, condition, prec, succ) \]

Here, \( prec \) and \( succ \), also known as ordering filters, are sets of event types which enforce temporal limitations on an event taken by this edge. Elements in \( prec \) must precede this event, while elements in \( succ \) must succeed it. Both sets may only contain event types selected among those already taken during evaluation until this point. In the example above, for the edge taking \( A \), the following holds: \( prec(A) = \emptyset; succ(A) = \{B, C\} \). Thus, while attempting to traverse this edge, the system will scan the buffer from the beginning, but only until the timestamp of \( B \) is reached. For an event type \( E \) to be accepted from the input stream, we require the condition \( succ(E) = \emptyset \).

The following sections will provide formal definitions of the different types of lazy NFAs, including the corresponding ordering filter definitions.

## 2.4 Lazy Chain NFA

The Chain NFA is the first of the two proposed NFA topologies utilizing the constructs of the lazy evaluation model presented in Section 2.3. This section presents the universal, pattern-independent definition of the Chain NFA and its applications for detecting a wide range of pattern types. The second topology, which we refer to as the Tree NFA, is discussed in Section 2.6.

We will start by providing the intuition behind the construction of a generic lazy Chain NFA. All pattern types discussed in this section will share this common structure with only minor adaptations. The following sub-sections will focus on each type, providing formal and detailed definitions.

Given a pattern containing \( n \) primitive event types, a corresponding Chain NFA consists of \( n + 2 \) states. The first \( n + 1 \) states are arranged in a chain according to
ascending frequency order of the events, which we assume to be given in advance. Each of the first $n$ states is responsible for detecting one primitive event in the pattern (the initial state detects the rarest event, the next state detects the second rarest event, etc.). The final state in a chain is the accepting state $F$. In addition, the rejecting state $R$ is connected to all states except for $q_1$ and $F$. Its purpose is to collect invalid and expired partial matches.

We will denote by $freq$ the ascending frequency order in which the first $n$ states are arranged. We will also denote by $e_i$ the $i^{th}$ event type in $freq$ and by $q_i$ the corresponding state in the chain. The state $q_i$ will generally have several types of outgoing edges. A take edge attempts to add the next event in the pattern to the partial match and to advance to the next state $q_{i+1}$. A store edge adds all events of types yet to be processed (succeeding $e_i$ in $freq$) to the input buffer. An ignore edge discards events of already processed types (preceding $e_i$ in $freq$). Finally, a timeout edge accepts the
special timeout event and proceeds to $R$.

Figure 2.3 exemplifies the common structure of a Chain NFA. Note that it is based solely on relative frequencies of primitive events. It depends neither on the operator(s) applied by the input pattern nor on its structure (e.g., on the requested temporal sequence order). Instead, pattern-specific requirements will be expressed by the parameters of the edges, as will be explained below.

### 2.4.1 Sequences

Sequences are patterns requiring a number of primitive events to arrive in a predefined order. The pattern presented in Example 1.0.1 is an example of a sequence of three events. A Chain NFA for sequence patterns conforms fully to the common structure.

Since temporal constraints are crucial for this pattern type, properly defining ordering filters on edges is of particular importance.

We will proceed now to the formal description. Let $E_i$ denote the set of outgoing edges of $q_i$. Let $\text{Prec}_{ord}(e)$ denote all event types preceding an event type $e$ in an order $\text{ord}$. Similarly, let $\text{Succ}_{ord}(e)$ denote all event types succeeding $e$ in $\text{ord}$. Then, $E_i$ will contain the following edges:

- $e^\text{ignore}_i = (q_i, q_i, \text{ignore}, \text{Prec}_{ord}(e_i), \text{true}, \emptyset, \emptyset)$: any event whose type corresponds to one of the already taken events is ignored.
- $e^\text{store}_i = (q_i, q_i, \text{store}, \text{Succ}_{ord}(e_i), \text{true}, \emptyset, \emptyset)$: any event that might be taken in one of the following states is stored in the input buffer.
- $e^\text{timeout}_i = (q_i, R, \text{ignore}, \text{timeout}, \text{true}, \emptyset, \emptyset)$: if a timeout event is detected, the NFA instance proceeds to the rejecting state and is subsequently discarded.
- $e^\text{take}_i = (q_i, q_{i+1}, \text{take}, e_i, \text{cond}_i, \text{prec}_i, \text{succ}_i)$: an event of type $e_i$ is taken only if it satisfies the conditions required by the initial pattern (denoted by $\text{cond}_i$).

Now we will define how ordering filters $\text{prec}_i$ and $\text{succ}_i$ are calculated. Given a set $T$ of event types, let $B(T)$ denote the set of currently buffered events belonging to the types in $T$. Given a set $S$ of events, let $\text{Latest}(S)$ be the latest event in $S$ (i.e., an event with the largest timestamp value). Correspondingly, let $\text{Earliest}(S)$ denote the earliest event in $S$. Finally, let $\text{seq}$ denote the original sequence order as specified by the input pattern.

The ordering filters for a take edge $e^\text{take}_i$ will be defined as follows:

$$\text{prec}_i = \begin{cases} \text{Latest}(B(\text{Prec}_{freq}(e_i) \cap \text{Prec}_{seq}(e_i))) & \text{ if } \text{Prec}_{freq}(e_i) \cap \text{Prec}_{seq}(e_i) \neq \emptyset \\ \emptyset & \text{ otherwise} \end{cases} ;$$
\[
succ_i = \begin{cases} 
  \text{Earliest} \left( B \left( \text{Prec}_{\text{freq}}(e_i) \cap \text{Succ}_{\text{seq}}(e_i) \right) \right) & \text{if } \text{Prec}_{\text{freq}}(e_i) \cap \text{Succ}_{\text{seq}}(e_i) \neq \emptyset \\
  \emptyset & \text{otherwise}
\end{cases}
\]

It can be seen that \( \text{prec}_i \) and \( \text{succ}_i \) consist of a single element each. \( \text{prec}_i \) contains the latest event preceding \( e_i \) in the original sequence order, out of those already accepted when the state \( q_i \) is reached. Similarly, \( \text{succ}_i \) holds the earliest event, out of those available, succeeding \( e_i \) in \( \text{seq} \).

The Chain NFA for sequence patterns will thus be defined as follows:

\[
A = (Q, E, q_1, F, R);
Q = \{q_i|1 \leq i \leq n\} \cup \{F, R\};
E = \bigcup_{i=1}^{n} E_i.
\]

Figure 2.3 demonstrates the Chain NFA for the pattern from Section 2.2.1.

### 2.4.2 Conjunctions

Conjunction patterns detect a set of events in the input stream, regardless of their mutual order of arrival. This pattern type presents a considerable challenge to the traditional NFA-based approaches. The complication follows from the nature of a finite automaton, which requires specifying an order in which pattern elements will be accepted. However, as all orders between the participating events are valid, the only legitimate way to construct a NFA for the desired pattern is by incorporating all of them.

Consider the following simple conjunction pattern:

\[
\text{PATTERN AND}(A \ a, B \ b, C \ c)
\]
\[
\text{WITHIN 1 hour}.
\]

Figure 2.5 illustrates an eager NFA accepting this pattern. It can be seen that the number of states and transitions is exponential in length of the AND clause. As a result, the matching process becomes highly inefficient, even for a small number of event types.

The lazy Chain NFA solves the aforementioned problem. Instead of attempting to match all possible orders, only the ascending frequency order is incorporated, as presented in Figure 2.3. To the best of our knowledge, this is the first method for automata-based detection of conjunction patterns that requires only a linear number of states.

The definition of the Chain NFA for conjunctions is similar to the one presented above for sequence patterns. The only difference is the absence of ordering filters. Since no constraints can be defined on the mutual order of primitive events, the entire content of the input buffer has to be examined during each search operation, with no possibility to discard parts of it. Thus, buffer searches are rather slow as compared to searches
that take place during evaluation of sequences. However, this solution still significantly outperforms the traditional (eager) approach, as our experimental results in Section 2.7 demonstrate.

More formally, a take edge $e^\text{take}_i$ for an event type $e_i$, detected by a state $q_i$, is defined as follows:

$$e^\text{take}_i = (q_i, q_{i+1}, \text{take}, e_i, \text{cond}_i, \emptyset, \emptyset).$$

Otherwise, the construction is identical to the one shown in Section 2.4.1.

### 2.4.3 Partial Sequences

Partial sequence patterns are conjunctions in which temporal constraints exist between subsets of the primitive events involved. As an example, consider a pattern:

\[
\text{PATTERN AND(SEQ(A a, B b), SEQ(C c, D d), E e) WITHIN 1 hour.}
\]

Here, an event of type $B$ must appear after an event of type $A$. However, it may arrive either before or after events of types $C$, $D$ and $E$. Similarly, $D$ has to appear after $C$, and $E$ can appear at any place in a match.

Note that sequence and conjunction patterns, as presented above, are two opposite edge cases of a partial sequence.

The Chain NFA for a partial sequence will be defined identically to the case of a full sequence. It will, however, incorporate ordering filters only for those events participating in at least one sub-sequence. Also, for events appearing in multiple sub-sequences, $\text{prec}_i$ and $\text{succ}_i$ will contain several values. Among those, only the most restrictive ones will be chosen at runtime.
We will now proceed to the formal definition of ordering filters. Let $SEQ = \{seq_1, \ldots, seq_k\}$ denote all sub-sequences in a pattern. Note that the sub-sequences in $SEQ$ do not necessarily contain independent sets. In addition, let

$$Prec_{SEQ}(e) = \bigcup_{seq \in SEQ} Prec_{seq}(e)$$

$$Succ_{SEQ}(e) = \bigcup_{seq \in SEQ} Succ_{seq}(e).$$

The ordering filters for an edge $e_i^{take}$ are then defined as follows:

$$prec_i = \begin{cases} 
\text{Latest} \left( B \left( Prec_{freq}(e_i) \cap Prec_{SEQ}(e_i) \right) \right) & \text{if } Prec_{freq}(e_i) \cap Prec_{SEQ}(e_i) \neq \emptyset \\
\emptyset & \text{otherwise}
\end{cases}$$

$$succ_i = \begin{cases} 
\text{Earliest} \left( B \left( Prec_{freq}(e_i) \cap Succ_{SEQ}(e_i) \right) \right) & \text{if } Prec_{freq}(e_i) \cap Succ_{SEQ}(e_i) \neq \emptyset \\
\emptyset & \text{otherwise}
\end{cases}$$

Otherwise, the construction is identical to the one shown in Section 2.4.1.

### 2.4.4 Negations

In a pattern with negation, some of the primitive event types are not allowed to appear at the predefined places. We will denote them as negated events. Negated events can be specified anywhere in a pattern and form mutual conditions with positive events. Patterns of any type may include a negation part.

The following is an example of a sequence pattern with a negated event, referred below as an example negation pattern:

```
PATTERN SEQ(A a, NOT(B b), C c, D d)
WHERE skip_till_any_match {b.x < c.y}
WITHIN 1 hour.
```

In this case, a successful pattern match will contain instances of $A$, $C$ and $D$ alone. Note that only events of type $B$ satisfying the condition with an event of type $C$ are not allowed to appear.

Existing NFA-based CEP frameworks employ two different techniques for treating negated events. The first is to check for negative conditions as a post-processing step, after the accepting state is reached. This technique introduces a potential performance issue. Consider a case in which events of type $B$ are very frequent and their presence results in the discarding of all partial matches. Matches detected by NFA will thus be invalidated only during post-processing, causing superfluous computations. This situation could be avoided by moving the negated event check to an earlier stage.
The second technique consists of augmenting an NFA with “negative edges” that lead to a rejecting state upon detection of a forbidden event. To the best of our knowledge, existing solutions of this kind only solve limited cases. Namely, only sequences are considered and no conditions between primitive events are supported. The reason for these restrictions is that, when no event buffering is used, it is impossible to verify the absence of a negated event that depends on some future event. Consider again the example above. When an event of type $B$ arrives, an eager NFA cannot check whether it satisfies a condition with some event of type $C$, since $C$ has not yet been received. Our framework avoids this issue, as evaluation is performed out of order.

We propose two Chain NFA-based solutions for detecting negations. The first, which we call a Post-Processing Chain NFA, implements the post-processing paradigm outlined above. The second, First-Chance Chain NFA, attempts to detect a negated event as soon as possible. From an analytical standpoint, neither solution is superior to the other, and each can be favorable in different situations. A more efficient mechanism for a particular negation pattern can be easily selected, either automatically or manually.

**Post-Processing Negation**

The Lazy Post-Processing Chain NFA first detects a sub-chain of positive events, and then proceeds to a second sub-chain, where each state corresponds to a single negated event. For this negative sub-chain, descending frequency order is used (as opposed to ascending order for a positive part). Each negative state is responsible for verifying absence of one negated event. Thus, transitions between negative states are triggered by either reaching a timeout or by an unsuccessful search in the input buffer. These situations are indicated by special timeout and search failed events. Arrival of a forbidden event in a negative state triggers a transition to the rejecting state. The last negative state is followed by the accepting state.

Figure 2.6 demonstrates the Post-Processing Chain NFA for the example negation pattern. Since we only need to check for an occurrence of $B$ before $C$, a scan of the input buffer is sufficient. Hence, only a search failed event is expected. For patterns in which a negated event may appear at the end (e.g., for conjunctions), an edge taking timeout is also required.

We will now formally define the Lazy Post-Processing Chain NFA.

Let $P = \{e_1, \ldots, e_k\}$ be all positive event types in a pattern. Let $N = \{h_1, \ldots, h_l\}$ be all negated event types. We will denote by $freq_p$ the ascending frequency order of the events in $P$, and by $freq_n$ the descending frequency order of the events in $N$. Let $Q_p = \{q_1, \ldots, q_k\}$ be a set of states corresponding to positive events ordered according to $freq_p$, and let $Q_n = \{r_1, \ldots, r_l\}$ be a set of states corresponding to positive events ordered according to $freq_n$. Finally, let $E_q$ denote the set of outgoing edges of a state $q$.

For each positive state $q_i \in Q_p; i \leq k$, the edges in $E_{q_i}$ are defined as for the Chain NFA for the underlying pattern, with the exception of an edge $e_i^{store}$, which stores all
Figure 2.6: Lazy Post-Processing Chain NFA for the example negation pattern. The frequency order of $C, A, D$ is assumed. For simplicity, ordering filters are omitted.

events in $N$ in addition to those in $Succ_{freq}(e_i)$. Also, for $q_k$, the take edge $e_i^{take}$ leads to the first negative state, $r_1$.

For a negative state $r_i \in Q_n; i \leq l$, the edges in $E_r$, are:

- $e_i^{ignore} = (r_i, r_i, ignore, Prec_{freq} (e_i) \cup P, true, \varnothing, \varnothing)$: a positive event or previously checked negative event is ignored.

- $e_i^{store} = (r_i, r_i, store, Succ_{freq} (e_i), true, \varnothing, \varnothing)$: an event which may be potentially taken in one of the following states is stored in the input buffer.

- $e_i^{take} = (r_i, R, take, h_i, cond_i, prec_i, succ_i)$: an instance of a negated event $h_i$ satisfying the conditions triggers a transition to the rejecting state.

If $h_i$ can only be accepted from the input buffer (i.e., the condition $succ(h_i) \neq \varnothing$ holds), an additional edge is present:

- $e_i^{search, failed} = (r_i, r_{i+1}, ignore, search, failed, true, \varnothing, \varnothing)$: in case of a failed search for a buffered event, the execution successfully proceeds to the next state.

Otherwise, if $h_i$ can be accepted from the input stream, an additional edge is present:

- $e_i^{timeout} = (r_i, r_{i+1}, ignore, timeout, true, \varnothing, \varnothing)$: in case of a timeout, the negation test is passed and the execution proceeds to the next state.

Also, for $r_1$, the timeout or search_failed edge leads to the accepting state $F$. The ordering filters $prec_i$ and $succ_i$ are calculated in the same manner as for the underlying pattern type.

We are now ready to define the Lazy Post-Processing Chain NFA:

$$A = (Q, E, q_1, F, R);$$

$$Q = Q_p \cup Q_n \cup \{F, R\}; E = \bigcup_{q \in Q_p \cup Q_n} E_q.$$
Although this NFA shares the previously discussed drawbacks of the post-processing method, it also has several benefits over the other negation NFA, described below. First, since event buffering is an inherent part of the lazy evaluation mechanism, implementing this NFA on top of the existing framework is straightforward. Second, in some scenarios the best strategy is to postpone negation until the end. One example is a very frequent negated event with a highly selective filter condition. Third, this is the only possible approach for conjunctions with negation and for sequences with a negated event at the end. In these cases, we have to wait until the time window expires to perform a negation check.

**First-Chance Negation**

The lazy First-Chance Chain NFA implements a paradigm opposite to that of the Post-Processing Negation NFA. It operates by pushing the detection of negated events to the earliest point possible. The key observation is that it is often unnecessary to wait for all positive events to arrive before launching a negated event check.

Consider the example negation pattern again. Clearly, a potential event of type $B$ is only dependent on events of types $A$ (temporal condition) and $C$ (explicit and temporal conditions). Consequently, we only need to have $A$ and $C$ in our match buffer to check for conflicting $B$ events. However, a solution based on post-processing will also wait for $D$ to arrive. As a result, if an event of type $B$ is found, it will lead to discarded matches and to redundant operations, including the same search for $B$ for every instance of $D$. Furthermore, if $D$ had mutual conditions with $A$ or $C$, they would be verified repeatedly, only to be invalidated later.

Lazy First-Chance Chain NFA overcomes this performance issue. For each negated event, it determines the earliest state in which a check for this event can be executed. From this state, a *take* edge is added, which leads to the rejecting state if an event matching the conditions is encountered. This type of NFA therefore does not have a state corresponding to each negated event. Instead, it consists only of the positive chain, augmented with the aforementioned *take* edges.

As mentioned above, this approach cannot be efficiently applied on a pattern in which a negated event may appear after all positive events in a match. This is because for such an event, the earliest state in which it can be detected is the accepting state.

Figure 2.7 demonstrates the First-Chance Chain NFA for the example negation pattern. As the figure shows, the absence of $B$ is verified before $D$ is accepted, which is indeed the earliest point possible for this pattern. Note that there are two edges between $q_3$ and $F$, one handling the timeout case and the other detecting the negated event type $B$.

We will now proceed to formally define the First-Chance Chain NFA.

Let $P = \{e_1, \cdots, e_k\}$ be all positive event types in a pattern and let $N = \{h_1, \cdots, h_l\}$ be all negated event types. Let $A_{pos} = (Q_{pos}, E_{pos}, q_1, F, R)$ denote a Chain NFA for the
positive part of the pattern. For each negated event type $h_i$ we will define the following:

- $\text{ImmPrec}(h_i)$, the latest event type preceding $h_i$.
- $\text{ImmSucc}(h_i)$, the earliest event type succeeding $h_i$.
- $\text{Cond}(h_i)$, set of all event types forming mutual conditions with $h_i$.
- $\text{DEP}(h_i) = \{\text{ImmPrec}(h_i), \text{ImmSucc}(h_i)\} \cup \text{Cond}(h_i)$, set of all event types which must be detected before the absence of $h_i$ can be validated.
- $q_{\text{DEP}}(h_i)$, the earliest state in $Q_{\text{pos}}$ in which all event types in $\text{DEP}(h_i)$ are already detected.

The First-Chance Chain NFA will then be constructed by augmenting $E_{\text{pos}}$ with an edge from $q_{\text{DEP}}(h_i)$ to $R$ for each $h_i$:

$$A = (Q_{\text{pos}}, E_{\text{pos}} \cup E_{\text{rej}}, q_1, F, R);$$
$$E_{\text{rej}} = \{(q_{\text{DEP}}(h_i), R, \text{take}, h_i, \text{cond}_i) \mid h_i \in N\}.$$ 

In addition, each store edge in $E_{\text{pos}}$ will be modified to apply to events in $N$ as well.

### 2.4.5 Iterations

The term iteration operator (also called Kleene closure) refers to patterns in which given events are allowed to appear multiple and unbounded numbers of times. Detecting iterations is particularly challenging under the skip-till-any-match event selection strategy because of the exponential number of output combinations [ZDI14]. Our solution utilizes lazy evaluation principles to minimize the number of NFA instances and the calculations performed during the detection process.
For clarity of presentation, we will only discuss sequence patterns with a single iterated event. The concepts presented below can be easily extended to conjunction and partial sequence patterns and to an arbitrary number of events under iteration.

Consider the following sequence pattern with an iterated event:

\[ \text{PATTERN SEQ}(A \ a, \ B^+ \ b[], \ C \ c) \]
\[ \text{WITHIN} \ 1 \ \text{hour}. \]

For a sample input stream \(a, b_1, b_2, b_3, c\), the expected output will be:

\[ab_1c, ab_2c, ab_3c, ab_1b_2c, ab_1b_3c, ab_2b_3c, ab_1b_2b_3c.\]

Our approach is to convert this pattern into a regular sequence \(\text{SEQ}(A, B, C)\) and to address the subsets of \(b_1, b_2, b_3\) as separate “events”. For example, the input stream above can be converted to:

\[a(b_1)(b_2)(b_3)(b_1b_2)(b_1b_3)(b_2b_3)(b_1b_2b_3)c.\]

In this case our artificial “\(B^+\)” event type will become the most frequent regardless of the original frequency of \(B\). Consequently, if we were to construct an ordinary Chain NFA for this sequence, we would place the state responsible for detecting \(B^+\) at the end, as this event type would be the last in the ascending frequency order.

Following this principle, two modifications are required for constructing an Iteration Chain NFA from a Sequence Chain NFA. First, an iterated event type has to be placed at the end of the frequency order, regardless of its actual arrival rate. Second, the \(\text{take}\) edge corresponding to this event is required to produce all subsets of its instances.

To implement the second modification, we introduce a new edge action called \(\text{iterate}\). An \(\text{iterate}\) edge operates similarly to a \(\text{take}\) edge, consuming an event and adding it to the match buffer. However, as an \(\text{iterate}\) edge traverses the input buffer, it produces subsets of events belonging to the required type and returns them all. For example, if an NFA with an \(\text{iterate}\) edge for type \(B\) is applied on a stream above, an input buffer will contain 3 \(B\) events and the \(\text{iterate}\) edge will return 7 subsets, thus detecting 7 matches for the pattern.

When an event can be taken from the input stream, an \(\text{iterate}\) edge will add the new event to the input buffer, and then will only generate subsets including this event.

A Chain NFA for the above iteration pattern looks identical to the one displayed in Figure 2.3, with an event type \(B\) pushed to the end and its corresponding edge action changed to \(\text{iterate}\).

Now we are ready to formally define a lazy Iteration Chain NFA. Let our pattern be \(P = \text{SEQ}(e_1, \cdots, e_k^*, \cdots, e_n)\) and let \(\text{freq} = e_{i_1}, \cdots, e_k, \cdots, e_{i_n}\) denote the ascending frequency order of the primitive event types. The desired automaton will be created by the following steps:
1. Define \( freq' = e_{i_1}, \cdots, e_{i_n}, e_k \) (an order identical to \( freq \) except for moving \( e_k \) to the end).

2. Create a Chain NFA \( A_{seq} \) for
\[ P' = SEQ(e_1, \cdots, e_k, \cdots, e_n) \] with respect to the order \( freq' \).

3. Let \( e_k^{take} = (q_{e_k}, F, take, e_k, cond_{e_k}, prec_k, succ_k) \) denote the take edge for \( e_k \) in \( A_{seq} \). Define
\[ e_k^{iterate} = (q_{e_k}, F, iterate, e_k, cond_{e_k}, prec_k, succ_k) \).

4. Produce a new NFA \( A_{iterate} \) by replacing \( e_k^{take} \) in \( A_{seq} \) with \( e_k^{iterate} \).

More formally, if \( A_{seq} = (Q, E, q_1, F, R) \), then:
\[ A_{iterate} = \left( Q, (E \setminus \{ e_k^{take} \}) \cup \{ e_k^{iterate} \}, q_1, F, R \right). \]

**Aggregations**

Aggregation functions (SUM, AVG, MIN, MAX, etc.) can easily be integrated into the framework using the presented approach for iteration patterns. Since the lazy evaluation mechanism inherently supports event buffering, aggregation is performed in a straightforward manner by invoking a desired function on the input buffer contents. Note that an aggregation function can only be applied on events under an iteration operator. Consider the following example:

```
PATTERN SEQ(A a, B+ b[], C c)
WHERE skip_till_any_match {AVG(b[i].x) < c.y}
WITHIN 1 hour.
```

The condition will be evaluated upon traversal of a corresponding \( iterate \) edge. For each fetched subset of the available \( B \) events, the edge action will apply an aggregate function on this subset and validate the condition.

**Repetitions**

A repetition operator is a bounded version of an iteration. In repetition patterns, we require a primitive event to appear at least \( l \) and at most \( m \) times. An iteration pattern is thus a repetition pattern with \( l \) set to one and \( m \) set to infinity.

Support for repetitions can be trivially added by defining lower and upper bounds on the size of a subset returned by an \( iterate \) edge.

**Group-By-Attribute Optimization**

In some cases, creating all subsets of events before any processing takes place is not the most efficient strategy. For example, consider the following iteration pattern:
PATTERN SEQ(A a, B+ b[], C c)
WHERE skip_till_any_match \{b[i].x = b[i-1].x\}
WITHIN 1 hour.

For this pattern, most subsets of instances of B will not be valid, since all subset members are required to share the same value of the attribute \(x\). Therefore, the evaluation procedure described above will perform poorly, even compared to an eager approach.

For scenarios like the one shown, our framework introduces an optimization that allows a user to specify a group-by-attribute. Instances of the iterated event are then hashed in the input buffer by the value of this attribute. Upon traversal of an iterate edge, the generated subsets will only contain events sharing the same attribute value.

2.5 Lazy Multi-Chain NFA

Despite the flexibility of the Chain NFA, its simple structure is insufficient for detecting more complex patterns, involving separate sets of events and matching conditions. One notable example is a pattern featuring a disjunction operator. Disjunction patterns consist of several sub-patterns, of which only one needs to be detected for the successful match to be reported. Consequently, an NFA detecting a disjunction must include multiple paths from the initial to the final state. However, a Chain NFA only contains a single path between \(q_1\) and \(F\), which renders it unsuitable for processing patterns of this type.

We will overcome this limitation by defining a new topology, which we call Multi-Chain NFA. This type of NFA will possess all the qualities of the Chain NFA, but will also provide path selection functionality. To construct it, we will first produce a Chain NFA for each of the nested pattern parts. Then, we will merge the initial, the accepting, and the rejecting states of all sub-automata and join all the remaining states into a single automaton. The resulting NFA will thus be structured as a union of chains leading to the accepting state. Whenever a match for a sub-query is retrieved, one of the paths will be traversed and the match will be reported. Figure 2.8 demonstrates an example of this construction.

In Section 2.5.1 we will formally describe the structure of a Multi-Chain NFA for disjunction patterns. We will then present a generalization to composite patterns in Section 2.5.2.

2.5.1 Disjunctions

Disjunctions are the most commonly used type of composite patterns. They consist of multiple sub-patterns, which can belong to either of the types discussed above. As a set of events satisfying at least a single sub-pattern is detected, it is reported as a match.
Figure 2.8: Lazy Multi-Chain NFA example (frequency order C,B,A,D,E).

for the whole pattern. For example, consider the following:

```
PATTERN OR(SEQ(A a, B b, C c), SEQ(C c, D d, E e))
WITHIN 1 hour.
```

A lazy Multi-Chain NFA for this pattern is displayed in Figure 2.8. Note that a single primitive event is allowed to appear in multiple sub-patterns. In this case, upon arrival of an event of type C, two edge traversals from q₁ will be executed in a non-deterministic manner and two new NFA instances will be created.

We will now proceed to the formal definition. Let $p_1, \cdots, p_m$ be the sub-patterns of the disjunction pattern and let

$$A_1 = (Q_1, E_1, q_1^1, F_1, R_1)$$

$$\vdots$$

$$A_m = (Q_m, E_m, q_1^m, F_m, R_m)$$

denote the Chain NFA for $p_1, \cdots, p_m$.

Let $q_1, F, R$ be the initial, the accepting, and the rejecting states of the new Multi-Chain NFA $A_{OR}$, respectively.

The following definitions describe the outgoing edges of the initial states of $A_1, \cdots, A_m$ and the incoming edges of their final (accepting and rejecting) states.

$$E_{\text{start}}^j = \{ e | e = (q_1^j, r, \text{action, type, condition}) \};$$

$$E_{\text{acc}}^j = \{ e | e = (q, F_j, \text{action, type, condition}) \};$$
Now, we will define the new edges that will replace the existing ones and lead to $q_1, F, R$ in the new NFA.

$$E^j_{\text{rej}} = \{ e | e = (q, R, \text{action}, \text{type}, \text{condition}) \};$$

$$E^j_{\text{start}} = \bigcup_{j=1}^{m} E^j_{\text{start}}; 
E^j_{\text{acc}} = \bigcup_{j=1}^{m} E^j_{\text{acc}}; 
E^j_{\text{rej}} = \bigcup_{j=1}^{m} E^j_{\text{rej}},$$

In addition, let

$$Q_{\text{start}} = \{ q_j^1 | 1 \leq j \leq m \};$$

$$Q_F = \{ F_j | 1 \leq j \leq m \};$$

$$Q_R = \{ R_j | 1 \leq j \leq m \}.$$

Now, we are ready to define the Multi-Chain NFA:

$$A_{\text{OR}} = (Q, E, q_1, F, R);$$

$$Q = \left( \left( \bigcup_{j=1}^{m} Q_j \right) \setminus (Q_{\text{start}} \cup Q_F \cup Q_R) \right) \cup \{ q_1, F, R \};$$

$$E = \left( \left( \bigcup_{j=1}^{m} E_j \right) \setminus (E_{\text{start}} \cup E_{\text{acc}} \cup E_{\text{rej}}) \right) \cup (E_{\text{OR-start}} \cup E_{\text{OR-acc}} \cup E_{\text{OR-rej}}).$$

### 2.5.2 General Composite Patterns

A disjunction pattern can be alternatively viewed as a Boolean formula normalized to its DNF form. Moreover, all non-unary operators presented earlier can be expressed as operations of Boolean calculus. Since any Boolean statement can be converted to DNF, it is possible to employ a lazy Multi-Chain NFA for matching an arbitrary composite SASE+ pattern. The only step to be added to the construction algorithm from the previous section is transforming the input pattern accordingly.

---

1For the purpose of the DNF conversion procedure, we represent a sequence pattern (full or partial) as a conjunction with additional temporal conditions between primitive events.
It should be noted, however, that applying the above procedure may cause some sub-expressions to appear in multiple branches of the main disjunction pattern. This will, consequently, lead to superfluous calculations and NFA instances. For example, consider the following pattern:

```
PATTERN SEQ(A a, OR(B b, C c), D d)
WHERE skip_till_any_match {a.price > d.price}
WITHIN 1 hour.
```

The DNF form of this pattern is

```
OR(SEQ(A a, B b, D d), SEQ(A a, C c, D d)).
```

The derived Multi-Chain NFA will contain two branches, both of which may perform computations for same pairs of events of types $A$ and $D$. The problem becomes even more severe if $B$ and $C$ are negated types. In this case, the same output match $\{a,d\}$ may be reported twice, since it satisfies both sub-patterns.

This issue may be solved by applying known multi-query techniques, e.g., as described in [DGH+06]. We discuss multi-pattern approaches in Chapter 5.

### 2.6 Lazy Tree NFA

The Chain NFA described in Section 2.4 may significantly improve evaluation performance, provided we know the correct frequency order. As shown in the examples above, the more drastic the difference between the arrival rates of different events, the greater the potential improvement.

There are, however, several drawbacks which severely limit the applicability of the Chain NFA in real-life scenarios. First, the assumption of specifying the frequency order in advance is not always realistic. In many cases, it is hard or even impossible to predict the actual arrival rates of primitive event types. Note that the described model is very sensitive to wrong guesses, as specifying a low-frequency event before a high-frequency event will yield many redundant evaluations and overall poor performance. Second, even if it is possible to set up the system with a correct frequency order, we can rarely guarantee that it will remain the same during the run. In many real-life applications the data is highly dynamic, and arrival rates of different events are subject to change on-the-fly. Such diversity may cause an initially efficient Chain NFA to start performing poorly at some point. Continual changes may come, for example, in the form of bursts of usually rare events.

To overcome these problems we introduce the notion of ad hoc order. Instead of relying on a single frequency order specified at the beginning of the run, we determine the current arrival rates on-the-fly and modify the actual evaluation chain according to the order reflecting the current frequencies of the events. Our NFA will thus have a tree
structure, with each of its nodes (states) “routing” the incoming events to the next “hop” according to this dynamically changing order. By performing these “routing decisions” at each evaluation step, we guarantee that any partial match will be evaluated using the most efficient order possible at the moment.

To implement the desired functionality, we require that each state have knowledge regarding the current frequency of each event type. We will use the input buffer introduced above to this end. By its definition, the input buffer of a particular NFA instance contains all events that arrived from the input stream within the specified time window. For each event type, we will introduce a counter containing the current number of events matched with this event type inside the buffer. This counter will be incremented on each insertion of a new event with the corresponding type and decremented upon its removal.

Matching the pattern requires at least one event corresponding to each event type to be present in the input buffer. Hence, we will add a condition stating that no evaluation will be made by a given instance until all the counters are greater than zero. Only when all of the event counters are greater than zero does it make sense to determine the evaluation order, since otherwise the missing event(s) may not arrive at all and the partial matching process will be redundant. After the above condition is satisfied, we can derive the exact ascending order of frequencies based on the currently available data by sorting the counters.

The above calculation will be performed by each state on each matching attempt, and the resulting value will be used to determine the next step in the evaluation order. In terms of NFA, this means that a state needs to select the next state for a partial match based on the current contents of the input buffer. To this end, a state has several outgoing *take* edges as opposed to a single one in Chain NFA. Each edge takes a different event type and the edges point to different states. We will call the NFA employing this structure a *Tree NFA* and will formally define this model below.

We will illustrate the above using the following example. Consider the stock trading scenario from Example 1.0.1. Assume now that all stock updates are represented using a single event type, which we will denote by $E$. This event type has two data attributes: a categorical attribute called “ticker,” which represents the stock for which the event has occurred, and a numerical attribute called “price,” which is the price of the stock. The arrival rates of events with a particular values for “ticker” attribute (i.e., events corresponding to updates of a specific stock) are unknown and subject to change. We are interested in monitoring the following SASE+ pattern:
PATTERN SEQ(E a, E b, E c)
WHERE skip_till_any_match {
  a.ticker = MSFT
  AND b.ticker = GOOG
  AND c.ticker = AAPL
  AND a.price < b.price
  AND b.price < c.price}
WITHIN 1 hour.

Here, MSFT, GOOG and AAPL stand for the stocks of Microsoft, Google and Apple respectively.

Figure 2.9 illustrates a Tree NFA for this pattern. For simplicity, ignore and timeout edges are omitted.

In formal terms, a Tree NFA is structured as a tree of depth \( n - 1 \), the root being the initial state and the leaves connected to the accepting state. Nodes located at each layer \( k \); \( 0 \leq k \leq n - 1 \) (i.e., all nodes in depth \( k \)) are all states responsible for all orderings of \( k \) event types out of the \( n \) event types defined in the pattern. Each such node has \( n - k \) outgoing edges, one for each event type which does not yet appear in the partial ordering this node is responsible for. Those edges are connected to states at the next layer, responsible for all extensions of the ordering of this particular node to length of \( k + 1 \). The only exceptions to this rule are the leaves, which have a single outgoing edge, connected directly to the final state.

For instance, in the example in Figure 2.9, layer 0 contains the initial state \( q_0 \), layer 1 contains states \( q_1, q_2, q_3 \), and layer 2 contains the states \( q_{12}, q_{13}, q_{21}, q_{23}, q_{31}, q_{32} \).

More formally, the states for a Tree NFA are defined as follows. Let \( O_k \) denote the ordered subsets of size \( k \) of the event types \( e_1, \ldots, e_n \). Let

\[
Q_k = \{q_{ord} | ord \in O_k\}
\]

denote the set of states at the layer \( k \) (note that \( Q_0 = \{q_0\} \)). Then the set of all states of the Tree NFA is

\[
Q = \bigcup_{k=0}^{n-1} Q_k \cup \{F, R\}
\]

where \( q_0 = q_\emptyset \).

To describe the edges and their respective conditions, some preliminary definitions are needed.

First, we will complete the definitions required for the ordering filters. Since each state \( q_{ord} \) corresponds to some evaluation order prefix \( ord \), we will set \( ord_e = ord \) for each outgoing edge \( e \) of \( q_{ord} \). As mentioned earlier, it is enough for \( ord_e \) to be a partial order ending with \( \hat{e} \). In other words, each \( take \) edge in the tree derives the corresponding
Let $c_e$ denote the value of the counter of events associated with the type $e$ in the input buffer. Let $se(q_{ord}) = \min \{c_e | e \notin ord\}$ denote the most infrequent event in the input buffer during the evaluation step in which $q_{ord}$ is the current state. Finally, we will define the predicate $p_{ne}(q_{ord})$ (non-empty) as the condition on the input buffer of state $q_{ord}$ to contain at least a single instance of each primitive event not appearing in $ord$ and another predicate $p_{se}(q_{ord}, e)$ to be true if and only if an event $e$ corresponds to event type $se(q_{ord})$. Let $E_{ord}$ denote the set of outgoing edges of $q_{ord}$. Then, $E_{ord}$ will contain the following edges:

- $e_{\text{ignore}} = (q_{ord}, q_{ord}, \text{ignore}, ord, true, \emptyset, \emptyset)$: any event whose type corresponds to one of the already taken events (appearing in the ordering this state corresponds to) is ignored.

- For each event type $e \notin ord$:
  - $e_{\text{store}} = (q_{ord}, q_{ord}, \text{store}, e, \neg p_{ne}(q_{ord}) \lor \neg p_{se}(q_{ord}, e), \emptyset, \emptyset)$: when either the $p_{ne}$ or $p_{se}$ condition is not satisfied, the incoming event is stored into the input buffer.
\( e_{\text{take}}^{\text{ord},e} = (q_{\text{ord}}, q_{\text{ord},e}, \text{take}, e, p_{\text{ne}} (q_{\text{ord}}) \land p_{\text{se}} (q_{\text{ord}, e}) \land \text{cond}_{e} \land \text{prec}_{e} \land \text{succ}_{e}) ; \)

if the contents of the input buffer satisfy the \( p_{\text{ne}} \) and \( p_{\text{se}} \) predicates and an incoming event with a type \( e \) (1) satisfies the conditions required by the initial pattern (denoted by \( \text{cond}_{e} \)); and (2) is located within the scope defined for this state, it is taken into the match buffer and the NFA instance advances to the next layer of the tree.

- For states in the last layer (where \( |\text{ord}| = n \)), the \( \text{take} \) edges are of the form

\( e_{\text{store}}^{\text{ord},e} = (q_{\text{ord}}, F, \text{take}, e, p_{\text{ne}} (q_{\text{ord}}) \land \text{cond}_{e} \land \text{InScope}_{\text{ord}} (e) \land \text{prec}_{e} \land \text{succ}_{e}) . \)

The ordering filters \( (\text{prec}_{e}, \text{succ}_{e}) \) for a Tree NFA are calculated the same way as for a Chain NFA, as described in Section 2.4. However, since the Tree NFA does not have a predefined frequency order \( \text{freq} \) to be used for calculating the ordering filters, \( \text{ord}_{e} \) will be used instead. This order is the effective frequency order applied on the current input.

In addition, each state features a \textit{timeout} edge similarly to the one described for the Chain NFA. We will denote the set of all timeout edges as \( E_{\text{timeout}} \).

The set of all edges for Tree NFA is defined as follows:

\[
E = \left( \bigcup_{\{\text{ord}|q_{\text{ord}} \in Q\}} E_{i} \right) \cup E_{\text{timeout}},
\]

and the NFA itself is defined as follows:

\[
A = (Q, E, q_{1}, F, R),
\]

where \( Q \) and \( E \) are as defined above.

It can be observed that a Tree NFA contains all the possible Chain NFAs for a given pattern, with shared states for common subpatterns. Thus, the execution of a Tree NFA on any input is equivalent to the execution of some Chain NFA on that input. The conditions on Tree NFA edges are designed in such a way that the least frequent event is chosen at each evaluation step. Hence, this Chain NFA is always the one whose given frequency order is the actual frequency order as observed from the input stream. An example is shown in Figure 2.9. Nodes and edges marked in bold illustrate the evaluation path for an input stream satisfying \( \text{count}(AAPL) \leq \text{count}(GOOG) \leq \text{count}(MSFT) \), i.e., corresponding to the frequency order \( c,b,a \).

When implementing the Tree NFA, the number of states might be exponential in \( n \). To overcome this limitation, we propose to implement lazy instantiation of NFA states – only those states reached by at least a single active instance will be instantiated and will actually occupy memory space. After all NFA instances reaching a particular state are terminated, the state will be removed from the NFA as well. Even though the worst case complexity remains exponential in this case, in practice there will be fewer
changes in the event rates than there will be new instances created. This conclusion is supported by our experiments, which are explained in the next section.

For disjunctions and composite patterns, the solution described above can be extended to a Multi-Tree NFA similarly to the way a Chain NFA was extended to a Multi-Chain NFA in Section 2.5. In this topology, the initial state contains an outgoing take edge for each of the disjunction clauses in a pattern, leading to the dedicated Tree NFA corresponding to this clause.

### 2.7 Experimental Evaluation

This section presents a detailed experimental analysis of our method. We applied lazy (Multi-)Chain and Tree NFAs on wide range of patterns, assessing their efficiency and scalability. A comparison with an eager NFA-based CEP framework was conducted, demonstrating an overall performance gain of up to two orders of magnitude in most scenarios.

Our metrics for this study are throughput, memory consumption, and runtime complexity. Throughput is defined as the number of events processed per second. For memory consumption, we estimate the peak size (in MB) of the system memory allocated by our CEP engine during processing. Runtime complexity is measured as the number of calculations and memory modifications per event. For convenience, these two types of operations were evaluated separately.

Real-world historical stock price data from the NASDAQ stock market, taken from [EOD], was used in the experiments. This data spans a 1-year period, covering over 2100 stock identifiers with prices updated on a per minute basis. Our input stream contained 80,509,033 primitive events, each consisting of a stock identifier, a timestamp, and a current price. We augmented the event format with additional attributes, namely a history list of older prices and an identifier of a region to which the stock belongs, e.g., North America or Europe. For each of the 8 regions defined by NASDAQ, a corresponding event type was defined. A primitive event was then considered to belong to a type corresponding to the region of its stock identifier.

The patterns used during evaluation differed only in their operator (e.g., SEQ or AND), the participating event types (i.e., regions) and the time window size. All patterns shared an identical condition of high correlation between each pair of primitive events. That is, the Pearson correlation coefficient between price history lists was required to exceed a predefined threshold $T$. For example, for a sequence operator, three event types, $\text{NACompany}$, $\text{EuCompany}$, $\text{AfrCompany}$, and a time window $h$, the corresponding pattern would be declared as follows:
The following pattern types were used during this study:

- Sequences of 3, 4 and 5 primitive events (marked on all graphs as SEQ3, SEQ4 and SEQ5, respectively).
- Conjunctions of 2 and 3 primitive events (marked as AND2, AND3). Due to the extremely rapid growth of NFA instances during eager evaluation attempts, experiments for higher numbers of events could not be conducted in the environment available for this research.
- Negation patterns for SEQ3, AND3 and SEQ5. In all of the above, the second event was negated. For SEQ5 alone, we evaluated both methods for lazy negation.
Figure 2.11: Peak memory consumption as a function of time window size: (a) sequences; (b) negations; (c) disjunctions; (d) conjunctions and iteration patterns.

presented in Section 2.4.4. They are marked as Lazy-PP and Lazy-FC on all graphs. For the rest of the patterns, only Post-Processing Chain NFA was used.

- Iteration pattern of 3 events (i.e., SEQ(a,b+,c)), with the second event being iterated (marked as ITER3).

- Disjunction patterns consisting of: (1)two sequences of two events (OR2SEQ2); (2)four sequences of two events (OR4SEQ2); (3)two sequences of four events (OR2SEQ4).

All NFA models under examination were implemented in Java. The experiments were run on a machine with 2.20 Ghz CPU and 16.0 GB RAM.

2.7.1 Time Window Size

In our first set of experiments, we compared how the lazy and the eager strategies scale with a growing time window size. Both algorithms were repeatedly applied on a data stream, with values for $h$ ranging between 5 and 30 minutes.

All figures summarizing this experiment contain several sub-graphs, presenting results for different pattern types as listed above.
Figure 2.12: Peak number of NFA instances as a function of time window size (logarithmic scale): (a) sequences; (b) negations; (c) disjunctions; (d) conjunctions and iteration patterns.

Figure 2.10 describes the throughput as a function of $h$. For presentation clarity, the logarithmic scale was used. A steady increase of one to two orders of magnitude can be observed for the lazy approach as compared to its eager counterpart. The performance decline for larger values of $h$ is generally smoother for lazy NFA or comparable in both models. For the negated sequence of five events, a clear advantage of the First-Chance Chain NFA over the Post-Processing method can be seen. This holds due to the negated event being the second in the sequence. Hence, according to First-Chance strategy, verification of its absence can occur as the third event is accepted, rather than after all positive events have arrived. For disjunctions, sub-sequence length affects the overall throughput significantly more than the number of sub-sequences.

Memory consumption measurements are presented in Figures 2.11 and 2.12. Figure 2.11 compares the total memory used during eager and lazy evaluation of various patterns. For all pattern types, lazy automata require less memory than eager automata. This is most evident for sequences and disjunctions, for which the lazy solutions are 1.5 to 3 times more economical for larger time windows. The smallest gain is observed for conjunctions, which can be explained by the absence of ordering filters for this pattern type (Section 2.4.2).
Figure 2.13: Number of computations per event as a function of time window size (logarithmic scale): (a) sequences; (b) negations; (c) disjunctions; (d) conjunctions and iteration patterns.

It should be noted that the above figure only displays the peak memory usage, i.e., the maximal recorded amount of required memory. However, the difference in average memory consumption is more critical. Whereas an eager NFA maintains the same high amount of reserved memory most of the time, a Chain NFA only achieves its peak during brief ‘bursts’. These bursts follow an arrival of a rare event, which triggers a search in the input buffer and creates a large number of intermediate instances.

Two main types of objects contribute to the overall memory consumption. First, new NFA instances are created for each partial match. Second, in lazy NFA primitive events are buffered upon arrival. While the lazy evaluation method obviously keeps more buffered events, it creates dramatically fewer instances. Figure 2.12 demonstrates this by comparing the peak number of simultaneously active instances during evaluation. Again, the logarithmic scale was used. On average, a (Multi-)Chain NFA maintains 5 to 100 times fewer instances at the same time. The biggest gain was achieved for large negation and disjunction patterns.

Now we proceed to the runtime complexity comparison. In this experiment, we analyzed the two evaluation mechanisms in terms of operations performed per primitive event and per successful pattern match. These operations can be divided into two separate categories. The first includes the calculations executed during predicate evaluation. Memory modifications, such as NFA instance creation and deletion, buffering primitive events and removing them upon evaluation, comprise the second category.

In Figure 2.13 the number of predicate evaluations for each tested pattern can be
Figure 2.14: Number of memory operations per event as a function of time window size:
(a) sequences; (b) negations (logarithmic scale); (c) disjunctions; (d) conjunctions and
iteration patterns (logarithmic scale).

Figure 2.15: Number of computations per pattern match as a function of time window
size (logarithmic scale): (a) sequences; (b) negations; (c) disjunctions; (d) conjunctions
and iteration patterns.
Figure 2.16: Number of memory operations per pattern match as a function of time window size (logarithmic scale): (a) sequences; (b) negations; (c) disjunctions; (d) conjunctions and iteration patterns.

seen. On average, lazy NFAs execute 10 to 200 times fewer computations per primitive event. This difference tends to be larger for more complex patterns involving more event types.

Figure 2.14 presents the results for memory modifications. Despite the need to maintain a complex data structure for input buffering and to perform a large volume of insertions, searches and removals, the lazy method does not suffer any performance penalty in comparison to the eager approach. Furthermore, for most patterns, significantly fewer operations were recorded during lazy evaluation. This can be explained by the drastically reduced number of instances to be created (and hence destroyed).

Finally, Figures 2.15 and 2.16 display the runtime complexity measurements adjusted per detected pattern match. The results are similar to those demonstrated above for per event operations.

2.7.2 Rarest-to-most-frequent Event Ratio

The core principle of the lazy evaluation mechanism is to re-order the input stream so that less frequent events will be processed earlier. Thus, from an analytical standpoint, this method achieves its biggest performance gain when the frequencies of the participating events are highly varied. As a consequence, the impact of this re-ordering will diminish as events arrive at at more similar rates, and will completely vanish if all event types are of identical frequency. An interesting question is, how does a Chain NFA perform in these scenarios as compared to an eager NFA?
In our next experiment we attempted to answer this question. To this end, we evaluated a subset of patterns presented above on different selections of event types, using a parameter called \textit{rarest-to-most-frequent event ratio}. It is defined as the maximal ratio of arrival rates of two event types in a pattern.

The frequency of each type was estimated according to the overall number of companies belonging to the respective region. For example, while NASDAQ has 267 registered companies located in Europe, there are only 8 such companies in Africa. Hence, the ratio of the event type \textit{AfrCompany} to \textit{EuCompany} is 8:267, or approximately 1:33. By carefully choosing sets of event types, we created a number of patterns with a rarest-to-most-frequent event ratio ranging from 1:3 to 1:700. Each of the patterns was then evaluated with time window size set to 20. All performance metrics presented in the beginning of this section were measured during this experiment.

The results are displayed in Figure 2.17. Note that in this case we are not interested in the absolute values. Instead, we examine the performance difference of the eager and the lazy evaluation mechanisms. As expected, this difference is inversely proportional to the rarest-to-most-frequent ratio. For small values of this ratio, the results for the lazy method are always at most equal to those of the eager method.
2.7.3 Chain NFA vs. Tree NFA on Dynamic Data

In our next experiment we compared the performance of the NFAs discussed above on data with dynamically changing frequencies of all primitive events. For this experiment alone, synthetic data was used, generated using the FINCoS framework [MBM]. An artificial stream was produced in which the rarest event was switched after each 100,000 incoming events. Then, all NFAs were tested against this input stream using a simple sequence pattern of three event types. The number of computations was measured after each 10,000 events.

Figure 2.18 demonstrates the results. Some of the Chain NFAs were omitted due to very similar results. The x-axis represents the number of events from the beginning of the stream. It can be thought of as the closest estimate to the time axis. The y-axis represents the number of computations per 10,000 events.

This figure illustrates the superiority of the Tree NFA over its competitors and its high adaptivity to changes in event arrival rates. At any single point there is one frequency order that is the most efficient given the current event frequencies. The performance gain of the Chain NFA based on that order over the other Chain NFAs reaches up to two orders of magnitude. However, as soon as the event frequencies change, this NFA loses its advantage. On the other hand, the Tree NFA shows consistent improvement over all Chain NFAs regardless of the input arrival rates.

2.7.4 State-of-the-art Comparison

In our last experiment we compared the performance of the lazy Chain NFA with the official implementation of SASE+ [ZDI10]. The set-up described in Section 2.7.1 was used. Sequence, negation and iteration patterns were employed. To avoid modifying the code of SASE+, we only measured throughput during the evaluation.

Figure 2.19 shows the results. Since our implementation of the eager NFA was based
on SASE+ and carefully followed its formal description, a high degree of similarity with Figure 2.10 can be observed. As demonstrated earlier, using the lazy evaluation method yields a performance gain of one to two orders of magnitude in terms of processed events per time unit.
Chapter 3

Join Query Optimization
Techniques for Complex Event Processing Applications

3.1 Introduction

As described in previous chapters, the part of a CEP system responsible for actual event detection is the evaluation mechanism. The topology of an evaluation mechanism monitoring a pattern $P$ reflects the structure of $P$ and serves as its internal engine-specific representation. In some systems [ADGI08, WDR06], the translation from a pattern specification to a corresponding representation is a one-to-one mapping. Other frameworks [AcT08, MM09, REG11, SMMP09] introduce the notion of a cost-based evaluation plan, where multiple representations of $P$ are possible, and one is chosen according to the user's preference or some predefined cost metric. While normally NFA-based CEP evaluation mechanisms fall into the former category, the lazy NFA that we described in Chapter 2 is a representative of the latter type due to its out-of-order processing capabilities.

We will illustrate the above notions using the following example.

Example 3.1.1. Assume that we are receiving periodical readings from four traffic cameras $A$, $B$, $C$ and $D$. We are required to recognize a sequence of appearances of a particular vehicle on all four cameras in order of their position on a road, e.g., $A \rightarrow B \rightarrow C \rightarrow D$. Assume also that, due to a malfunction in camera $D$, it only transmits one frame for each 10 frames sent by the other cameras.

Figure 3.1(a) displays an eager NFA for detecting this pattern, as described in [WDR06]. A state is defined for each prefix of a valid match. During evaluation, a combination of camera readings matching each prefix will be represented by a unique instance of the NFA in the corresponding state. Transitions between states are triggered.
nondeterministically by the arrival of an event satisfying the constraints defined by the pattern. A new NFA instance is created upon each transition.

The structure of the above automaton is uniquely dictated by the order of events in the given sequence. However, replacing an eager NFA with a lazy Chain NFA would allow us to utilize any of the \((4!)\) possible evaluation plans, that is, mutual orders of \(A, B, C\) and \(D\). More specifically, due to the low transmission rate of \(D\) it would be beneficial to wait for its signal before examining the local history for previous readings of \(A, B\) and \(C\) that match the constraints. Figure 3.1(b) demonstrates a Chain NFA for the rewritten pattern. It starts by monitoring the rarest event type \(D\) and storing the other events in the dedicated buffer. As a reading from camera \(D\) arrives, the buffer is inspected for events from \(A, B\) and \(C\) preceding the one received from \(D\) and located in the same time window. This plan is more efficient than the one implicitly used by the first NFA in terms of the number of partial matches created during evaluation.

Not all CEP evaluation mechanisms represent a plan as an evaluation order. Figure 3.1(c) depicts a tree-based evaluation mechanism [MM09] for detecting the above pattern. Events are accepted at the corresponding leaves of the tree and passed towards the root where full matches are reported. This model requires an evaluation plan to be supplied, because, for a pattern of size \(n\), there are at least \(n! \cdot C_{n-1} = \frac{(2n-2)!}{(n-1)!}\) possible trees (where \(C_n\) is the \(n^{th}\) Catalan number) [Moe14].

In many scenarios, we will prefer the evaluation mechanisms supporting cost-based
Figure 3.2: Execution plans for a join of four relations $R_1, R_2, R_3, R_4$: (a) a left-deep tree; (b) a bushy tree.

plan generation (which we will refer to as plan-based evaluation mechanisms from now on) over those mechanisms allowing for only one such plan to be defined. This way, we can drastically boost system performance subject to selected metrics by picking more efficient plans. However, as the space of potential plans is at least exponential in pattern size, finding an optimal plan is not a trivial task.

Numerous authors have identified and targeted this issue. Some of the proposed solutions are based on rewriting the original pattern according to a set of predefined rules to maximize the efficiency of its detection [REG11, SMMP09]. Other approaches, including the one presented in Chapter 2 of this thesis, discuss various strategies and algorithms for generating an evaluation plan that maximizes the performance for a given pattern according to some cost function [AcT08, MM09]. While the above approaches demonstrate promising results, the applied methods often result in suboptimal plans. For example, a simple algorithm sorting the event types in a pattern in ascending order of frequency that we used in Chapter 2 might produce an extremely inefficient evaluation plan when a highly selective condition is defined. The research field of plan-based CEP evaluation mechanisms remains largely unexplored, and the space of the potential optimization techniques is still far from being exhausted.

The problem described above closely resembles the problem of estimating execution plans for large join queries. As opposed to CEP plan generation, this is a well-known, established, and extensively targeted research topic. A plethora of methods and approaches producing close-to-optimal results were published during the last few decades. These methods range from simple greedy heuristics, to exhaustive dynamic programming techniques, to randomized and genetic algorithms [KBZ86, LSC97, MN06, SAC+79, SMK97, Swa89]. Figure 3.2 illustrates two main types of execution plans for join queries, a left-deep tree (3.2(a)) and a bushy tree (3.2(b)).

Both problems look for a way to efficiently combine multiple data items such that some cost function is minimized. Also, both produce solutions possessing similar
structures. If we reexamine Figures 3.1 and 3.2, we can see that left-deep tree plans (3.2(a)) and bushy tree plans (3.2(b)) closely resemble evaluation plans for NFAs (3.1(b)) and trees (3.1(c)) respectively. An interesting question is whether join-related techniques can be used to create better CEP plans using a proper reduction.

In this chapter, we attempt to close the gap between the two areas of research. We study the relationship between CEP Plan Generation (CPG) and Join Query Plan Generation (JQPG) problems and show that any instance of CPG can be transformed into an instance of JQPG. Consequently, any existing method for JQPG can be made applicable to CPG. ¹

Our contributions in this chapter can be summarized as follows:

• We formally prove the equivalence of JQPG and CPG for a large subset of CEP patterns, the conjunctive patterns. The proof addresses the two major classes of evaluation plans, the order-based plans and the tree-based plans (Section 3.4).

• We extend the above result by showing how other pattern types can be converted to conjunctive patterns, thus proving that any instance of CPG can be reduced to an instance of JQPG (Section 3.5).

• The deployment of a JQPG method to CPG is not trivial, as multiple CEP-specific issues need to be addressed, such as detection latency constraints, event consumption policies, and adaptivity considerations. We present and discuss the steps essential for successful adaptation of JQPG techniques to the CEP domain (Section 3.6).

• We validate our theoretical analysis in an extensive experimental study. Several well-known JQPG methods, such as Iterative Improvement [Swa89] and Dynamic Programming [SAC+79], were applied on a real-world event dataset and compared to the existing state-of-the-art CPG mechanisms. The results demonstrate the superiority of the adapted JQPG techniques (Section 3.7).

3.2 Complex Event Patterns and Evaluation Mechanisms

In this section, we introduce the notations used throughout this chapter. We present the elements of a CEP pattern, including a brief taxonomy of commonly used pattern types. Then, we define and describe two classes of plan-based CEP evaluation mechanisms that are the focus of this work: the order-based and the tree-based CEP. The results obtained in the later sections are based on but not limited to these two representation models, and they can be extended to more complex schemes, such as event processing networks [EN10, REG11].

¹It should be noted that the above only applies to the stage of creating a plan for efficient query processing or pattern detection. The actual deployment of the resulting evaluation mechanism on an event stream proceeds as described in Chapter 2 regardless of the algorithm utilized for plan generation.
3.2.1 CEP Patterns

As presented in Chapter 2, a pattern is defined by a combination of primitive events, operators, a set of predicates to be satisfied by the participating events, and a time window. Each event is represented by a type and a set of attributes, including the occurrence timestamp. The operators describe the relations between different events comprising a pattern match. The predicates, usually organized in a Boolean formula, specify the constraints on the attribute values of the events. As an example, consider the following pattern specification syntax, taken from SASE [WDR06]:

\[
\begin{align*}
\text{PATTERN} & \quad \text{op} (T_1 e_1, T_2 e_2, \cdots, T_n e_n) \\
\text{WHERE} & \quad (c_{1,1} \land c_{1,2} \land \cdots \land c_{n,n-1} \land c_{n,n}) \\
\text{WITHIN} & \quad W.
\end{align*}
\]

Here, the PATTERN clause specifies the events \(e_1, \cdots, e_n\) we would like to detect and the operator \(\text{op}\) to combine them (see below). The WHERE clause defines a Boolean CNF formula of inter-event constraints, where \(c_{i,j}; 1 \leq i, j \leq n\) stands for the mutual condition between attributes of \(e_i\) and \(e_j\). \(c_{i,i}\) declares filter conditions on \(e_i\). Any of \(c_{i,j}\) can be empty. For the rest of this chapter, we assume that all conditions between events are at most pairwise (i.e., a single condition involves at most two different events). This assumption is for presentational purposes only, as our results can be easily generalized to arbitrary predicates. The WITHIN clause sets the time window \(W\), which is the maximal allowed difference between the timestamps of any pair of events in a match.

Throughout this work we assume that each primitive event has a well-defined type, i.e., the event either contains the type as an attribute or it can be easily inferred from other attributes using negligible system resources. While this constraint may seem limiting, it is easy to overcome in most cases by redefining what a pattern creator considers a type.

In this work, we will consider the most commonly used operators, namely AND, SEQ, and OR. The AND operator requires the occurrence of all events specified in the pattern within the time window. The SEQ operator extends this definition by also expecting the events to appear in a predefined temporal order. The OR operator corresponds to the appearance of any event out of those specified.

Two additional operators of particular importance are the negation operator (NOT) and the Kleene closure operator (KL). They can only be applied on a single event and are used in combination with other operators. \(\text{NOT}(e_i)\) requires the absence of the event \(e_i\) from the stream (or from a specific position in the pattern in the case of the SEQ operator), whereas \(\text{KL}(e_i)\) accepts one or more instances of \(e_i\). In the remainder of this chapter, we will refer to NOT and KL as \textit{unary operators}, while AND, SEQ and OR will be called \textit{n-ary operators}.

The PATTERN clause may include an unlimited number of n-ary and unary operators. We will refer to patterns containing a single n-ary operator, and at most a single unary
operator per primitive event, as simple patterns. On the contrary, nested patterns are allowed to contain multiple n-ary operators (e.g., a disjunction of conjunctions and/or sequences will be considered a nested pattern). Nested patterns present an additional level of complexity and require advanced techniques (e.g., as described in [LRD+11]).

We will further divide simple patterns into subclasses. A simple pattern whose n-ary operator is an AND operator will be denoted as a conjunctive pattern. Similarly, sequence pattern and disjunctive pattern will stand for patterns with SEQ and OR operators, respectively. In addition, a simple pattern containing no unary operators will be called a pure pattern.

The “four cameras pattern” described in Section 3.1 illustrates the above. This is a pure sequence pattern, written in SASE as follows:

```
PATTERN SEQ(A a, B b, C c, D d)
WHERE(a.vehicleID = b.vehicleID =
     = c.vehicleID = d.vehicleID)
WITHIN W.
```

The following example depicts a nested pattern, consisting of a (non-pure) conjunctive pattern and an inner pure disjunctive pattern:

```
PATTERN AND(A a, NOT(B b), OR(C c, D d))
WITHIN W.
```

### 3.2.2 Order-based Evaluation Mechanisms

Order-based evaluation mechanisms play an important role in CEP engines based on state machines. One of the most commonly used models following this principle is the NFA (Nondeterministic Finite Automaton) [ADGI08, DGP+07, WDR06], thoroughly discussed in Chapter 2. An NFA consists of a set of states and conditional transitions between them. Each state corresponds to a prefix of a full pattern match. Transitions are triggered by the arrival of the primitive events, which are then added to partial matches. Conditions between events are verified during the transitions. Figure 3.1(a) depicts an example of a NFA constructed for the “four cameras” sequence pattern. While in theory NFAs may possess an arbitrary topology, non-nested patterns are normally detected by a chain-like structure.

The basic NFA model does not include any notion of altering the “natural” evaluation order or any other optimization based on pattern rewriting. In Section 2.3, we presented a method for constructing NFAs with out-of-order processing support, which we denoted as lazy NFAs. The simplest form of a lazy NFA, a Chain NFA is capable of following a specified evaluation order. Given a pattern of n events and a user-specified order $O$ on the event types appearing in the pattern, a chain of $n + 1$ states is constructed, with each state $k$ corresponding to a match prefix of size $k - 1$. The order of the states matches $O$. If a type appears more than once in a pattern, it will also appear
multiple times in \( O \). The \((n + 1)^{th}\) state in the chain is the accepting state. To achieve out-of-order evaluation, incoming events are stored locally. A buffered event is retrieved and processed when its corresponding state in the chain is reached. Figure 3.1(b) presents an example of this construction for \( O = (D, A, B, C) \).

This construction method allows us to apply all possible \((n!)\) orders without affecting the detection correctness.

### 3.2.3 Tree-based Evaluation Mechanisms

An alternative to NFA, the tree-based evaluation mechanism [MM09] specifies which subsets of full pattern matches are to be tracked by defining tree-like structures. For each event participating in a pattern, a designated leaf is created. During evaluation, events are routed to their corresponding leaves and are buffered there. The non-leaf nodes accumulate the partial matches. The computation at each non-leaf node proceeds only when all of its children are available (i.e., all events have arrived or partial matches have been calculated). Matches formed at the tree root are reported to the end users. An example is shown in Figure 3.1(c).

The main advantage of the tree-based evaluation mechanism over NFA is its flexibility. Instead of relying on a single evaluation order, it allows primitive events to arrive in any possible order and still be efficiently processed.

The original ZStream paper [MM09] assumed a batch-iterator setting. To perform our study under a unified framework, we modify this behavior to support arbitrary time windows. As described above with regard to NFAs, a separate tree instance will be created for each currently found partial match. As a new event arrives, an instance will be created containing this event. Every instance \( I \) corresponds to some subtree \( s \) of the tree plan, with the leaves of \( s \) holding the primitive events in \( I \). Whenever a new instance \( I' \) is created, the system will attempt to combine it with previously created “siblings”, that is, instances corresponding to the subtree sharing the parent node with \( s' \). As a result, another new instance containing the unified subtree will be generated. This in turn will trigger the same process again, and it will proceed recursively until the root of the tree is reached or no siblings are found.

Contrary to NFA, ZStream includes an algorithm for determining the optimal tree structure for a given pattern. This algorithm is based on a cost model that takes into account the arrival rates of the primitive events and the selectivities of their predicates. However, since leaf reordering is not supported, a subset of potential plans is missed. We will illustrate this drawback using the following example:

\[
\text{PATTERN SEQ(A a, B b, C c) WHERE (a.x = c.x) WITHIN W.}
\]

We assume that all events arrive at identical rates, and that the condition between \( A \) and \( C \) is very restrictive. Figures 3.3(a) and 3.3(b) present the only two possible
Figure 3.3: Evaluation trees for a pattern $SEQ(A,B,C)$: (a) a left-deep tree produced by ZStream; (b) a right-deep tree produced by ZStream; (c) an optimal evaluation tree, which cannot be produced by ZStream.

plans according to the algorithm presented in [MM09]. However, due to the condition between $A$ and $C$, the most efficient evaluation plan is the one displayed in Figure 3.3(c). It will be shown later how join query optimization methods can be incorporated to overcome this issue.

3.3 Plan Generation Problems

This section defines and describes the two problems whose relationship will be closely studied in the subsequent sections. We start by presenting the CEP Plan Generation problem and explaining its two variations. Then, we briefly outline the Join Query Plan Generation problem.

3.3.1 CEP Plan Generation

We will start with the definition of the CEP evaluation plan. The evaluation plan provides a scheme for the evaluation mechanism, according to which its internal pattern representation is created. Therefore, different evaluation plans are required for different CEP frameworks. In this chapter, we distinguish between two main types of plans, the order-based plan and the tree-based plan.

An order-based plan consists of a permutation of the primitive event types declared...
by the pattern. An order-based CEP engine uses this plan to set the order in which events are processed at runtime. Order-based plans are applicable to mechanisms evaluating a pattern event-by-event, as described in Section 3.2.2.

A tree-based plan extends the above by providing a tree-like scheme for pattern evaluation. In this scheme, the structure of the internal nodes serves as a loose order in which events are to be matched. It specifies which subsets of valid matches are to be locally buffered and how to combine them into larger partial matches. Plans of this type are used by the evaluation mechanism presented in Section 3.2.3.

We can thus define two variations of the CEP Plan Generation problem, order-based CPG and tree-based CPG. In each variation, the goal is to determine an optimal evaluation plan \( P \) subject to some cost function \( \text{Cost}(P) \). Different CEP systems define different metrics to measure their efficiency. In this chapter we will consider a highly relevant performance optimization goal: reducing the number of active partial matches within the time window (denoted below simply as number of partial matches).

Regardless of the system-specific performance objectives, the implicit requirement to monitor all valid subsets of primitive events can become a major bottleneck. Because any partial match might form a full pattern match, their number is worst-case exponential in the number of events participating in a pattern. Further, as a newly arrived event needs to be checked against all (or most of) the currently stored partial matches, the processing time and resource consumption per event can become impractical for real-time applications. Other metrics, such as detection latency or network communication cost, may also be negatively affected. Thus, given the crucial role of the number of partial matches in all aspects of CEP, it was chosen as our primary cost function.

### 3.3.2 Join Query Plan Generation

Join Query Plan Generation is a well-known problem in query optimization [KBZ86, SAC+79, Swa89]. In this problem, we are given relations \( R_1, \ldots, R_n \) and a query graph describing the conditions to be satisfied by the tuples in order to be included in the result. A condition between a pair of relations \( R_i, R_j \) has a known selectivity \( f_{i,j} \in [0, 1] \) (we set \( f_{i,j} = 1 \) if no such condition is defined). The goal is to produce a join query plan, such that a predefined cost function defined on the plan space will be minimized.

One popular choice for the cost function is the number of intermediate tuples produced during plan execution. For the rest of this chapter, we will refer to it as the intermediate results size. In [CM95], the following expression is given to calculate this function for each two-way join of two input relations:

\[
C(R_i, R_j) = |R_i| \cdot |R_j| \cdot f_{i,j},
\]

where \( |R_i|, |R_j| \) are the cardinalities of the joined relations. This formula is naturally
extended to relations produced during join calculation:

$$C(S, T) = |S| \cdot |T| \cdot f_{S,T}.$$ 

Here, $S = R_{i_1} \bowtie \cdots \bowtie R_{i_k}$; $T = R_{j_1} \bowtie \cdots \bowtie R_{j_l}$ are the partial join results of some subsets of $R_1, \cdots, R_n$ and $f_{S,T} = (|S \bowtie T| / |S \times T|)$ is the product of selectivities defined between the individual relations comprising $S$ and $T$.

The two most popular classes of join query plans are the left-deep trees and the bushy trees. Algorithms based on the former type restrict their output to plans with a so-called “left-deep” topology. This type of join tree processes the input relations one-by-one, adding a new relation to the current intermediate result during each step. Hence, for this class of techniques, a valid solution is a join order rather than a join plan. Figure 3.2(a) depicts an example of a left-deep tree for a join query of four relations.

Approaches based on bushy trees pose no limitations on the plan topology, allowing it to contain arbitrary branches. An example is shown in Figure 3.2(b). Here a valid solution specifies a complete join tree rather than merely an order.

The JQPG problem was shown by multiple authors to be NP-complete [CM95, IK84], even when only left-deep trees are considered.

### 3.4 The Equivalence of CPG and JQPG for Pure Conjunctive Patterns

This section presents the formal proof of equivalence between CPG and JQPG for pure conjunctive patterns. We show that, when the pattern to be monitored is a pure conjunctive pattern and the CPG cost function represents the number of partial matches, the two problems are equivalent. From this result, we deduce the NP-completeness of CPG.

#### 3.4.1 Order-Based Evaluation

We will first focus on a CPG variation for order-based evaluation plans. In this section we will show that this problem is equivalent to JQPG restricted to left-deep trees. To that end, we will define the cost model functions for both problems and then present the equivalence theorem.

Our cost function $Cost_{ord}$ will reflect the number of partial matches coexisting in memory within the time window. As we discussed in Section 3.3.1, this cost model reflects a variety of performance metrics such as the system throughput, memory consumption, and detection latency. The calculations will be based on the arrival rates of the events and the selectivities of the predicates.

Let $sel_{i,j}$ denote the selectivity of $c_{i,j}$, i.e., the probability of a partial match containing instances of events of types $T_i$ and $T_j$ to pass the condition. Additionally,
let $r_1, \cdots r_n$ denote the arrival rates of corresponding event types $T_1, \cdots T_n$. Then, the expected number of primitive events of type $T_i$ arriving within the time window $W$ is $W \cdot r_i$. Let $O = (T_{p_1}, T_{p_2}, \cdots T_{p_n}); p_i \in [1, n]$ denote an execution order. Then, during pattern evaluation according to $O$, the expected number of partial matches of length $k, 1 \leq k \leq n$ is given by:

$$PM(k) = W^k \cdot \prod_{i=1}^{k} r_{p_i} \cdot \prod_{i,j \leq k; i \leq j} sel_{p_i, p_j}.$$ 

The overall cost function we will attempt to minimize is thus the sum of partial matches of all sizes, as follows:

$$Cost_{ord}(O) = \sum_{k=1}^{n} \left( W^k \cdot \prod_{i=1}^{k} r_{p_i} \cdot \prod_{i,j \leq k; i \leq j} sel_{p_i, p_j} \right).$$

For the JQPG problem restricted to output left-deep trees only, we will use the two-way join cost function $C(S, T)$ defined in Section 3.3.2. Let $L$ be a left-deep tree and let $\{i_1, i_2, \cdots, i_n\}$ be the order in which input relations are to be joined according to $L$. Let $P_k, 1 \leq k < n$ denote the result of joining the first $k$ tables by $L$ (that is, $P_1 = R_{i_1}, P_2 = R_{i_1} \bowtie R_{i_2}$, etc.). In addition, let $C_1 = |R_{i_1}| \cdot f_{i_1, i_1}$ be the cost of the initial selection from $R_{i_1}$. Then, the cost of $L$ will be defined according to a left-deep join (LDJ) cost function:

$$Cost_{LDJ}(L) = C_1 + \sum_{k=2}^{n} C(P_{k-1}, R_{i_k}).$$

We are now ready to formally prove the statement formulated in the beginning of the section.

**Theorem 3.1.** Given a pure conjunctive pattern $P$, the problem of finding an order-based evaluation plan for $P$ minimizing $Cost_{ord}$ is equivalent to the Join Query Plan Generation problem for left-deep trees subject to $Cost_{LDJ}$.

We will start by proving that $CPG \subseteq JQPG$, i.e., given an instance of the CPG problem satisfying the aforementioned conditions, there exists a corresponding instance of the JQPG problem, such that the solutions for the two problem instances have an identical structure. To that end, we will present the corresponding reduction.

Given a pure conjunctive pattern $P$ defined as follows:

```
PATTERN AND (T_1 e_1, \cdots T_n e_n)
WHERE (c_{1,1} \land c_{1,2} \land \cdots \land c_{n-1,n} \land c_{n,n})
WITHIN W,
```

let $R_1, \cdots, R_n$ be a set of relations such that each $R_i$ corresponds to an event type $T_i$. 55
For each attribute of $T_i$, including the timestamp, a matching column will be defined in $R_i$. The cardinality of $R_i$ will be set to $W \cdot r_i$, and, for each predicate $c_{i,j}$ with selectivity $sel_{i,j}$, an identical predicate will be formed between the relations $R_i$ and $R_j$. We will define the query corresponding to $P$ as follows:

\[
SELECT * \\
FROM R_{1}, \ldots, R_n \\
WHERE (c_{1,1} \text{ AND } \cdots \text{ AND } c_{n,n}).
\]

We will show that a solution to this instance of the JQPG problem is also a solution to the initial CPG problem. Recall that a left-deep JQPG solution $L$ minimizes the function $Cost_{LDJ}$. By opening the recursion and substituting the parameters with those of the original problem, we get:

\[
Cost_{LDJ}(L) = C_1 + \sum_{k=2}^{n} C(P_{k-1}, R_{ik}) = |R_{i1}| \cdot f_{i1,i1} + \sum_{k=2}^{n} \left( \prod_{j=1}^{k} |R_{ij}| \cdot \prod_{j,k \leq j \leq l} f_{ij,ii} \right) = \sum_{k=1}^{n} \left( \prod_{j=1}^{k} (W \cdot r_{ij}) \cdot \prod_{j \leq k-1, j \leq l} sel_{ij,ii} \right) = Cost_{ord}(O).
\]

Consequently, the solution that minimizes $Cost_{LDJ}$ also minimizes $Cost_{ord}$, which completes the proof of the first direction of the theorem.

We will now proceed to proving the second direction, i.e., $JQPG \subseteq CPG$. Let $R_1, \ldots, R_n$ be the relations to be joined with mutual predicates $F_{i,j}$ of selectivities $f_{i,j}$. We will demonstrate a decomposition of this problem to an instance of the CPG problem.

Let $T_1, \ldots, T_n$ be primitive event types such that each $T_i$ corresponds to a relation $R_i$. Let each instance of $T_i$ have the attributes identical to the columns of $R_i$. In addition, let $S$ be an input stream containing a primitive event $t_{i,k}$ of type $T_i$ for each tuple $k$ of a relation $R_i$, where $1 \leq k \leq |R_i|$ is an index of a tuple in a relation. Let the timestamp of each such event be defined as $t_{i,k,ts} = k^2$. Define the time window as $W = \max |R_i|$, where $|R_i|$ is the cardinality of $R_i$. Finally, let $r_i$, the arrival rate of events of type $T_i$, be equal to $\frac{|R_i|}{W}$.

Now we are ready to define a CEP conjunctive pattern corresponding to the join of

\footnote{Since our CEP model allows multiple events to have the same timestamp, no violation is introduced by this definition.}
We will show that a solution to this instance of the CPG problem is also a solution to the initial JQPG problem. Recall that such a solution minimizes the function Cost\textsubscript{ord}. By substituting the parameters with those of the original problem, we get:

\[
\text{Cost}_{\text{ord}}(O) = \sum_{k=1}^{n} \left( W^k \cdot \prod_{i=1}^{k} \left| R_{p_i} \right| \cdot \prod_{i,j \leq k; i \leq j} f_{p_i,p_j} \right) = \sum_{k=1}^{n} \left( \prod_{i=1}^{k-1} \left| R_{p_i} \right| \cdot \prod_{i,j \leq k-1; i \leq j} f_{p_i,p_j} \right) \cdot \left| R_{p_k} \right| \cdot \prod_{i \leq k} f_{p_i,p_k}.
\]

Here, the first sub-expression inside the summation is the intermediate results size until the point when the \(k\)th relation is to be joined, as follows from the definition above. The second sub-expression is the cardinality of the relation \(R_{p_k}\), and the third sub-expression is the product of all predicates to be applied on tuples from \(R_{p_k}\) upon joining it with \(R_{p_1}, \ldots, R_{p_{k-1}}\). Therefore, the resulting expression is identical to \(\text{Cost}_{LDJ}\) (in particular, the first element of the summation equals \(C_1\)). Consequently, the solution that minimizes \(\text{Cost}_{\text{ord}}\) also minimizes \(\text{Cost}_{LDJ}\), which completes the proof.

In [CM95] the authors showed the problem of Join Query Plan Generation for left-deep trees to be NP-complete for the general case of arbitrary query graphs. From this result and from the above theorem we will deduce the following corollary.

**Corollary 3.2.** The problem of finding an order-based evaluation plan for a general pure conjunctive complex event pattern that minimizes \(\text{Cost}_{\text{ord}}\) is NP-complete.

### 3.4.2 Tree-Based Evaluation

In this section, we will extend Theorem 3.1 to tree-based evaluation plans. This time the reduction will be performed from the unrestricted JQPG problem, allowed to return bushy trees. Similarly to the previous section, we will start by defining the cost functions and then proceed to the proof of the extended theorem.

We will define the cost model for evaluation trees in a manner similar to Section 3.4.1. We will estimate the number of partial matches accumulated in each node of the evaluation tree and sum them up to produce the cost function.

For a leaf node \(l\) collecting events of type \(T_i\), the expected number of partial matches is equal to the number of events of type \(T_i\) arriving inside a time window:

\[
PM(l) = W \cdot r_i.
\]
To obtain an estimate for an internal node \(in\), we multiply the cost function values of its children by the total selectivity of the predicates verified by this node:

\[
PM(in) = PM(in.left) \cdot PM(in.right) \cdot SEL_{LR}(in),
\]

where \(SEL_{LR}\) is the selectivity of the predicates defined between event types accepted at the left and the right sub-trees of node \(in\), or, more formally:

\[
SEL_{LR}(in) = \prod_{e_i \in in.ltree; e_j \in in.rtree} sel_{i,j}.
\]

The total cost function on a tree \(T\) is thus defined as follows:

\[
Cost_{tree}(T) = \sum_{N \in \text{nodes}(T)} PM(N).
\]

For bushy trees, we will extend the cost function defined in Section 3.4.1. The cost of a tree node \(N\) will be defined as follows:

\[
C(N) = \begin{cases} 
|R_i| & \text{N is a leaf representing } R_i \\
|L| \cdot |R| \cdot f_{L,R} & \text{N is an internal node representing a sub-join } L \bowtie R,
\end{cases}
\]

with the bushy join (BJ) cost function defined as follows:

\[
Cost_{BJ}(T) = \sum_{N \in \text{nodes}(T)} C(N).
\]

We will now extend Theorem 3.1 to tree-based plans.

**Theorem 3.3.** Given a pure conjunctive pattern \(P\), the problem of finding a tree-based evaluation plan for \(P\) minimizing \(Cost_{tree}\) is equivalent to the Join Query Plan Generation problem subject to \(Cost_{BJ}\).

To prove the theorem, we decompose each of the tree cost functions \(Cost_{tree}, Cost_{BJ}\) defined above for the CPG and the JQPG problems into two components, separately.
calculating the cost of the leaves and the internal nodes:

\[
\begin{align*}
\text{Cost}^l_{\text{tree}} (T) &= \sum_{N \in \text{leaves}(T)} PM(N) \\
\text{Cost}^{\text{in}}_{\text{tree}} (T) &= \sum_{N \in \text{in nodes}(T)} PM(N) \\
\text{Cost}^l_{BJ} (T) &= \sum_{N \in \text{leaves}(T)} C(N) \\
\text{Cost}^{\text{in}}_{BJ} (T) &= \sum_{N \in \text{in nodes}(T)} C(N).
\end{align*}
\]

Obviously, the following equalities hold:

\[
\begin{align*}
\text{Cost}_{\text{tree}} (T) &= \text{Cost}^l_{\text{tree}} (T) + \text{Cost}^{\text{in}}_{\text{tree}} (T) \\
\text{Cost}_{BJ} (T) &= \text{Cost}^l_{BJ} (T) + \text{Cost}^{\text{in}}_{BJ} (T).
\end{align*}
\]

Thus, it is sufficient to prove that

\[
\begin{align*}
\text{Cost}^l_{\text{tree}} (T) &= \text{Cost}^l_{BJ} (T) \\
\text{Cost}^{\text{in}}_{\text{tree}} (T) &= \text{Cost}^{\text{in}}_{BJ} (T)
\end{align*}
\]

for every \( T \). From here it will follow that the solution minimizing \( \text{Cost}_{\text{tree}} \) will also minimize \( \text{Cost}_{BJ} \) and vice versa.

Applying either direction of the reduction from Theorem 3.1, we observe the following for the first pair of functions:

\[
\begin{align*}
\text{Cost}^l_{\text{tree}} (T) &= \sum_{N \in \text{leaves}(T)} PM(N) = \sum_{i=1}^{n} W \cdot r_i = \\
&= \sum_{i=1}^{n} |R_i| = \sum_{N \in \text{leaves}(T)} C(N) = \text{Cost}^l_{BJ} (T).
\end{align*}
\]

Similarly, for the second pair of functions:

\[
\text{Cost}^{\text{in}}_{\text{tree}} (T) = \sum_{N \in \text{in nodes}(T)} PM(N) = \sum_{N \in \text{in nodes}(T)} PM(N.left) \cdot PM(N.right) \cdot SEL_{LR}(N).
\]

Opening the recursion, we get:

\[
\text{Cost}^{\text{in}}_{\text{tree}} (T) = \sum_{N \in \text{in nodes}(T)} \left( \prod_{m \in \text{leaves}(N)} W \cdot r_m \cdot \prod_{i,j \in \text{leaves}(N)} sel_{i,j} \right).
\]
By applying an identical transformation on \( \text{Cost}_{JB}^n \) we obtain the following:

\[
\text{Cost}_{BJ}^n (T) = \sum_{N \in \text{in_nodes}(T)} \left( \prod_{m \in \text{leaves}(N)} |R_m| \cdot \prod_{i,j \in \text{leaves}(N)} f_{i,j} \right).
\]

After substituting \( r_m = \frac{|R_m|}{W} \) and \( \text{sel}_{p_i,p_j} = f_{p_i,p_j} \), the two expressions are identical, which completes the proof. □

By Theorem 3.3 and the generalization of the result in [CM95], we derive the following corollary.

**Corollary 3.4.** The problem of finding a tree-based evaluation plan for a general pure conjunctive complex event pattern that minimizes \( \text{Cost}_{\text{tree}} \) is NP-complete.

### 3.4.3 Join Query Types

As Corollaries 3.2 and 3.4 imply, no efficient algorithm can be devised to optimally solve CPG for a general conjunctive pattern unless \( P = NP \). However, better complexity results may be available under certain assumptions regarding the pattern structure. Numerous works considered the JQPG problem for restricted *query types*, that is, specific topologies of the query graph defining the inter-relation conditions. Examples of such topologies include clique, tree, and star.

It was shown in [IK84, KBZ86] that an optimal plan can be computed in polynomial time for left-deep trees and queries forming an acyclic graph (i.e., tree queries), provided that the cost function has the ASI (adjacent sequence interchange) property [MS79]. The left-deep tree cost function \( \text{Cost}_{LDJ} \) has this property [CM95], making the result applicable for our scenario. A polynomial algorithm without the ASI requirement was proposed for bushy tree plans for chain queries [Orl92]. From Theorems 3.1 and 3.3 we can conclude that, for conjunctive patterns only, CPG∈P under the above constraints.

However, these results only hold when the plans produced by a query optimizer are not allowed to contain cross products [CM95, Orl92]. While this limitation is well-known in relational optimization [VM96], it is not employed by the existing CPG methods [AcT08, MM09, SMMP09]. Moreover, it was shown that when cross products are omitted, cheaper plans might be missed [OL90]. Thus, even when an exact polynomial algorithm is applicable to CPG, it is inferior to native algorithms in terms of the considered search space and can only be viewed as a heuristic. In that sense, it is similar to the greedy and randomized approaches [SMK97, Swa89].

Other optimizations utilizing the knowledge of the query type were proposed. For example, the optimal bushy plan was empirically shown to be identical to the optimal left-deep plan for star queries and, in many cases, for grid queries [SMK97]. This observation allows us to utilize a cheaper left-deep algorithm for the above query types without compromising the quality of the resulting plan.
With the introduction of additional pattern types (Section 3.5) and event selection strategies (Section 3.6.2), new query graph topologies might be identified and type-specific efficient algorithms designed. This topic is beyond the scope of this work and is a subject for future research.

Although not used directly by the JQPG algorithms, the order-based CPG cost functions $Cost_{ord}$ and $Cost_{ord}^{lat}$ (that we will introduce in Section 3.6.1) also have the ASI property. We formally prove this statement in Appendix B.

### 3.5 JQPG for General Pattern Types

The reduction from CPG to JQPG presented above only applies to pure conjunctive patterns. However, the patterns encountered in real-world scenarios are much more diverse. To complete the solution, we have to consider simple patterns containing SEQ, OR, NOT and KL operators. We also have to address nested patterns.

This section describes how a pattern of each of the aforementioned types can be represented and detected as either a pure conjunctive pattern or their union. To that end, we will utilize some of the ideas from Section 2.4. Note that the transformations presented below are only applied for the purpose of plan generation, that is, no actual conversion takes place during execution on a data stream.

#### 3.5.1 Sequence patterns

We observe that a sequence pattern is merely a conjunctive pattern with additional temporal constraints, i.e., predicates on the values of the timestamp attribute. Thus, a general pure sequence pattern of the form

```latex
PATTERN SEQ (T_1 e_1, T_2 e_2, \cdots, T_n e_n)
WHERE (c_{1,1} \land c_{1,2} \land \cdots \land c_{n,n-1} \land c_{n,n})
WITHIN W
```

can be rewritten in the following way without any semantic change:

```latex
PATTERN AND (T_1 e_1, T_2 e_2, \cdots, T_n e_n)
WHERE (c_{1,1} \land \cdots \land c_{n,n}\land
\land (e_1.ts < e_2.ts) \land \cdots \land (e_{n-1}.ts < e_n.ts))
WITHIN W.
```

An instance of the sequence pattern is thus reduced from CPG to JQPG similarly to a conjunctive pattern, with the timestamp column added to each relation $R_i$ representing an event type $T_i$, and constraints on the values of this column introduced into the query representation.

We will now formally prove the correctness of the above construction.
**Theorem 3.5.** Let $S$ be a pure sequence pattern specified by primitive event types $T_1, \ldots, T_n$, a Boolean predicate of constraints $P_S = \bigwedge_{1 \leq i, j \leq n} c_{i,j}$, and a time window $W$. Additionally, let $C$ be a pure conjunctive pattern specified by $T_1, \ldots, T_n$, a time window $W' = W$, and a predicate $P_C = P_S \land P_O$, where $P_O = \bigwedge_{1 \leq i < n} (e_i.ts < e_{i+1}.ts)$. Then, $S$ is equivalent to $C$, i.e., both patterns specify the same set of matches when applied on the same stream of events.

We will prove this theorem by double inclusion.

$S \subseteq C$: Let $M = \{e_1, \ldots, e_n\}$ be a match for $S$. Then, by definition, $M$ satisfies $P_S$. In addition, since $S$ is a sequence pattern, events in $M$ follow the temporal order $e_1.ts < e_2.ts < \cdots < e_n.ts$. Hence, $M$ also satisfies $P_O$, and thus $P_C$ as well, i.e., $M$ is a match for $C$.

$C \subseteq S$: Let $M = \{e_1, \ldots, e_n\}$ be a match for $C$. Since $M$ satisfies $P_O$, its events satisfy the ordering constraints $e_1.ts < \cdots < e_n.ts$. By definition of a sequence pattern, $\{e_1, \ldots, e_n\}$ form a valid sequence within the time window $W$. Additionally, $M$ satisfies $P_S$. Hence, by definition, $M$ is a match for $S$.\[\blacksquare\]

### 3.5.2 Kleene closure patterns

In a pattern with an event type $T_i$ under a KL operator, any subset of events of $T_i$ within the time window can participate in a match. During plan generation, we are interested in modeling this behavior in a way comprehensible by a JQPG algorithm, that is, using an equivalent pattern without Kleene closure. To that end, we introduce a new type $T'_i$ to represent all event subsets accepted by $KL(T_i)$, that is, the power set of events of $T_i$. A set of $k$ events of type $T_i$ will be said to contain $2^k$ “events” of type $T'_i$, one for each subset of the original $k$ events. The new pattern is constructed by replacing $KL(T_i)$ with $T'_i$. Since a time window of size $W$ contains $2^{r_i \cdot W}$ subsets of $T_i$ (where $r_i$ is the arrival rate of $T_i$), the arrival rate $r'_i$ of $T'_i$ is set to $2^{r_i \cdot W}$. The predicate selectivities remain unchanged.

For example, given the following pattern with the arrival rate of 5 events per second for each event type:

\[
\text{PATTERN AND}(A \ a, KL(B \ b), C \ c) \\
\text{WHERE} \ (true) \ \text{WITHIN} \ 10 \ \text{seconds},
\]

the pattern to be utilized for plan generation will be:

\[
\text{PATTERN AND}(A \ a, B' \ b, C \ c) \\
\text{WHERE} \ (true) \ \text{WITHIN} \ 10 \ \text{seconds}.
\]

The arrival rate of $B'$ will be calculated as $r'_B = \frac{2^{r \cdot W}}{W} = 10 \cdot 2^{50}$. A plan generation algorithm will then be invoked on the new pattern. Due to an extremely high arrival...
rate of $B'$, its processing will likely be postponed to the latest step in the plan, which is also the desired strategy for the original pattern in this case. $B'$ will then be replaced with $B$ in the resulting plan, and the missing Kleene closure operator will be added in the respective stage (by modifying an edge type for a NFA (Section 2.4.5) or a node type for a tree [MM09]), thus producing a valid plan for detecting the original pattern.

We will now formally prove the correctness of the above construction.

**Theorem 3.6.** Let $K$ be a conjunctive pattern specified by primitive event types $T_1, \cdots, T_n$, a Boolean predicate of constraints $P_K = \bigwedge_{1 \leq i, j \leq n} c_{i,j}$, and a time window $W$. In addition, let $K$ contain a Kleene closure operator applied on an event type $T_i$. Let a new type $T'_i$ represent the power set of events of the type $T_i$, i.e., for each set of events $\{e^1_i, e^2_i, \cdots, e^k_i\}$ of type $T_i$ within the time window an event of type $T'_i$ is created, containing each of $e^j_i$ as an attribute. Let $C$ be a pure conjunctive pattern specified by $T_1, \cdots, T_i', \cdots, T_n$, a predicate $P_C = P_K$, and a time window $W' = W$. Then, $K$ is equivalent to $C$, i.e., both patterns specify the same set of matches when applied on the same stream of events.

We will show that for an arbitrary input stream both patterns will produce identical sets of pattern matches. Let $S = \{s_1, \cdots, s_m\}$ be an input stream. W.l.o.g., assume that each event in $S$ belongs to one of the types $T_1, \cdots, T_n$ and that all events are within the time window $W$. Additionally, let $S' \subset S$ denote the set of all events of type $T_i$ in $S$. Then, while monitoring the pattern $K$, the system will create $2^{|S'|} - 1$ partial matches for each unique combination of events of types $T_1, \cdots, T_i-1, T_i+1, \cdots, T_n$.

Now, let $S'' \subset S$ denote the set of all events of type $T'_i$ in $S$. While monitoring $C$, for each combination of events of types $T_1, \cdots, T_{i-1}, T_{i+1}, \cdots, T_n$ the detecting framework will create $|S''|$ partial matches, one for each primitive event in $S''$. For each non-empty subset of $S'$, $S''$ contains an event corresponding to this subset, and vice versa. As the constraints on both patterns are identical, as well as the time window, the resulting full matches will also be the same.

**Corollary 3.7.** Theorem 3.6 holds also for sequence patterns.

The correctness of this corollary follows from the transitivity of conversions in Theorems 3.5 and 3.6.

**Corollary 3.8.** Theorem 3.6 and Corollary 3.7 hold for an arbitrary number of non-nested Kleene closure operators in a pattern.

The proof of this corollary is by iteratively applying Theorem 3.6 or Corollary 3.7 on each Kleene closure operator.

### 3.5.3 Negation patterns
Patterns with a negated event will not be rewritten. Instead, we will introduce a negation-aware evaluation plan creation strategy. First, a plan will be generated for a positive part of a pattern as described above (we assume that a pattern always contains a non-empty positive part). Then, a check for the appearance of a negated event will be added at the earliest point possible, when all positive events it depends on are already received. This construction process will be implemented by augmenting a plan with a transition to the rejecting state for a NFA (Section 2.4.4) or with a NSEQ node for a ZStream tree [MM09]. For example, given a pattern \( SEQ(A, NOT(B), C, D) \), the existence of a matching \( B \) in the stream will be tested immediately after the latest of \( A \) and \( C \) have been accepted. Since both Lazy NFA and ZStream incorporate event buffering, this technique is feasible and easily applicable.

3.5.4 Nested patterns

Patterns of this type can contain an unlimited number of n-ary operators. After transforming \( SEQ \) to \( AND \) as shown above, we are left with only two such operator types, \( AND \) and \( OR \). Given a nested pattern, we convert the pattern formula to DNF form, that is, an equivalent nested disjunctive pattern containing a list of simple conjunctive patterns is produced. Then, a separate evaluation plan is created for each conjunctive subpattern, and their detection proceeds independently. The returned result is the union of all subpattern matches.

Note that applying the DNF transformation can cause some expressions to appear in multiple subpatterns. For example, a nested pattern of the form \( AND(A, B, OR(C, D)) \) will be converted to a disjunction of conjunctive patterns \( AND(A, B, C) \) and \( AND(A, B, D) \). As a result, redundant computations will be performed by automata or trees corresponding to different subpatterns (comparing \( A \)'s to \( B \)'s in our example). This problem can be solved by applying known multi-query techniques for shared subexpression processing, such as those described in [DGH+06, LRG+11, RLR16, RRL+13, ZVDH17].

3.6 Adapting JQPG Algorithms to Complex Event Processing

The theoretical results from previous sections have two important implications. First, unless \( P = NP \), no efficient algorithm can be devised to optimally solve CPG. On a positive note, the second implication is that any existing technique for determining a close-to-optimal execution plan for a join query can be adapted and used in CEP applications.

However, many challenges arise when attempting to perform this transformation procedure in practice. First, despite the benefits of the cost function introduced in Section 3.3.1, simply counting the partial matches is not always sufficient. Additional
performance metrics are often essential, such as the average response time. Second, com-
plex event specification languages contain various constructs not present in traditional
databases, such as event selection strategies. Third, the arrival rates of event types and
the predicate selectivities are rarely obtained in advance and can change rapidly over
time. A solution must be devised to measure the desired statistics on-the-fly and adapt
the evaluation plan accordingly.

In this section, we show how detection latency and event selection strategies can be
incorporated into existing JQPG algorithms. We also address the problem of adapting
to dynamic changes in the input stream.

3.6.1 Pattern Detection Latency

Latency is commonly defined as a time difference between the arrival of the last event
comprising a full pattern match and the time of reporting this match. As many
existing applications involve strong real-time requirements, pattern detection latency
is an important optimization goal for CEP systems. Unfortunately, in most cases
it is impossible to simultaneously achieve maximal throughput and minimal latency.
Trade-offs between the two are widely studied in the CEP context [AcT08, YLW16].

Detection schemes utilizing out-of-order evaluation, like those discussed in this work,
often suffer from increased latency as compared to simpler approaches. The main reason
is that, when an execution plan is optimized for maximal throughput, the last event
in the pattern may not be the last event in the plan. After this event is accepted,
the evaluation mechanism still needs to walk through the remaining part of the plan,
resulting in late detection of the full match.

Algorithms adopted from JQPG are not normally aimed at the optimization goal of
minimizing latency. However, since they are generally independent of the cost model,
this problem can be solved by providing an appropriate cost function. In addition to
functions presented in Sections 3.4.1 and 3.4.2, which we will refer to as Cost_trpt and
Cost_tree, a new pair of functions, Cost_lat and Cost_tree, will reflect the expected latency
of a plan. To combine the functions, many existing multi-objective query optimization
techniques can be used, e.g., pareto optimal plan calculation [AcT08] or parametric
methods [TK16]. Systems with limited computational resources may utilize simpler and
less expensive solutions, such as defining the total cost function as a weighted sum of
its two components:

\[
\text{Cost}(\text{Plan}) = \text{Cost}_{\text{trpt}}(\text{Plan}) + \alpha \cdot \text{Cost}_{\text{lat}}(\text{Plan}),
\]

where \(\alpha\) is a user-defined parameter adjusted to fit the required throughput-latency
trade-off. This latter model was used during our experiments (Section 3.7).

We will now formally define the latency cost functions. For a sequence pattern, let \(T_n\)
denote the last event type in the order induced by the pattern. Then, for an order-based
plan \(O\), let \(\text{Succ}_O(T_n)\) denote the event types succeeding \(T_n\) in \(O\). Following the arrival
of an event of type $T_n$, in the worst case we need to examine all locally buffered events of types in $\text{Succ}_O(T_n)$. As defined in Section 3.4.1, there are $W \cdot r_i$ such events of type $T_i$, hence:

$$\text{Cost}_{\text{ord}}^{\text{lat}}(O) = \sum_{T_i \in \text{Succ}_O(T_n)} W \cdot r_i.$$ 

Similarly, for a tree-based plan $T$, let $\text{Anc}_T(T_n)$ denote all ancestor nodes of the leaf corresponding to $T_n$ in $T$, i.e., nodes located on a path from $T_n$ to the root (excluding the root). Let us examine the traversal along this path. When an internal node $N$ with two children $L$ and $R$ receives a partial match from, say, the child $L$, it compares this match to all partial matches currently buffered on $R$. Thus, the worst-case detection latency of a sequence pattern ending with $T_n$ is proportional to the number of partial matches buffered on the siblings of the nodes in $\text{Anc}_T(T_n)$. More formally, let $\text{sibling}(N)$ denote the other child of the parent of $N$ (for the root this function will be undefined). Then,

$$\text{Cost}_{\text{tree}}^{\text{lat}}(T) = \sum_{N \in \text{Anc}_T(T_n)} \text{PM}(\text{sibling}(N)).$$

For a conjunctive pattern, estimating the detection latency is a more difficult problem, as the last arriving event is not known in advance. One possible approach is to introduce a new system component, called the output profiler. The output profiler examines the full matches reported as output and records the most frequent temporal orders in which primitive events appear. Then, as enough information is collected, the latency function may be defined as in the previous case, subject to the event arrival order with the highest probability of appearance.

Finally, for a disjunctive pattern, we define the latency cost function as the maximum over the disjunction operands. This definition applies also for arbitrary nested patterns.

### 3.6.2 Event Selection Strategies

In addition to event types, operators and predicates, CEP patterns are further defined using the event selection strategies [ADGI08, CM12b, EN10]. An event selection strategy specifies how events are selected from an input stream for partial matches. In this section, we discuss four existing strategies and show how a reduction from JQPG to CPG can support them.

Until now, we have implicitly assumed the skip-till-any-match selection strategy [ADGI08], which permits a primitive event to participate in an unlimited number of matches. This strategy is the most flexible, as it allows all possible combinations of events comprising a match to be detected. However, some streaming applications do not require such functionality. Thus, additional strategies were defined, restricting the participation of an event in a match.

The skip-till-next-match selection strategy [ADGI08] limits an event to appear in at most a single full match. This is enforced by “consuming” events already assigned
to a match. While this strategy prevents some matches from being discovered, it also considerably simplifies the detection process. In a CEP system operating under the skip-till-any-match policy, our cost model will no longer provide a correct estimate for a number of partial matches, which would lead to arbitrarily inefficient evaluation plans. However, since most JQPG algorithms do not depend on a specific cost function, we can solve this issue by replacing Cost\textsubscript{ord} and Cost\textsubscript{tree} with newly devised models.

Let us examine the number of partial matches in an order-based setting under the skip-till-next-match strategy. We will denote by \( m[k] \) the number of matches of size \( k \) expected to exist simultaneously in a time window. Obviously, \( m[1] = W \cdot r_{p_1} \), where \( T_{p_1} \) is the first event type in the selected evaluation order. For the estimate of \( m[2] \), there are two possibilities. If \( r_{p_1} > r_{p_2} \), there will not be enough instances of \( T_{p_2} \) to match all existing instances of \( T_{p_1} \), and some of the existing matches of size 1 will never be extended. Hence, \( m[2] = W \cdot r_{p_2} \) in this case. Otherwise, as an existing partial match cannot be extended by more than a single event of type \( T_{p_2} \), \( m[1] \) will be equal to \( m[2] \). In addition, if a mutual condition exists between \( T_{p_1} \) and \( T_{p_2} \), the resulting expression has to be multiplied by \( sel_{p_1,p_2} \).

By extending this reasoning to an arbitrary partial match, we obtain the following expression:

\[
  m[k] = W \cdot \min (r_{p_1}, r_{p_2}, \ldots, r_{p_k}) \cdot \prod_{i,j \leq k; i \leq j} sel_{p_i,p_j}; \quad 1 \leq k \leq n.
\]

And the new cost function for order-based CPG is

\[
  \text{Cost}_{\text{next}}^{\text{ord}} (O) = \sum_{k=1}^{n} (W \cdot m[k]).
\]

Using similar observations, we extend the above result for the tree-based model:

\[
  PM(n) = W \cdot \min_{T_i \in \text{subtree}(n)} (r_i) \cdot \prod_{T_i,T_j \in \text{subtree}(n)} sel_{i,j}; \nonumber
\]

\[
  \text{Cost}_{\text{next}}^{\text{tree}} (T) = \sum_{n \in \text{nodes}(T)} PM(n).
\]

The two remaining selection strategies, strict contiguity and partition contiguity [ADGI08], further restrict the appearance of events in a match. The strict contiguity requirement forces the selected events to be contiguous in the input stream, i.e., it allows no other events to appear in between. The partition contiguity strategy is a slight relaxation of the above. It partitions the input stream according to some condition and only requires the events located in the same partition to be contiguous.

To support JQPG-based solutions for CPG under strict or partition contiguity, we will explicitly model the constraints imposed by the above strategies. In addition, the
cost model presented earlier for skip-till-next-match will be used for both selection strategies.

To express strict contiguity, we will augment each primitive event with a new attribute reflecting its unique serial number in the stream. Then, we will add a new condition for each pair of potentially neighboring events, requiring the numbers to be adjacent.

For partition contiguity, the new attribute will represent an inner, per-partition order rather than a global one. Unless the partitioning condition is very costly to evaluate (which is rarely the case), this transformation can be efficiently and transparently applied on the input stream. The new contiguity condition will first compare the partition IDs of the two events, and only verify their serial numbers if the IDs match. We assume that the value distribution across the partitions remains unchanged. Otherwise, the evaluation plan is to be generated on a per-partition basis. Techniques incorporating per-partition plans are beyond the scope of this work and are a subject for our future research.

3.6.3 Adaptive Complex Event Processing

As their definition implies, JQPG algorithms can only be used when event arrival rates and predicate selectivities are given in advance. However, in real-life scenarios this a priori knowledge is rarely available. Moreover, the data characteristics are subject to frequent on-the-fly fluctuations. To ensure efficient operation, a CEP engine must continuously estimate the current statistic values and, when a significant deviation is detected, adapt itself by recalculating the affected evaluation plans. Developing efficient adaptive mechanisms is considered a hard problem and a hot research topic in several fields [BMM+04, DIR07, MM09].

Due to the considerable generality, importance, and complexity of adaptive complex event processing, Chapter 4 of this thesis is entirely devoted to the discussion of this problem. In it, we propose a novel adaptivity mechanism and study it theoretically and empirically in conjunction with a JQPG-based evaluation plan generator.

3.7 Experimental Evaluation

In this section, we present our experimental study on real-world data. Our main goal was to compare some of the well-known JQPG algorithms, adapted for CPG as described above, to the currently used methods developed directly for CPG. The results demonstrate the superiority of the former in terms of quality and scalability of the generated plans.

In the following section we describe the algorithms compared during the study. Then we present the experimental setup, followed by the obtained results.
3.7.1 CPG and JQPG Algorithms

We implemented 5 order-based and 3 tree-based CPG algorithms. Out of those, 3 order-based and 2 tree-based algorithms are JQPG methods adapted to the CEP domain. Our main goal is to evaluate those algorithms against the rest, which are native CPG techniques. The order-based plan generation algorithms included the following:

- Trivial order (TRIVIAL) - the evaluation plan is set to the initial order of the sequence pattern. This strategy is used in various CEP engines based on NFAs, such as SASE [WDR06] and Cayuga [DGP+07].

- Event frequency order (EFREQ) - the events are processed by the ascending order of their arrival frequencies. Described and studied in Chapter 2, this is the algorithm of choice for frameworks such as PB-CED [AcT08] and the lazy NFA.

- Greedy cost-based algorithm (GREEDY) - this greedy heuristic algorithm for JQPG proceeds by selecting at each step the relation which minimizes the value of the cost function [Swa89]. Here and below, unless otherwise stated, we will use cost functions minimizing the intermediate results size (Sections 3.4.1 and 3.4.2).

- Iterative improvement algorithm (II-RANDOM / II-GREEDY) - a local search JQPG algorithm, starting from some initial execution plan and attempting a set of moves to improve the cost function, until a local minimum is reached. In this study, we experimented with two variations of this algorithm, presented in [Swa89]. The first, denoted as II-RANDOM, starts from a random order. The second, denoted as II-GREEDY, first applies a greedy algorithm to create an initial state. In both cases, the functions used to traverse between states are swap (the positions of two event types in a plan are swapped) and cycle (the positions of three event types are shifted).

- Dynamic programming algorithm for left-deep trees (DP-LD) - first presented in [SAC+79], this exponential-time algorithm utilizes dynamic programming to produce a provably optimal execution plan. The result is limited to a left-deep tree topology.

For the tree-based plan generation algorithms, the following were used:

- ZStream plan generation algorithm (ZSTREAM) - creates an evaluation tree by iterating over all possible tree topologies for a given sequence of leaves [MM09].

- ZStream with greedy cost-based ordering (ZSTREAM-ORD) - as was demonstrated in Section 3.2.3, the limitation of the ZStream algorithm is in its inability to modify the order of tree leaves. This algorithm attempts to utilize an order-based JQPG method to overcome this drawback. It operates by first executing GREEDY on the leaves of the tree to produce a ‘good’ ordering, then applying ZSTREAM on the resulting list.

- Dynamic programming algorithm for bushy trees (DP-B) - same as DP-LD [SAC+79], but without the topology restriction.

3.7.2 Experimental Setup

Two real-world datasets were used in the experiments. The first dataset, taken from the NASDAQ stock market historical records [EOD], is the one used in Chapter 2 for
the experimental evaluation of lazy NFA (Section 2.7). In this dataset, each record represents a single update to the price of a stock, spanning a 1-year period and covering over 2100 stock identifiers with prices periodically updated. Our input stream contained 80,509,033 primitive events, each consisting of a stock identifier, a timestamp, and a current price. For each identifier, a separate event type was defined. In addition, we augmented the event format to include the difference between the current and the previous price of each stock. The differences were calculated during the preprocessing stage.

The second dataset contains the vehicle traffic sensor data, provided by City of Aarhus, Denmark [AGM15] and collected over a period of 4 months from 449 observation points, with 13,577,132 primitive events overall. Each event represents an observation of traffic at the given point. The attributes of an event include, among others, the point ID, the average observed speed, and the total number of observed vehicles during the last 5 minutes.

To compare a set of plan generation algorithms, we need to use them to create a set of evaluation plans for the same pattern and apply the resulting plans on the input data stream using a CEP platform of choice. To that end, we implemented two evaluation mechanisms discussed in this chapter, the lazy Chain NFA (Section 2.4) and the instance-based tree model based on ZStream [MM09] as presented in Section 3.2.3. The former was then used to evaluate plans created by each order-based CPG or JQPG algorithm on the patterns generated as described below. The latter was similarly used for comparing tree-based plans.

The majority of the experiments were performed separately on 5 sets of patterns: (1)pure sequences; (2)sequences with a negated event (marked as 'negation' patterns in the graphs below); (3)conjunctions; (4)sequences containing an event under KL operator (marked as 'Kleene closure' patterns); (5)composite patterns, consisting of a disjunction of three sequences (marked as 'disjunction' patterns). Each set contained 500 patterns with the sizes (numbers of the participating events) ranging from 3 to 7, 100 patterns for each value. The pattern time window was set to 20 minutes.

For the stock dataset, the pattern structure was motivated by the problem of monitoring the relative changes in stock prices. Each pattern included a number of predicates, roughly equal to half the size of a pattern, comparing the difference attributes of two of the involved event types. Note that this type of patterns is different from the one utilized for the experiments in Section 2.7. For example, one pattern of size 3 from the set of conjunction patterns was defined as follows:

```
PATTERN AND(MSFT_Stock m, GOOG_Stock g, INTC_Stock i)
WHERE (m.difference < g.difference)
WITHIN 20 minutes.
```

The intention of this particular pattern is to examine the shift in the value of Intel’s stock in situations where Google’s stock price change is higher than Microsoft’s.
The patterns for the traffic dataset were constructed similarly. Their general structure was motivated by normal driving behavior, where the average speed tends to decrease with the increase in the number of vehicles on the road. We requested to detect the violations of this model, i.e., combinations (sequences, conjunctions, etc., depending on the operator involved) of three or more observations with either an increase or a decline in both the number of vehicles and the average speed.

All arrival rates and predicate selectivities were calculated during the preprocessing stage. The measured arrival rates varied between 1 and 47 events per second, and the selectivities ranged from 0.002 to 0.92. As discussed in Section 3.6.3, in most real-life scenarios these statistics are not available in advance and may fluctuate frequently and significantly during runtime. We experimentally study the impact of these issues in Chapter 4.

We selected throughput and memory consumption as our performance metrics for this study. Throughput was defined as the number of primitive events processed per second during pattern detection using the selected plan. To estimate the memory consumption, we measured the peak memory required by the system during evaluation. The metrics were acquired separately for each pattern, and the presented results were then calculated by taking the average.

All models and algorithms under examination were implemented in Java. The experiments were run on a machine with 2.20 Ghz CPU and 16.0 GB RAM and took more than 1.5 months to complete for each dataset.

### 3.7.3 Experimental Results

Figures 3.4 and 3.5 present the comparison of the plan generation algorithms described in Section 3.7.1 in terms of throughput and memory consumption, respectively. Each group represents the results obtained on a particular set of patterns described above, and each bar depicts the average value of a performance metric for a particular algorithm. For clarity, order-based and tree-based methods are shown separately. More detailed results partitioned by the pattern size are presented in Appendix C.

On average, the plans generated using JQPG algorithms achieve a considerably higher throughput than those created using native CPG methods. For order-based plans, the perceived gain of the best-performed DP-LD over EFREQ ranged from a factor of 1.3 for sequence patterns over traffic data to 2.7 for conjunctions over stock data. Similar results were obtained for tree-based plans (ZSTREAM vs. DP-B). JQPG methods also display better overall memory utilization. The order-based JQPG plans consume about 60-85% of the memory required by those produced by EFREQ. An even greater difference was observed for tree-based plans, with DP-B using up to almost 4 times less memory than the CEP-native ZSTREAM.

Unsurprisingly, the best performance was observed for plans created using the exhaustive algorithms based on dynamic programming, namely DP-LD and DP-B.
Figure 3.4: Throughput for different pattern types: (a) stock dataset, order-based methods; (b) stock dataset, tree-based methods; (c) traffic dataset, order-based methods; (d) traffic dataset, tree-based methods.

Figure 3.5: Memory consumption for different pattern types: (a) stock dataset, order-based methods; (b) stock dataset, tree-based methods; (c) traffic dataset, order-based methods; (d) traffic dataset, tree-based methods.
However, due to the exponential complexity of these algorithms, their use in practice may be problematic for large patterns, especially in systems where new evaluation plans are to be generated with high frequency. Thus, one goal of the experimental study was to test the exhaustive JQPG methods against the nonexhaustive ones (such as GREEDY and II algorithms) to see whether the performance gain of the former category is worth the high plan generation cost.

For the order-based case, the answer is indeed negative, as the results for DP-LD and the heuristic JQPG algorithms are comparable and no significant advantage is achieved by the former for most pattern types. Due to the relatively small size of the left-deep tree space, the heuristics usually succeed in locating the globally optimal plan. Moreover, the II-GREEDY algorithm generally produces plans that are slightly more memory-efficient. This can be attributed to our cost model, which only counts the partial matches, but does not capture the other factors such as the size of the buffered events. The picture looks entirely different for the tree-based methods, where DP-B displays a convincing advantage over both the basic ZStream algorithm and its combination with the greedy heuristic method.

Another important conclusion from Figures 3.4 and 3.5 is that methods following the tree-based model greatly outperform the order-based ones, both in throughput and memory consumption. This is not a surprising outcome, as the tree-based algorithms are capable of creating a significantly larger space of plans. However, note that the best order-based JQPG algorithm (DP-LD) is comparable or even superior to the CPG-native ZStream in most settings.

In our next experiment, we evaluated the quality of the cost functions used during plan generation. To that end, we created 60 order-based and 60 tree-based plans for patterns of various types using different algorithms. The plans were then executed on the stock dataset. The throughput and the memory consumption measured during each execution are shown in Figure 3.6 as the function of the cost assigned to each plan by the corresponding function (Costord or Costtree). The obtained throughput seems to be inversely proportional to the cost, behaving roughly as $\frac{1}{c}$; $c \geq 1$. For memory consumption, an approximately linear dependency can be observed. These results match our expectations, as a cheaper plan is supposed to yield better performance and require less memory. We may thus conclude that the costs returned by Costord and Costtree provide a reasonably accurate estimation of the actual performance of a plan.

The above conclusion allowed us to repeat the experiments summarized in Figures 3.4 and 3.5 for larger patterns, using the plan cost as the objective function. We generated 200 stock-based patterns of sizes ranging from 3 to 22. We then created a set of plans for each pattern using different algorithms and recorded the resulting plan costs. Due to the exponential growth of the cost with the pattern size, directly comparing the costs was impractical. Instead, the normalized cost was calculated for every plan. The normalized cost of a plan $P_l$ created by an algorithm $A$ for a pattern $P$ was defined as the cost of a plan generated for $P$ by the empirically worst algorithm (the CEP-native
The results for selected algorithms are depicted in Figure 3.7(a). Each data point represents an average normalized cost for all plans of the same size created by the same algorithm. As we observed previously, the DP-based join algorithms consistently produced significantly cheaper plans (up to a factor of 57) than the heuristic alternatives. Also, the worst JQPG method (GREEDY) and the best CPG method (ZSTREAM) produced plans of similar quality, with the former slightly overperforming the latter for larger pattern sizes. The worst-performing EFREQ algorithm was used for normalized cost calculation and is thus not shown in the figure.

Figure 3.7(b) presents the plan generation times measured during the above experiment. The results are displayed in logarithmic scale. While all algorithms incur only negligible optimization overhead for small patterns, it grows rapidly for methods based on dynamic programming (for a pattern of length 22, it took over 50 hours to create a plan using DP-B). This severely limits the applicability of the DP-based approaches when the number of events in a pattern is high. On the other hand, all non-DP algorithms were able to complete in under a second even for the largest tested patterns. The join-based greedy algorithm (GREEDY) demonstrated the best overall trade-off between optimization time and quality.

Figure 3.6: Performance metrics as a function of the cost for order-based and tree-based patterns: (a) throughput; (b) memory consumption.
Figure 3.7: Generation of large plans (selected algorithms): (a) average normalized plan cost; (b) average plan generation time (logarithmic scale). The results are presented as a function of pattern size.

Next, we studied the performance of the hybrid throughput-latency cost model introduced in Section 3.6.1. Each of the 6 JQPG-based methods discussed in Section 3.7.1 was evaluated on the stock dataset using three different values for the throughput-latency trade-off parameter $\alpha$: 0, 0.5 and 1. Note that for the first case ($\alpha = 0$) the resulting cost model is identical to the one defined in Section 3.4 and used in the experiments above. For each algorithm and for each value of $\alpha$, the throughput and the average latency (in milliseconds) were measured.

Figure 3.8 demonstrates the results, averaged over 500 patterns included in the sequence pattern set. Measurements obtained using the same algorithm are connected by straight lines, and the labels near the highest points (diamonds) indicate the algorithms corresponding to these points. It can be seen that increasing the value of $\alpha$ results in a significantly lower latency. However, this also results in a considerable drop in throughput for most algorithms. By fine-tuning this parameter, the desired latency can be achieved with minimal loss in throughput. It can also be observed that the tree-based algorithms DP-B and ZSTREAM-ORD (and, to some extent, the order-based II-GREEDY) achieve a substantially better throughput-latency trade-off as compared to other methods.
Figure 3.8: Throughput vs. latency using different values for the alpha parameter of the cost model.

Figure 3.9: Throughput for different event selection strategies (logarithmic scale): (a) order-based methods; (b) tree-based methods.
Finally, we performed a comparative throughput evaluation of the sequence pattern set under three different event selection strategies: *skip-till-any-match*, *skip-till-next-match* and *contiguity* (Section 3.6.2). The results are depicted in Figure 3.9 for all algorithms under examination. This experiment was only conducted on the stock data. Due to large performance gaps between the examined methods, the results are displayed in logarithmic scale.

For *skip-till-next-match*, JQPG methods hold a clear advantage, albeit less significant than the one demonstrated above for *skip-till-any-match*. This outcome supports our expectations, as the two selection strategies are very similar, with the only difference being the treatment of appearance of a primitive event in multiple full matches (allowed by *skip-till-any-match* and forbidden by *skip-till-next-match*).

The opposite observation can be made about the *contiguity* strategy, where the trivial algorithm following a static plan outperforms other, more complicated methods. Due to the simplicity of the event detection process and the lack of nondeterminism in this case, the plan set by an input specification always performs best, while the alternatives introduce a slight additional overhead of reordering and event buffering.
Chapter 4

Efficient Adaptive Detection of Complex Event Patterns

4.1 Introduction

In Chapter 3, we established the notion of a plan-based evaluation mechanism, used by a CEP engine to create an internal representation for each pattern \( P \) to be monitored. This representation is constructed according to the evaluation plan, which reflects the structure of \( P \). The evaluation plan, created by a plan generation algorithm based on a predefined cost model, defines how primitive events are combined into partial matches. Typically, pattern detection performance can be dramatically improved if the statistical characteristics of the monitored data are taken into account during the construction of the evaluation plan. As we discussed above, systems tuned according to the a priori knowledge of these statistics can boost performance by up to several orders of magnitude, especially for highly skewed data.

As an example, consider the following scenario.

Example 4.1.1. A system for managing an array of smart security cameras is installed in a building. All cameras are equipped with face recognition software, and periodical readings from each camera are sent in real time to the main server. We are interested in identifying a scenario in which an intruder accesses the restricted area via the main gate of the building rather than from the dedicated entrance. This pattern can be represented as a sequence of three primitive events: 1) camera A (installed near the main gate) detects a person; 2) later, camera B (located inside the building’s lobby) detects the same person; 3) finally, camera C detects the same person in the restricted area. In addition, we assume that fewer people access the restricted area than pass through the main building entrance. Consequently, the expected number of face recognition notifications arriving from camera C is significantly smaller than the expected number of similar events from cameras A and B.

Figure 4.1(a) demonstrates an example of an eager non-deterministic finite automaton
for detecting this simple pattern by a CEP engine. This NFA is created according to the simple evaluation plan reflecting the structure of the pattern specification and ignoring the statistical properties of the event occurrence. First, a stream of events arriving from camera A is inspected. For each accepted event, the stream of B is probed for subsequently received events specifying the same person. If found, we wait for a corresponding event to arrive from camera C.

Figure 4.1(b) depicts a NFA implementing the lazy evaluation principle (Section 2.3) to employ a more beneficial evaluation plan. First, the stream of events from the least active camera C is monitored. When an event is accepted, the local history is examined for previous readings of B and A.

Unfortunately, in real-life scenarios the statistical properties of the data in the input streams is rarely obtained in advance. Moreover, the data characteristics can change rapidly over time, which may render an initial evaluation plan extremely inefficient. In Example 4.1.1, the number of people near the main entrance might drop dramatically in late evening hours, making the event stream from camera A the first in the plan, as opposed to the event stream from C.

To overcome this problem, a CEP engine must continuously estimate the current values of the target parameters and, if and whenever necessary, adapt itself to the changed data characteristics. We will denote systems possessing such capabilities as *Adaptive CEP (ACEP)* systems.

A common structure of an ACEP system is depicted in Figure 4.2. The evaluation mechanism starts processing incoming events using some initial plan. A dedicated component calculates up-to-date estimates of the statistics (e.g., event arrival rates in Example 4.1.1) and transfers them to the optimizer. The optimizer then uses these values to decide whether the evaluation plan should be updated. If the answer is positive, a plan generation algorithm is invoked to produce a new plan (e.g., a new NFA), which is
then delivered to the evaluation mechanism to replace the previously employed structure. In Example 4.1.1, this algorithm simply sorts the event types in the ascending order of their arrival rates and returns a chain-structured NFA conforming to that order.

Correct decisions by the optimizer are crucial for the successful operation of an adaptation mechanism. As the process of creating and deploying a new evaluation plan is very expensive, we would like to avoid “false positives,” that is, launching reoptimizations that do not improve the currently employed plan. “False negatives,” occurring when an important shift in estimated data properties is missed, are equally undesirable. A flawed decision policy may severely diminish or even completely eliminate the gain achieved by an adaptation mechanism.

The problem of designing efficient and reliable algorithms for reoptimization decision making has been well studied in areas such as traditional query optimization [DIR07]. However, it has received only limited attention in the CEP domain [MM09]. Tree-based NFA that we described in Section 2.6 provide limited support for such functionality. A tree-based automaton continuously reorganizes itself according to the currently observed arrival rates of the primitive events. The main strength of this method is that it is guaranteed to produce the optimal evaluation plan for any given set of events. However, it introduces a substantial computational overhead due to the unconditional modification of the NFA structure. This is especially evident for stable event streams with little to no data variance, for which this technique would be outperformed by a non-adaptive solution using a static plan.
Another approach, introduced in [MM09], defines a constant threshold $t$ for all monitored statistics. When any statistic deviates from its initially observed value by more than $t$, plan reconstruction is activated. This solution is much cheaper computationally than the previous one. However, some reoptimization opportunities may be missed.

Consider Example 4.1.1 again. Recall that we are interested in detecting the events by the ascending order of their arrival rates, and let the rates for events generated by cameras A, B and C be $rate_A = 100$, $rate_B = 15$, $rate_C = 10$ respectively. Obviously, events originating at A are significantly less sensitive to changes than those originating at B and C. Thus, if we monitor the statistics with a threshold $t > 6$, a growth in C to the point where it exceeds B will not be discovered, even though the reoptimization is vital in this case. Alternatively, setting a value $t < 6$ will result in detection of the above change, but will also cause the system to react to fluctuations in the arrival rate of A, leading to redundant plan recomputations.

No single threshold in the presented scenario can ensure optimal operation. However, by removing the conditions involving $t$ and monitoring instead a pair of constraints \{rate$_A >$rate$_B$, rate$_B >$rate$_C$\}, plan recomputation would be guaranteed if and only if a better plan becomes available.

This chapter presents a novel method for making efficient and precise on-the-fly adaptation decisions. Our method is based on defining a tightly bounded set of conditions on the monitored statistics to be periodically verified at runtime. These conditions, which we call \textit{invariants}, are generated during the initial plan creation, and are constantly recomputed as the system adapts to changes in the input. The invariants are constructed to ensure that a violation of at least one of them guarantees that a better evaluation plan is available.

To the best of our knowledge, our proposed mechanism is the first to provably avoid false positives on reoptimization decisions. It also achieves notably low numbers of false negatives as compared to existing alternatives, as shown by our empirical study. This method can be applied to any deterministic algorithm for evaluation plan generation and used in any stream processing scenario.

The contributions and the structure of this chapter can thus be summarized as follows:

- We formally define the reoptimizing decision problem for the complex event processing domain (Section 4.2).
- We present a novel method for detecting reoptimization opportunities in ACEP systems by verifying a set of invariants on the monitored data characteristics and formally prove that no false positives are possible when this method is used. We also extend the basic method to achieve a balance between computational efficiency and precision (Section 4.3).
- We demonstrate how to apply the invariant-based method on two known algorithms for evaluation structure creation, the greedy order-based algorithm (described in Chapter
3) and ZStream algorithm [MM09], and discuss the generalization of these approaches to broader categories of algorithms (Section 4.4).

- We conduct an extensive experimental evaluation, comparing the invariant-based method to existing state-of-the-art solutions. The results of the experiments, performed on two real-world datasets, show that our proposed method achieves the highest accuracy and the lowest computational overhead (Section 4.5).

4.2 Preliminaries

In this section, we briefly repeat the basic notations introduced in Chapters 2 and 3 and used throughout this chapter, outline the event detection process in an ACEP system, and provide a formal definition of the reoptimizing decision problem, which will be further discussed in the subsequent sections.

4.2.1 Notations and Terminology

A pattern recognized by a CEP system is defined by a combination of primitive events, operators, a set of predicates, and a time window. The patterns are formed using declarative specification languages ([CM10, DGH+06, WDR06]).

Each event is represented by a type and a set of attributes, including the occurrence timestamp. Throughout this chapter we assume that each primitive event has a well-defined type, i.e., the event either contains the type as an attribute or it can be easily inferred from the event attributes using negligible system resources. We will denote the pattern size (i.e., the number of distinct primitive events in a pattern) by $n$.

The predicates to be satisfied by the participating events are usually organized in a Boolean formula. Any condition can be specified on any attribute of an event, including the timestamp (e.g., for supporting multiple time windows).

The operators describe the relations between the events comprising a pattern match. Among the most commonly used operators are sequence (SEQ), conjunction (AND), disjunction (OR), negation (typically marked by ‘~’, requires the absence of an event from the stream) and Kleene closure (marked by ‘*’, accepts multiple appearances of an event in a specified position). A pattern may include an arbitrary number of operators.

To illustrate the above, consider Example 4.1.1 again. We will define three event types according to the identifiers of the cameras generating them: A, B and C. For each primitive event, we will set the attribute person_id to contain a unique number identifying a recognized face. Then, to detect a sequence of occurrences of the same person in three areas in a 10-minute time period, we will use the following pattern specification syntax, taken from SASE [WDR06]:
On system initialization, the pattern declaration is passed to the plan generation algorithm \( A \) to create the evaluation plan. The evaluation plan provides a scheme for the CEP engine, according to which its internal pattern representation is created. The plan generation algorithm accepts a pattern specification \( P \) and a set of statistical data characteristic values \( Stat \). It then returns the evaluation plan to be used for detection. If these values are not known in advance, a default, empty \( Stat \), is passed. All native CPG algorithms and adapted JQPG algorithms evaluated in Section 3.7 are examples of plan generation algorithms supporting patterns with arbitrarily complex combinations of the aforementioned operators.

In Example 4.1.1, \( Stat \) contains the arrival rates of event types A, B and C, the evaluation plan is an ordering on the above types, and \( A \) is a simple sorting algorithm, returning a plan following the ascending order of the arrival rates. The CEP engine then adheres to this order during pattern detection. Another popular choice for a statistic to be monitored is the set of selectivities (i.e., the probabilities of success) of the inter-event conditions defined by the pattern.

The plan generation algorithm attempts to utilize the information in \( Stat \) to find the best possible evaluation plan subject to some predefined set of performance metrics, which we denote as \( Perf \). These metrics may include throughput, detection latency, network communication cost, power consumption, and more. For instance, one possible value for \( Perf \) in Example 4.1.1 is \( \{ \text{throughput}, \text{memory} \} \), as processing the events according to the ascending order of their arrival rates was shown in previous chapters to vastly improve memory consumption and throughput of a CEP system.

In the general case, we consider \( A \) to be a computationally expensive operation. We also assume that this algorithm is optimal; that is, it always produces the best possible solution for the given parameters. While this assumption rarely holds in practice, the employed techniques usually tend to produce empirically good solutions (Chapter 3).

An evaluation plan is not constrained to be merely an order. Figure 4.3 demonstrates two possible tree-structured plans as defined by ZStream [MM09]. An evaluation structure following such a plan accumulates the arriving events at their corresponding leaves, and the topology of the internal nodes defines the order in which they are matched and their mutual predicates are evaluated. Matches reaching the tree root are reported to the end users. As we defined in Section 3.3.1, such plans are denoted as tree-based plans, whereas plans similar to the one used for Example 4.1.1 are called order-based plans. While the methods discussed in this chapter are independent of the specific plan structure, we will use order-based and tree-based plans in our examples.
4.2.2 Detection-Adaptation Loop

During evaluation, an ACEP system constantly attempts to spot a change in the statistical properties of the data and to react accordingly. This process, referred to as the detection-adaptation loop, is depicted in Algorithm 4.1.

The system accepts events from the input stream and processes them using the current evaluation plan. At the same time, the values of the data statistics in \( Stat \) are constantly reestimated by the dedicated component (Figure 4.2), often as a background task. While monitoring simple values such as the event arrival rates is trivial, more complex expressions (e.g., predicate selectivities) require advanced solutions. In this work, we utilize existing state-of-the-art techniques from the field of data stream processing [BDMO03, DGIM02]. These histogram-based methods allow to efficiently maintain a variety of stream statistics over sliding windows with high precision and require negligible system resources.

Opportunities for adaptation are recognized by the reoptimizing decision function \( D \), defined as follows:

\[
D : STAT \rightarrow \{true, false\},
\]

where \( STAT \) is a set of all possible collections of the measured statistic values. \( D \) accepts the current estimates for the monitored statistic values (or the initially available values if invoked on startup) and decides whether reoptimization is to be attempted\(^1\). Whenever \( D \) returns \( true \), the detection-adaptation loop invokes \( A \). The output of \( A \) is a new evaluation plan, which, if found more efficient than the current plan subject to the metrics in \( Perf \), is subsequently deployed.

Methods for replacing an evaluation plan on-the-fly without significantly affecting system performance or losing intermediate results are a major focus of current research [DIR07]. Numerous advanced techniques were proposed in the field of continuous query

---

\(^1\)In theory, nothing prevents \( D \) from using additional information, such as the past and the current system performance. We will focus on the restricted definition where only data-related statistics are considered.
processing in data streams [AAOM14, KYC+06, ZRH04]. In our work, we use the CEP-based strategy introduced in Section 2.6. Let $t_0$ be the time of creation of the new plan. Then, partial matches containing at least a single event accepted before $t_0$ are processed according to the old plan $p_{old}$, whereas the newly created partial matches consisting entirely of “new” events are treated according to the new plan $p_{new}$. Note that since $p_{old}$ and $p_{new}$ operate on disjoint sets of matches, there is no duplicate processing during execution. At time $t_0 + W$ (where $W$ is the time window of the monitored pattern), the last “old” event expires and the system switches fully to $p_{new}$.

In general, we consider the deployment procedure to be a costly operation and will attempt to minimize the number of unnecessary plan replacements.

**Algorithm 4.1** Detection-adaptation loop in an ACEP system

Input: pattern specification $P$, plan generation algorithm $A$, reoptimizing decision function $D$, initial statistic values $\text{in\_stat} \in \text{STAT}$

```plaintext
curr_plan \leftarrow A(P, \text{in\_stat})
```

while more events are available:

- process incoming events using $curr_plan$
- $curr_stat \leftarrow$ estimate current statistic values
- if $D(curr_stat)$:
  - $new_plan \leftarrow A(P, curr_stat)$
  - if $new_plan$ is better than $curr_plan$:
    - $curr_plan \leftarrow new_plan$
    - apply $curr_plan$

### 4.2.3 Reoptimizing Decision Problem

The reoptimizing decision problem is the problem of finding a function $D$ that maximizes the performance of a CEP system subject to $\text{Perf}$. It can be formally defined as follows: given the pattern specification $P$, the plan generation algorithm $A$, the set of monitored statistics $\text{Stat}$, and the set of performance metrics $\text{Perf}$, find a reoptimizing decision function $D$ that achieves the best performance of the ACEP detection-adaptation loop (Algorithm 4.1) subject to $\text{Perf}$.

In practice, the quality of $D$ is determined by two factors. The first factor is the correctness of the answers returned by $D$. Wrong decisions can either fall into the category of false positives (returning true when the currently used plan is still the best possible) or false negatives (returning false when a more efficient plan is available). Both cases cause the system to use a sub-optimal evaluation plan. The second factor is the time and space complexity of $D$. As we will see in Section 4.5, an accurate yet resource-consuming implementation of $D$ may severely degrade system performance regardless of its output.

We can now analyze the solutions to the reoptimizing decision problem implemented by the adaptive frameworks which we discussed in Section 4.1. The tree-based NFA
(Section 2.6) defines a trivial decision function $D$, unconditionally returning true. In ZStream [MM09] this function loops over all values in the input parameter $curr_stat$ and returns true if and only if a deviation of at least $t$ is detected.

### 4.3 Invariant-Based Method for the Reoptimizing Decision Problem

As illustrated above, the main drawback of the previously proposed decision functions is their coarse granularity, as the same condition is verified for every monitored data property. We propose a different approach, based on constructing a set of fine-grained invariants that reflect the existing connections between individual data characteristics. The reoptimizing decision function $D$ will then be defined as a conjunction of these invariants.

In this section, we present the invariant-based decision method and discuss its correctness guarantees, time and space complexity, and possible optimizations.

#### 4.3.1 Invariant Creation

A decision invariant (or simply invariant) will be defined as an inequality of the following form:

$$f_1(stat_1) < f_2(stat_2),$$

where $stat_1, stat_2 \in STAT$ are sets of the monitored statistic values and $f_1, f_2 : STAT \rightarrow \mathbb{R}$ are arbitrary functions.

We are interested in finding a single invariant for each building block of the evaluation plan in current use. A building block is formally defined as the most primitive, indivisible element of an evaluation plan. A plan can then be seen as a collection of building blocks. For instance, the plan for detecting a sequence of three events of types A, B and C, which we discussed in Example 4.1.1, is formed by the following three blocks:

1. “Accept an event of type C”;
2. “Scan the history for events of type B matching the accepted C”;
3. “Scan the history for events of type A matching the accepted C and B”.

In general, in an order-based plan each step in the selected order will be considered a block, whereas for tree-based plans a block is equivalent to an internal node.

We know that the specific plan from the above example was chosen because the plan generation algorithm $\mathcal{A}$ sorts the event types according to their arrival rates. If, for instance, the rate of B exceeded that of A, the second block would have been “Scan the history for events of type A matching the accepted C” and the third would also have changed accordingly. In other words, the second block of the plan is so defined because, during the run of $\mathcal{A}$, the condition $rate_B < rate_A$ was at some point checked, and
the result of this check was positive. Following the terminology defined above, in this example \( STAT \) consists of all valid arrival rate values and \( f_1, f_2 \) are trivial functions, i.e., \( f_1(x) = f_2(x) = x \).

We will denote any condition (over the measured statistic values) whose verification has led the algorithm to include some building block in the final plan as a \textit{deciding condition}. Obviously, no generic method exists to distinguish between a deciding condition and a regular one. This process is to be applied separately on any particular algorithm \( A \) based on its semantics. In our example, assume that the arrival rates are sorted using a simple min-sort algorithm, selecting the smallest remaining one at each iteration. Then, any direct comparison between two arrival rates will be considered a deciding condition, as opposed to any other condition which may or may not be a part of the implementation of this particular algorithm’s implementation.

When \( A \) is invoked on a given input, locations can be marked in the algorithm’s execution flow where the deciding conditions are verified. We will call any actual verification of a deciding condition a \textit{block-building comparison} (BBC). For instance, assume that we start executing our min-sort algorithm and a deciding condition \( \text{rate}_C < \text{rate}_A \) is verified. Further assume that \( \text{rate}_C \) is smaller than \( \text{rate}_A \). Then, this verification is a BBC associated with the building block “Accept an event of type C first”, because, unless this deciding condition holds, the block will not be included in the final plan. This will also be the case if \( \text{rate}_C < \text{rate}_B \) is subsequently verified and \( \text{rate}_C \) is smaller. If \( \text{rate}_B \) is smaller, the opposite condition, \( \text{rate}_B < \text{rate}_C \), becomes a BBC associated with a block “Accept an event of type B first”. Overall, \((n - 1)\) BBCs take place during the first min-sort iteration, \((n - 2)\) during the second iteration, and so forth.

In general, for each building block \( b \) of any evaluation plan, we can determine a \textit{deciding condition set (DCS)}. A DCS of \( b \) consists of all deciding conditions that were actually checked and satisfied by BBCs belonging to \( b \) as explained above. Note that, by definition, the intersection of two DCSs is always empty. In our example, assuming that the blocks listed above are denoted as \( b_1, b_2, b_3 \), the deciding condition sets are as follows:

\[
\begin{align*}
DCS_1 &= \{\text{rate}_C < \text{rate}_B, \text{rate}_C < \text{rate}_A\}, \\
DCS_2 &= \{\text{rate}_B < \text{rate}_A\}, \\
DCS_3 &= \emptyset.
\end{align*}
\]

As long as the above conditions hold, no other evaluation plan can be returned by \( A \). On the other hand, if any of the conditions is violated, the outcome of \( A \) will result in generating a different plan. If we define the decision function \( D \) as a conjunctive Boolean formula containing the conditions from the deciding condition sets, we will recognize situations in which the current plan becomes sub-optimal with high precision and confidence.

However, verifying all deciding conditions for all building blocks is very inefficient.
In our simple example, the total number of such conditions is quadratic in the number of event types participating in the pattern. For more complicated plan categories and generation algorithms, this dependency may grow to a high-degree polynomial or even become exponential. Since the adaptation decision is made during every iteration of Algorithm 4.1, the overhead may not only decrease the system throughput, but also negatively affect the response time.

To overcome this problem, we will constrain the number of conditions to be verified by $D$ to one per building block. For each deciding condition set $DCS_i$, we will determine the tightest condition, that is, the one that was closest to being violated during plan generation. This tightest condition will be selected as an invariant of the building block $b_i$. In other words, we may alternatively define an invariant as a deciding condition selected for actual verification by $D$ out of a DCS. More formally, given a set

$$DCS_i = \{c_1, \ldots, c_m\}, c_k = (f_{k,1}(stat_{k,1}) < f_{k,2}(stat_{k,2}))$$

we will select a condition that minimizes the expression

$$(f_{k,2}(stat_{k,2}) - f_{k,1}(stat_{k,1}))$$

as an invariant of the building block $b_i$.

In the example above, the invariant for $DCS_i$ is $rate_C < rate_B$, since we know that $rate_B < rate_A$, and therefore $rate_B - rate_C < rate_A - rate_C$. It is clear that $rate_B$ is a tighter bound for the value of $rate_C$ than $rate_A$.

To summarize, the process of invariant creation proceeds as follows. During the run of $A$ on the current set of statistics $Stat$, we closely monitor its execution. Whenever a block-building comparison is detected for some block $b$, we add the corresponding deciding condition to the DCS of $b$. After the completion of $A$, the tightest condition of each DCS is extracted and added to the invariant list.

Figure 4.4 demonstrates the invariant creation process applied on the pattern from Example 4.1.1 and the rate-sorting algorithm $A$ discussed above. Each subfigure depicts a different stage in the plan generation and presents the DCSs and the BBCs involved at this stage.

As discussed above, this generic method has to be adapted to any specific implementation of $A$. This is trivially done for any $A$ which constructs the solution plan in a step-by-step manner, selecting and appending one building block at a time. However, for algorithms incorporating other approaches, such as dynamic programming, it is more challenging to attribute a block-building comparison to a single block of the plan. In Section 4.4, we will exemplify this process on two algorithms taken from the previous work in the field and discuss its applicability on broader algorithm categories.
4.3.2 Invariant Verification and Adaptation

During the execution of the detection-adaptation loop (Algorithm 4.1), $D$ traverses the list of invariants built as described above. It returns true if a violated invariant was found (according to the current statistic estimates) and false otherwise. This list is sorted according to the order of the respective building blocks in the evaluation plan. In Example 4.1.1, first the invariant $rate_C < rate_B$ will be verified, followed by $rate_B < rate_A$. The reason is that an invariant implicitly assumes the correctness of the preceding invariants (e.g., $rate_B < rate_A$ assumes that $rate_C < rate_B$ holds; otherwise, it should have been changed to $rate_C < rate_A$). For tree-based plans, the verification proceeds in a bottom-up order. For example, for the tree plan displayed in Figure 4.3(a), the order is $(A, B) \rightarrow (A, B, C)$.

If a violation of an invariant is detected, $A$ is invoked to create a new evaluation plan. In this case, the currently used invariants are invalidated and a new list is created following the process described above. Subsequent verifications performed by $D$ are then based on the new invariants.

Assuming that any invariant can be verified in constant time and memory, the complexity of $D$ using the invariant-based method is $O(B)$, where $B$ is the number of the building blocks in an evaluation plan. This number is bounded by the pattern size (the number of event types participating in a pattern) for both order-based and tree-based plans. To guarantee this result, an application of the invariant-based method
on a specific implementation of $A$ has to ensure that the verification of a single invariant is a constant-time operation, as we exemplify in Section 4.4.

### 4.3.3 Correctness Guarantees and the K-invariant Method

We will now formally prove that the invariant-based method presented above guarantees that no false positive detections will occur during the detection-adaptation loop.

**Theorem 4.1.** Let $D$ be a reoptimizing decision function implemented according to the invariant-based method. Let $A$ be a deterministic plan generation algorithm in use and let $p$ be the currently employed plan. Then, if at some point during execution $D$ returns true, the subsequent invocation of $A$ will return a plan $p'$, such that $p' \neq p$.

By definition, if $D$ returns true, then there is at least one invariant whose verification failed, i.e., its deciding condition does not hold anymore. Let $c$ be the first such condition, and let $b_i$ be the building block such that $c \in DCS_i$ (recall that there is only one such $b_i$). Then, by determinism of $A$ and by the ordering defined on the invariants, the new run of $A$ will be identical to the one that produced $p$ until the block-building comparison that checks $c$. At that point, by definition of the block-building comparison, the negative result of validating $c$ will cause $A$ to reject $b_i$ as the current building block and select a different one, thus producing a plan $p'$, which is different from $p$. ■

Since we assume $A$ to always produce the optimal solution, the above result can be extended.

**Corollary 4.2.** Let $D$ be an invariant-based reoptimizing decision function and let $A$ be a deterministic plan generation algorithm in use. Then, if at some point during execution $D$ returns true, the subsequent invocation of $A$ will return a plan that is more efficient than the currently employed one.

Note that the opposite direction of Theorem 4.1 does not hold. It is still possible that a more efficient evaluation plan can be deployed, yet this opportunity will not be detected by $D$ because we only pick a single condition from each deciding condition set (see Section 4.4.2 for an example). If we were to include the whole union of the above sets in the invariant set, even stronger guarantees could be achieved, as stated in the following theorem.

**Theorem 4.3.** Let $D$ be a reoptimizing decision function implemented according to the invariant-based method, with all conditions from all $DCS$s included in the invariant set. Let $A$ be a deterministic plan generation algorithm in use and let $p$ be the currently employed plan. Then, if and only if at some point during the execution $D$ returns true, the subsequent invocation of $A$ will return a plan $p'$, such that $p' \neq p$.

The first direction follows immediately from Theorem 4.1. For the second direction, let $p' \neq p$ and let $b_i \in p, b'_i \in p'$ be the first building blocks that differ in $p$ and $p'$. By
A's determinism, there exist \( f_1, f_2, \text{stat}_1, \text{stat}_2 \) s. t.

\[
(f_1 (\text{stat}_1) < f_2 (\text{stat}_2)) \in DCS_i,
\]

\[
(f_2 (\text{stat}_2) < f_1 (\text{stat}_1)) \in DCS_i',
\]

as otherwise there would be no way for \( A \) to deterministically choose between \( b_i \) and \( b_i' \). Since \( p' \) was created by \( A \) using the currently estimated statistic values, we can deduce that \( f_2 (\text{stat}_2) < f_1 (\text{stat}_1) \) holds. Consequently, \( f_1 (\text{stat}_1) < f_2 (\text{stat}_2) \) does not hold. By the assumption that all deciding conditions are included in the invariant set, \( D \) will necessarily detect this violation, which completes the proof. ■

The above result shows that greater precision can be gained if we do not limit the number of monitored invariants per building block. However, as discussed above, validating all deciding conditions may drastically increase the adaptation overhead.

The tradeoff between performance and precision can be controlled by introducing a new parameter \( K \), defined as the maximal number of conditions from a deciding set to select as invariants. We will refer to the method using a specific value of \( K \) as the \textit{K-invariant method}, as opposed to the \textit{basic invariant method} discussed above. Note that the 1-invariant method is equivalent to the basic one. The K-invariant method becomes more accurate and more time-consuming for higher values of \( K \). The total number of the invariants in this case is at most \( K \cdot (B - 1) \).

### 4.3.4 Distance-Based Invariants

By Corollary 4.2, it is guaranteed that a new, better evaluation plan will be produced following an invariant violation. However, the magnitude of its improvement over the old plan is not known. Consider a scenario in which two event types in a pattern have very close arrival rates. Further assume that there are slight oscillations in the rates, causing the event types to swap positions periodically when ordered according to this statistic. If an invariant is defined comparing the arrival rates of these two types, then \( D \) will discover these minor changes and two evaluation plans with little to no difference in performance will be repeatedly produced and deployed. Although not a “false positive” by definition, the overhead implied by this situation may exceed any benefit of using an adaptive platform.

To overcome this problem, we will introduce the notion of the minimal distance \( d \), defined as the smallest relative difference between the two sides of the inequality required for an invariant to be considered as violated. That is, given a deciding condition

\[
f_{k,1} (\text{stat}_{k,1}) < f_{k,2} (\text{stat}_{k,2}),
\]

we will construct the invariant to be verified by \( D \) as follows:

\[
(1 + d) \cdot f_{k,1} (\text{stat}_{k,1}) < f_{k,2} (\text{stat}_{k,2}).
\]
The experimental study in Section 4.5 demonstrates that a correctly chosen $d$ leads to a significant performance improvement over the basic technique. However, finding a sufficiently good $d$ is a difficult task, as it depends on the data, the type of statistics, the invariant expression, and the frequency and magnitude of the runtime changes. We identify the following directions for solving this problem:

1. Parameter scanning: empirically checking a range of candidate values to find the one resulting in the best performance. This method is the simplest, but often infeasible in real-life scenarios.

2. Data analysis methods: calculating the distance by applying a heuristic rule on the currently available statistics can provide a good estimate in some cases. For instance, we can monitor the initial execution of the plan generation algorithm and set $d$ as the average obtained relative difference between the sides of a deciding condition or, more formally:

$$d = \frac{\text{AVG} \left( \left| (f_{k,2}(\text{stat}_{k,2}) - f_{k,1}(\text{stat}_{k,1})) \right| \right)}{\min(f_{k,1}(\text{stat}_{k,1}), f_{k,2}(\text{stat}_{k,2}))}.$$ 

The effectiveness of this approach depends on the distribution and the runtime behavior of the statistical values. Specifically, false positives may be produced when the values are very close and the changes are frequent. Still, we expect it to perform reasonably well in the common case. This technique can also be utilized to produce a starting point for parameter scanning.

3. Meta-adaptive methods: dynamically tuning $d$ on-the-fly to adapt it to the current stream statistics. This might be the most accurate and reliable solution. We start with some initial value, possibly obtained using the above techniques. Then, as invariants are violated and new plans are computed, we modify $d$ to prevent repeated reoptimization attempts when the observed gain in plan quality is low. An even higher precision can be achieved by additionally utilizing fine-grained per-invariant distances. This advanced research direction is a subject for our future work.

We implement and experimentally evaluate the first two approaches in Section 4.5.

4.3.5 Tightest Conditions Selection Strategy

In Section 4.3.1 we explained that, given a DCS for a block $b$, the condition to be included in the invariant set is the one with the smallest difference between the sides of the inequality (according to the currently observed values of the statistics). The intention of this approach is to pick a condition most likely to be violated later. This, however, is merely a heuristic. In many cases, there may be no correlation at all between the difference of the currently observed values and the probability of the new values to
violate the inequality. Hence, this selection strategy may result in suboptimal invariant selection.

However, sometimes the information regarding the expected variance of a data property is either given in advance or can be calculated to some degree of precision and even approximated on-the-fly [BDMO03]. In these cases, a possible optimization would be to explicitly calculate the violation probability of every deciding condition and use it as a metric for selecting an invariant from a deciding condition set.

### 4.4 Applications of the Invariant-Based Method

In Section 4.3, we presented a generic method for defining a reoptimizing decision function $D$ as a list of invariants. As we have seen, additional steps are required in order to apply this method to a specific choice of the evaluation plan structure and the plan generation algorithm. Namely, the following should be strictly defined: 1) what is considered a building block in a plan; 2) what is considered a block-building comparison in $\mathcal{A}$; 3) how we associate a BBC with a building block. Additionally, efficient verification of the invariants must be ensured. In this section, we will exemplify this process on two plan-algorithm combinations taken from previous works in the field. The experimental study in Section 4.5 will also be conducted on these adapted algorithms. We also discuss how the presented techniques can be generalized to several classes of algorithms.

#### 4.4.1 Greedy Algorithm for Order-Based Plans

The greedy heuristic algorithm based on cardinalities and predicate selectivities was first described in [Swa89] for creating left-deep tree plans for join queries. We adapted it to the CEP domain in Chapter 3. The algorithm supports all operators described in Section 4.2.1 and their arbitrary composition. Its basic form, which we describe shortly, only targets conjunction and sequence patterns of arbitrary complexity. Support for other operators and their composition is implemented by either activating transformation rules on the input pattern or applying post-processing steps on the generated plan (e.g., to augment it with negated events). As these additional operations do not affect the application of the invariant-based method, we do not describe them here. The reader is referred to Section 3.5 for more details.

The algorithm proceeds iteratively, selecting at each step the event type which is expected to minimize the overall number of partial matches (subsets of valid pattern matches) to be kept in memory. At the beginning, the event type with the lowest arrival rate (multiplied by the selectivities of any predicates possibly defined solely on this event type) is chosen. At each subsequent step $i; i > 1$, the event type to be selected is
the one that minimizes the expression
\[
\left( \prod_{j=1}^{i} r_{p_j} \cdot \prod_{j, k \leq i} sel_{p_j, p_k} \right),
\]
where \( r_x \) stands for the arrival rate of the \( x^{th} \) event type in a pattern, \( sel_{x,y} \) is the selectivity of the predicate defined between the \( x^{th} \) and the \( y^{th} \) event types (equals to 1 if no predicate is defined), \( p_1, \cdots, p_{i-1} \) are the event types selected during previous steps, and \( p_i \) is the candidate event type for the current step. Since a large part of this expression is constant when selecting \( p_i \), it is sufficient to find an event type, out of those still not included in the plan, minimizing the expression
\[
\left( r_{p_i} \cdot sel_{p_i, p_i} \cdot \prod_{k < i} sel_{p_k, p_i} \right).
\]

Algorithm 4.2 Greedy algorithm for order-based plan generation (basic form)

Input: event types \( e_1, \cdots, e_n \), arrival rates \( r_1, \cdots, r_n \), inter-event predicate selectivities \( sel_{1,1}, \cdots, sel_{n,n} \)

Output: an order-based evaluation plan \( E = e_{p_1}, e_{p_2}, \cdots, e_{p_n} \)

\[
E \leftarrow \emptyset
\]
\[
p_1 = \text{argmin}_j \{ r_j \cdot sel_{j,j} \}
\]
add \( e_{p_1} \) to \( E \)

for \( i \) from 2 to \( n \):
\[
p_i = \text{argmin}_{j \notin E} \{ r_j \cdot sel_{j,j} \cdot \prod_{k < i} sel_{p_k,j} \}
\]
add \( e_{p_i} \) to \( E \)

return \( E \)

Algorithm 4.2 depicts the plan generation process. When all selectivities satisfy \( sel_{x,y} = 1 \), i.e., no predicates are defined for the pattern, this algorithm simply sorts the events in an ascending order of their arrival rates.

We will define a building block for order-based evaluation plans produced by Algorithm 4.2 as a single directive of processing an event type in a specific position of a plan. That is, a building block is an expression of the form “Process the event type \( e_j \) at \( i^{th} \) position in a plan”. Obviously, a full plan output by the algorithm contains exactly \( n \) blocks, and a total of \( O(n^2) \) blocks is considered during the run. Deciding conditions created for such a block are defined as:
\[
r_j \cdot sel_{j,j} \cdot \prod_{k < i} sel_{p_k,j} < r_{j'} \cdot sel_{j',j'} \cdot \prod_{k < i} sel_{p_k,j'}.
\]
Here, \( e_{j'}, j' \neq j \) is an event type which was considered to occupy \( i^{th} \) position at some point but eventually \( e_j \) was selected. Note that, while in the worst case the products may contain up to \( n - 1 \) multiplicands, in most cases the number of the predicates
defined over the events in a pattern is significantly lower than $n^2$. Therefore, invariant verification will be executed in near-constant time.

### 4.4.2 Dynamic Programming Algorithm for Tree-Based Plans

**Algorithm 4.3** ZStream algorithm for tree-based plan generation

Input: event types $e_1, \cdots, e_n$, arrival rates $r_1, \cdots, r_n$, inter-event predicate selectivities $sel_{1,1}, \cdots, sel_{n,n}$

Output: a tree-based evaluation plan $T$

$$subtrees \leftarrow \text{new two-dimensional matrix of size } n \times n$$

for $i$ from 1 to $n$:

$$subtrees[i][1].\text{cardinality} = subtrees[i][1].\text{cost} = r_i$$

for $i$ from 2 to $n$:

for $j$ from 1 to $n - i + 1$:

for $k$ from $j + 1$ to $j + i$:

$$\text{new}\_\text{cardinality} = \text{Card}(subtrees[k - j][j].\text{cardinality},$$

$$subtrees[i - (k - j)][k].\text{cardinality})$$

$$\text{new}\_\text{cost} = subtrees[k - j][j].\text{cost} +$$

$$+ subtrees[i - (k - j)][k].\text{cost} + \text{new}\_\text{cardinality}$$

if $\text{new}\_\text{cost} < subtrees[i][j].\text{cost}$:

$$subtrees[i][j].\text{tree} = \text{new}\_\text{tree}($$

$$subtrees[k - j][j], subtrees[i - (k - j)][k])$$

$$subtrees[i][j].\text{cardinality} = \text{new}\_\text{cardinality}$$

$$subtrees[i][j].\text{cost} = \text{new}\_\text{cost}$$

return $subtrees[n][1].\text{tree}$

The authors of ZStream [MM09] introduced an efficient algorithm for producing tree-based plans based on dynamic programming (Algorithm 4.3). The algorithm consists of $n - 1$ steps, where during the $i^{th}$ step the tree-based plans for all subsets of the pattern of size $i + 1$ are calculated (for the trees of size 1, the only possible tree containing the lone leaf is assumed). During this calculation, previously memoized results for the two subtrees of each tree are used. Memoization is implemented using a two-dimensional matrix with each entry $(i,j)$ containing the cost of the $j^{th}$ subtree of size $i$. To calculate the cost of a tree $T$ with the subtrees $L$ and $R$, the following formula is used:

$$\text{Cost}\left(T\right) = \begin{cases} 
  r_i & \text{if } T \text{ is a leaf} \\
  \text{Cost}\left(L\right) + \text{Cost}\left(R\right) + \text{Card}\left(L, R\right) & \text{otherwise},
\end{cases}$$

where $\text{Card}\left(L, R\right)$ is the cardinality (the expected number of partial matches reaching the root) of $T$, whose calculation depends on the operator applied by the root. For example, the cardinality of a conjunction node is defined as the product of the cardinalities of its operands multiplied by the total selectivity of the conditions between the events in $L$.
and the events in $R$. That is,

$$\text{Card}(T) = \text{Card}(L) \times \text{Card}(R) \times \text{SEL}(L,R),$$

where $\text{SEL}(L,R)$ is a product of all predicate selectivities $sel_{i,j}: i \in L, j \in R$. Leaf cardinalities are defined as the arrival rates of the respective event types. The reader is referred to [MM09] for more details.

To apply the invariant-based method on this algorithm, we will define each internal node of a tree-based plan as a building block. This way, up to $O(n^3)$ blocks will be formed during the run of Algorithm 4.3, with only $O(n)$ included in the resulting plan.

A comparison between the costs of two trees will be considered a block-building comparison for the root of the less expensive tree. The deciding conditions for this algorithm will be thus defined simply as $\text{Cost}(T_1) < \text{Cost}(T_2)$, where $T_1, T_2$ are the two compared trees. These comparisons are invoked at each step during the search for the cheapest tree over a given subset of events. For $k$ events, the number of candidate trees is $C_{k-1} = \frac{(2k-2)!}{(k-1)!k!}$, where $C_m$ is the $m^{th}$ Catalan number. Therefore, picking only one comparison as an invariant and dismissing the rest of the candidates may create a problem of false negatives, and K-invariant method is recommended instead (see discussion in Section 4.3.3).

The obvious problem with the above definition is that tree cost calculation is a recursive function, which contradicts our constant-time invariant verification assumption. We will eliminate this recursion by utilizing the following observation. In Algorithm 4.3, all block-building comparisons are performed on pairs of trees defined over the same set of event types. By the definition of an invariant, one of these trees is always a subtree of a plan currently being in use. Recall that invariants on tree-based plans are always verified in the direction from leaves to the root. Hence, if any change was detected in one of the statistics affecting the subtrees of the two compared trees, it would be noticed during verification of earlier invariants. Thus, it is safe to represent the cost of a subtree in an invariant as a constant whose value is initialized to the cost of that subtree during invariant creation (i.e., plan construction).

### 4.4.3 General Applicability of the Invariant-Based Method

The approaches described in Sections 4.4.1 and 4.4.2 only cover two special cases. Here, we generalize the presented methodologies to apply the invariant-based method to any greedy or dynamic programming algorithm. We also discuss the applicability of our method to other algorithm categories.

A generalized variation of the technique illustrated in Section 4.4.1 can be utilized for any greedy plan generation algorithm. To that end, a part of a plan constructed during a single greedy iteration should be defined as a building block. Additionally, a conjunction of all conditions evaluated to select a specific block is to be defined as a block-building comparison associated with this block. Since most greedy algorithms
require constant time and space for a single step, the complexity requirements for the invariant verification will be satisfied.

Using similar observations, we can generalize the approach described in Section 4.4.2 to any dynamic programming algorithm. A subplan memoized by the algorithm will correspond to a building block. A comparison between two subplans will serve as a BBC for the block that was selected during the initial run.

In general, the invariant-based method can be similarly adapted to any algorithm that constructs a plan in a deterministic, bottom-up manner, or otherwise includes a notion of a “building block”. To the best of our knowledge, the majority of the proposed solutions share this property.

In contrast, algorithms based on local search (such as ‘IL-RANDOM’ and ‘IL-GREEDY’ adapted to CEP in Chapter 3) cannot be used in conjunction with the invariant-based method. Rather than building a plan step-by-step, these algorithms start with a complete initial solution and modify it to create an improved version until some stopping condition is satisfied [AL97].

4.5 Experimental Evaluation

In this section, the results of our experimental evaluation are presented. The objectives of this empirical study were twofold. First, we wanted to assess the overall system performance achieved by our approach and the computational overhead implied by its adaptation process as compared to the existing strategies for ACEP systems, outlined in Section 4.1. Our second goal was to explore how changes in the parameters of our method and of the data characteristics impact the above metrics.

4.5.1 Experimental Setup

We implemented the two CEP models described in Section 4.4, the lazy NFA with the greedy order-based algorithm [Swa89] and the ZStream model with tree-based dynamic programming algorithm [MM09] (referred to as ‘DP-LD’ and ‘ZSTREAM’ respectively in the experiments described in Section 3.7). We also added support for three adaptation methods (i.e., implementations of $D$): 1) the unconditional reoptimization method utilized by the Tree NFA from Section 2.6; 2) the constant-threshold method from [MM09]; 3) the invariant-based method. To accurately estimate the event arrival rates and predicate selectivities on-the-fly, we utilized the algorithm presented in [DGIM02] for maintaining statistics over sliding window.

Since the plan generation algorithms used during this study create plans optimized for maximal throughput, we choose throughput as a main performance metric, reflecting the effectiveness of the above algorithms in the presence of changes in the input. We believe that similar results could be obtained for algorithms targeting any other optimization goal, such as minimizing latency or communication cost.
Two real-world datasets described in Sections 2.7 and 3.7 were again used in the experiments. The first dataset contains the vehicle traffic sensor data, provided by City of Aarhus, Denmark [AGM15]. Each event represents an observation of traffic at the given point. The arrival rates and selectivities for this dataset were highly skewed and stable, with few on-the-fly changes. However, the changes that did occur were mostly very extreme. The second dataset was taken from the NASDAQ stock market historical records [EOD]. Each record in this dataset represents a single update to the price of a stock, spanning a 1-year period. Contrary to the traffic dataset, low skew in data statistics was observed, with the initial values nearly identical for all event types. The changes were highly frequent, but mostly minor.

For each of the datasets, we created 5 sets of patterns containing different operators (Section 4.2.1), as follows: (1) sequences; (2) sequences with an additional event under negation; (3) conjunctions; (4) sequences with a single event under Kleene closure; (5) composite patterns, consisting of a disjunction of three sequences. Each set contained 6 patterns of sizes varying from 3 to 8. We defined pattern size as the number of events in a pattern for sets 1-4 and the number of events in each subpattern for set 5. Our main results presented in this section are averaged over all pattern sets unless otherwise stated. Full description of the specific results obtained for each set is provided in Appendix D.

The structure of the patterns for both datasets was identical to the one defined in Section 3.7.2. For the traffic dataset, we requested to detect combinations (sequences, conjunctions, etc., depending on the operator involved) of three or more observations with either an increase or a decline in both the number of vehicles and the average speed. For the stock dataset, the patterns to evaluate were defined as combinations of different stock identifiers (types), with the predefined price differences (e.g., for a conjunction pattern \( \text{AND} (A, B, C) \) we require \( A.\text{diff} < B.\text{diff} < C.\text{diff} \)).

All models and algorithms under examination were implemented in Java. All experiments were run on a machine with 2.20 Ghz CPU and 16.0 GB RAM.

4.5.2 Experimental Results

In our first experiment, we evaluated the performance of the invariant-based method for different values of the invariant distance \( d \), obtained by parameter scanning (Section 4.3.4). In this experiment, only the sequence pattern sets were used. For each of the four possible dataset-algorithm combinations, the system throughput was measured as a function of the tested pattern size and of \( d \), with its values ranging from 0 (which corresponds to the basic method) to 0.5.

The results are displayed in Figure 4.5. It can be observed that in each scenario there exists an optimal value \( d_{\text{opt}} \), which depends on the data and the algorithm in use, consistently outperforming the other values for all pattern sizes. For distances higher than \( d_{\text{opt}} \), too many changes in the statistics are undetected, while the lower values trigger unnecessary adaptations. Overall, the throughput achieved by using invariants
Figure 4.5: Throughput of the invariant-based method for different dataset-algorithm pairs as a function of the pattern size and the invariant distance $d$: (a) traffic dataset / greedy algorithm; (b) traffic dataset / ZStream algorithm; (c) stocks dataset / greedy algorithm; (d) stocks dataset / ZStream algorithm.

with distance $d_{opt}$ is 2 to 25 times higher than that of the basic method ($d = 0$).

Then, we validated the average relative difference method described in Section 4.3.4 by comparing its output value $d_{avg}$ to $d_{opt}$ (obtained via parameter scanning as described above) for each scenario. The results are summarized in Table 4.1.

For the traffic dataset, the computed values were considerably close to the optimal ones for patterns of length 6 and above, with precision reaching at least 87% (for ZStream algorithm and pattern length 7) and as high as 92% (Greedy algorithm, length 8). For the stocks dataset, the achieved accuracy was only 31-44%. We can thus conclude that the tested method does not function well in presence of low data skew, matching our expectations from Section 4.3.4. This highlights the need for developing better solutions, which is the goal of our future work.

It can also be observed that for all dataset-algorithm combinations the prevision of the average relative difference method increases with pattern size. We estimate that the scalability of this method would further increase for even larger patterns.

Next, we performed an experimental comparison of all previously described adaptation methods. The comparison was executed separately for each dataset-algorithm combination. For the invariant-based method, the $d_{opt}$ values obtained during the first
Table 4.1: Quality of distance estimates obtained by the average relative difference method

<table>
<thead>
<tr>
<th>Dataset</th>
<th>Algorithm</th>
<th>Pattern Size</th>
<th>$d_{avg}$</th>
<th>$d_{opt}$</th>
<th>$\min\left(\frac{d_{avg}}{d_{opt}}, \frac{d_{opt}}{d_{avg}}\right)$</th>
</tr>
</thead>
<tbody>
<tr>
<td>Traffic</td>
<td>Greedy</td>
<td>4</td>
<td>0.1695</td>
<td>0.1</td>
<td>0.59</td>
</tr>
<tr>
<td>Traffic</td>
<td>Greedy</td>
<td>5</td>
<td>0.1211</td>
<td>0.1</td>
<td>0.826</td>
</tr>
<tr>
<td>Traffic</td>
<td>Greedy</td>
<td>6</td>
<td>0.1163</td>
<td>0.1</td>
<td>0.86</td>
</tr>
<tr>
<td>Traffic</td>
<td>Greedy</td>
<td>7</td>
<td>0.0909</td>
<td>0.1</td>
<td>0.909</td>
</tr>
<tr>
<td>Traffic</td>
<td>Greedy</td>
<td>8</td>
<td>0.0921</td>
<td>0.1</td>
<td>0.921</td>
</tr>
<tr>
<td>Traffic</td>
<td>ZStream</td>
<td>4</td>
<td>0.9153</td>
<td>0.4</td>
<td>0.437</td>
</tr>
<tr>
<td>Traffic</td>
<td>ZStream</td>
<td>5</td>
<td>0.6514</td>
<td>0.4</td>
<td>0.614</td>
</tr>
<tr>
<td>Traffic</td>
<td>ZStream</td>
<td>6</td>
<td>0.4551</td>
<td>0.4</td>
<td>0.879</td>
</tr>
<tr>
<td>Traffic</td>
<td>ZStream</td>
<td>7</td>
<td>0.4581</td>
<td>0.4</td>
<td>0.873</td>
</tr>
<tr>
<td>Traffic</td>
<td>ZStream</td>
<td>8</td>
<td>0.4464</td>
<td>0.4</td>
<td>0.896</td>
</tr>
<tr>
<td>Stocks</td>
<td>Greedy</td>
<td>4</td>
<td>0.0556</td>
<td>0.2</td>
<td>0.278</td>
</tr>
<tr>
<td>Stocks</td>
<td>Greedy</td>
<td>5</td>
<td>0.0268</td>
<td>0.2</td>
<td>0.134</td>
</tr>
<tr>
<td>Stocks</td>
<td>Greedy</td>
<td>6</td>
<td>0.0661</td>
<td>0.2</td>
<td>0.331</td>
</tr>
<tr>
<td>Stocks</td>
<td>Greedy</td>
<td>7</td>
<td>0.0818</td>
<td>0.2</td>
<td>0.409</td>
</tr>
<tr>
<td>Stocks</td>
<td>Greedy</td>
<td>8</td>
<td>0.0866</td>
<td>0.2</td>
<td>0.443</td>
</tr>
<tr>
<td>Stocks</td>
<td>ZStream</td>
<td>4</td>
<td>0.095</td>
<td>0.4</td>
<td>0.24</td>
</tr>
<tr>
<td>Stocks</td>
<td>ZStream</td>
<td>5</td>
<td>0.102</td>
<td>0.4</td>
<td>0.255</td>
</tr>
<tr>
<td>Stocks</td>
<td>ZStream</td>
<td>6</td>
<td>0.1231</td>
<td>0.4</td>
<td>0.308</td>
</tr>
<tr>
<td>Stocks</td>
<td>ZStream</td>
<td>7</td>
<td>0.1563</td>
<td>0.4</td>
<td>0.391</td>
</tr>
<tr>
<td>Stocks</td>
<td>ZStream</td>
<td>8</td>
<td>0.1206</td>
<td>0.4</td>
<td>0.304</td>
</tr>
</tbody>
</table>

experiment were used. For the constant-threshold method, an optimal threshold $t_{opt}$ was empirically found for each of the above combinations using a similar series of runs.

Figures 4.6-4.9 summarize the results. Each figure represents the measurements for a particular dataset-algorithm combination and contains four graphs, presenting different statistics as a function of the pattern size. The first graph shows the throughput achieved using each of the adaptation methods. Here, we have also included the “static” method in our study, where no adaptation is supported and the dataset is processed using a single, predefined plan. The second graph is a different way of viewing the previous one, comparing the adaptation methods by the relative speedup they achieve over the “static plan” approach. The third graph depicts the total number of reoptimizations (actual plan replacements) recorded during each run. Finally, we report the computational overhead of each method, that is, a percentage of the total execution time spent on executions of $D$ and $A$ (i.e., checking whether a reoptimization is necessary and computing new plans).

The throughput comparison demonstrates the superiority of the invariant-based method over its alternatives for all scenarios. Its biggest performance gain is achieved in the traffic scenario, characterized by high skew and major statistic shifts (Figures 4.6-4.7). This gain reaches its peak for larger patterns, with the maximal recorded performance of more than 6 times that of the second-best constant-threshold method:
the greater the discrepancy between the data characteristics, the more difficult it is to find a single threshold to accurately monitor all the changes. Since this discrepancy may only increase as more statistic values are added to the monitored set, we expect the superiority of this method to keep growing with the pattern size beyond the values we experimented with. Figures 4.6(b) and 4.7(b) provide a clear illustration of the above phenomenon and of the invariant-based method scalability. Note also that, for larger pattern sizes, the constant-threshold method nearly converges to the unconditional one due to the increasing number of false positives it produces.

For the stocks dataset (Figures 4.8-4.9), the throughput measurements for the constant-threshold and the invariant-based methods are considerably closer. Due to the near-uniformity of the statistic values and of their variances, finding a single $t_{opt}$ is sufficient to recognize most important changes. Hence, the precision of the constant-threshold method is very high on this input. Nevertheless, the invariant-based method achieves a performance speedup for this dataset as well (albeit only about 30-60%) without adding significant overhead. Also, for the same reason, the static plan performs reasonably well in this scenario, decidedly outperforming the unconditional method. The latter suffers from extreme over-adapting to the numerous small-scale statistic shifts.

The total number of reoptimizations performed in each scenario (Figures 4.6(c),
Figure 4.7: Comparison of the adaptation methods applied on the traffic dataset in conjunction with ZStream algorithm: (a) throughput; (b) relative throughput gain over the non-adaptive method; (c) total number of plan reoptimizations; (d) computational overhead.

Figure 4.8: Comparison of the adaptation methods applied on the stocks dataset in conjunction with the greedy algorithm: (a) throughput; (b) relative throughput gain over the non-adaptive method; (c) total number of plan reoptimizations; (d) computational overhead.
Figure 4.9: Comparison of the adaptation methods applied on the stocks dataset in conjunction with ZStream algorithm: (a) throughput; (b) relative throughput gain over the non-adaptive method; (c) total number of plan reoptimizations; (d) computational overhead.

Figure 4.9(c), 4.8(c), 4.9(c)) backs up and augments the above results. The invariant-based method consistently requires few plan replacements while also achieving the best throughput. The extremely high numbers produced by the unconditional strategy lead to its poor performance. For the traffic dataset, the constant-threshold method tends to approach these numbers for larger patterns. This can either be a sign of multiple false positives or over-adapting. For the stocks dataset, this method becomes more similar to the invariant-based one, executing nearly identical reoptimizations.

Figures 4.6(d), 4.7(d), 4.8(d) and 4.9(d) present the computational overhead of the compared approaches. Here, the same behavior is observed for all dataset-algorithm combinations. While the invariant-based and the constant-threshold methods consume negligible system resources, unconditional reoptimization results in up to 11% of the running time devoted to the adaptation process.

As evident by the experiments with stock market data (Figures 4.8-4.9), smaller number of reoptimizations and lower computational overhead do not necessarily result in better overall system performance. On this dataset, the invariant-based method achieves the highest throughput despite a slightly higher overhead as compared to the second-best constant-threshold method. This can be attributed to the false negatives of the latter, that is, cases in which it missed a reoptimization opportunity and kept using an old plan despite a better one being available.
In all experiments, the relative gain of the invariant-based method was considerably higher for ZStream algorithm than for the greedy one. There are two reasons for this result. First, the more complex structure of the tree-based plans makes it more difficult to capture the dependencies between plan components without fine-grained invariants. Second, as this algorithm is more computationally expensive, the penalty for a redundant reoptimization is higher. Following these observations, we believe that the invariant-based method is capable of achieving even larger benefit for more advanced and precise (and hence more complex) plan generation algorithms. Utilizing this method will thus encourage the adoption of such algorithms by CEP engines.
Chapter 5

Real-Time Multi-Pattern Detection over Event Streams

5.1 Introduction

In previous chapters of this thesis, we presented a number of novel approaches for optimizing the process of detecting a single complex pattern in an event stream. However, most real-life CEP engines are typically required to support efficient simultaneous tracking of hundreds to thousands of patterns in multiple high-speed input streams of events. We will refer to systems possessing this functionality as multi-pattern complex event processing (MCEP) systems. Most MCEP systems also operate under tight real-time constraints, where a pattern match must be reported within milliseconds of its occurrence. In such scenarios, more advanced and sophisticated optimization techniques are required.

We will illustrate the above with the following example.

Example 5.1.1. Consider a security system tasked with monitoring a corporate building. Every room entrance is equipped with a sensor that emits a signal to the main controller whenever any large object passes through the doorway. We are interested in tracking a set of abnormal movement scenarios, in which a person (possibly an intruder) proceeds via a forbidden path inside the building. Whenever a restricted path traversal is detected through a sequence of observations from different cameras, an alarm must be raised. In presence of multiple prohibited paths, a MCEP system could specify a dedicated pattern for each monitored path. As some paths might intersect or contain interleaving sections, the respective patterns could possibly share common subexpressions.

Figure 5.1(a) depicts two possible NFAs for detecting one such scenario, in which an intruder is detected near doorway A, then immediately passes through entrance B, and finally enters doorway C. This pattern can be formulated as a sequence of three primitive events, each corresponding to getting a signal from a sensor corresponding to entrances A, B, and C respectively.
The presented automata are similar to those demonstrated in previous examples (Chapters 1-4). The first NFA (Figure 5.1(a)) utilizes an eager approach. The detection is triggered by the arrival of a signal from sensor A. For each accepted signal, the stream of events from sensor B is probed. If a new signal is subsequently received from B, we wait for a corresponding event from sensor C.

The second automaton (Figure 5.1(b)) is a lazy Chain NFA (Section 2.4), assuming that sensor C generates significantly fewer signals than A and B do. Instead of following the order $A \rightarrow B \rightarrow C$ specified by the pattern, it waits for a signal from C, then examines the local history for previous signals received from sensors B and A. As discussed in previous chapters, this improved plan allows to boost the system performance by generating drastically fewer partial matches.

Lazy evaluation method that we presented in previous chapters is a representative of a broader class of optimization strategies focused on modifying the order in which the events are processed. Cost-based evaluation trees [MM09], thoroughly discussed in Chapters 3 and 4, serve as another prominent example. From now on, we will refer to such methods as pattern rewriting methods.

Multi-pattern CEP systems can also significantly benefit from utilizing pattern sharing methods [AcT08, DGH+06, LRG+11, RLR16, ZVDH17], a class of optimization techniques utilizing the structural similarities between different patterns to unify the processing of common subexpressions. Figure 5.2 illustrates this approach. For
Figure 5.3: NFA optimization example for event sequences A,B,C,D and A,E,C,F: (a) no sharing or reordering; (b) reordering without sharing; (c) sharing without reordering; (d) a combination of reordering and sharing.

presentational purposes, we omit ‘ignore’ edges and ‘accept’ labels from now on. We are required to monitor a pair of patterns $P_1 : A \rightarrow B \rightarrow C$ and $P_2 : A \rightarrow B \rightarrow D$. Instead of processing them independently (Figure 5.2(a)), the system can merge the first three states of the respective NFAs to produce a joint automaton (Figure 5.2(b)). This optimization avoids duplicate instantiating and storing of partial matches, since each partial match of $P_1$ is also a partial match of $P_2$ and vice versa.

Pattern reordering and pattern sharing have evolved independently. They are generally considered as orthogonal techniques and cover different aspects of CEP performance optimization. This also implies that each of the two methods overlooks certain opportunities exploited by the other. A fusion of sharing and reordering could discover evaluation plans that would not be considered otherwise. More specifically, incorporating pattern reordering techniques can create additional sharing opportunities.

We will illustrate the above using the following example. The system is given two patterns depicted in Figure 5.3(a). Reordering the patterns by the ascending order of event arrival rates might result in a pair of locally optimal NFAs (Figure 5.3(b)). Alternatively, a global shared plan shown in Figure 5.3(c) can be obtained by sharing the first two states of each automaton. Now consider a combined application of the above techniques, where the NFAs are first reordered to maximize the common prefix length, and then this newly created subpattern is shared. Figure 5.3(d) shows the resulting plan. This plan would never be created if only one of the two optimizations was employed, or if they were used independently.

Despite the benefits of combining pattern sharing with rewriting for MCEP optimization, surprisingly little work has been done in this direction. The only related publications known to the authors either limit the discussion to a single shared subpattern [AcT08] or consider a restricted model for pattern decomposition without reordering [ZVDH17].

In this chapter, we demonstrate a novel approach for utilizing lazy evaluation and plan-based pattern detection techniques established in previous chapters to achieve a
drastic improvement in MCEP scenarios. Rather than merely maximize the sharing degree or create locally optimal plans, we aim to produce a globally optimal plan for the given workload of patterns using a mixture of the two. At the core of our framework lies the optimizer that uses sharing and reordering techniques as atomic building blocks to generate candidate evaluation plans. This fusion allows us to take advantage of subexpressions not normally considered for sharing. Our approach considers all possible plans, including those that avoid sharing.

To traverse the hyperexponential space of evaluation plans, we incorporate a novel method based on the local search paradigm [AL97]. As opposed to the traditional MCEP optimizers, our system can operate under arbitrarily tight time constraints due to the inherent balance between optimization time and solution quality. In addition, our architecture allows for easy implementations of extensions addressing various practical challenges of CEP system deployment, such as dynamically fluctuating arrival rates and selectivities, addition and removal of patterns on-the-fly, and the need to comply with the SLAs (service level agreements) specified for certain patterns. To the best of our knowledge, no existing work addresses these considerations in a multi-pattern CEP context.

The contributions of this chapter can thus be summarized as follows:

- We present a novel approach for optimizing large-scale MCEP systems by combining the power of state-of-the-art pattern sharing and reordering (lazy evaluation) techniques.
- We design a set of algorithms for efficiently searching the solution space. Our algorithms are highly precise and their execution time can be arbitrarily limited, making them perfectly applicable for real-time environments.
- We implement a MCEP engine capable of utilizing the plans created by our optimizer for efficient pattern detection.
- We show how advanced features, such as user-defined SLAs and adaptive behavior, can be easily supported by our optimizing framework.
- We empirically validate the performance of our solution, demonstrating its scalability and superiority over existing state-of-the-art approaches.

The remainder of this chapter is organized as follows. Section 5.2 provides the necessary background on MCEP and introduces the notations. In Section 5.3 we describe our system design and present a limited version of our optimization mechanism, restricted to prefix sharing. Section 5.4 presents the algorithmic framework for selecting the best evaluation plan from the solution space. Section 5.5 extends the basic method from Section 5.3 to consider sharing of arbitrary subsets. In Section 5.6, advanced extensions of our system are discussed. We report the results of our extensive experimental study in Section 5.7.
5.2 Background and Terminology

Formally, a MCEP system accepts three parameters: an input data stream $I$, a set of patterns $WL$ to be monitored (which we will refer to as the system workload), and a set of event type statistics $Stat$. The input stream $I = \{e_1, e_2, \cdots \}$ is an ordered, possibly infinite temporal sequence of primitive events, or simply events. Since in real-time scenarios the events can arrive from multiple sources, we define $I$ as a “logical” input source, possibly encapsulating multiple merged substreams. Each event $e_i$ is represented by a well-defined type and a set of attributes, including the occurrence timestamp. In Example 5.1.1, the event type is specified by the origin sensor ID, and the attribute set may include the movement speed of an intruder or the direction of passing through the doorway.

The workload $WL = \{P_1, \cdots, P_n\}$ contains a finite number of patterns the system is requested to detect. In this chapter, we utilize a slightly different formal definition for a pattern. Each pattern in $WL$ is defined by the tuple $P_i = (E_i, S_i, C_i, W_i)$, where $E_i = \{E_1, \cdots, E_m\}$ is the set of event types participating in $P_i$, $S_i$ denotes the structure of $P_i$ (which will be defined shortly), $C_i$ is the condition set specifying the constraints on the attribute values of the events, and $W_i$ is the time window defined for this pattern, that is, the maximal allowed time difference between the timestamps of a pair of events in a match.

The structure $S_i$ specifies how the events requested by the pattern are to be assembled to form a match. It is defined by a combination of event types and operators. Similarly to previous chapters, we will consider the most common operators such as AND, SEQ, and OR. The AND operator requires the occurrence of all events specified in the pattern. The SEQ operator also expects the events to appear in a predefined temporal order. The OR operator corresponds to the appearance of any event out of those specified. Two additional important operators are the negation operator (NOT), requiring the absence of an event from some position in the match, and the Kleene closure operator (KL), accepting one or more instances of an event.

To illustrate the above, the structure of the pattern from Figure 5.1 could be summarized as $SEQ(A, B, C)$, with $E = \{A, B, C\}$. If the order of receiving the signals was not important, the pattern would be formulated as $AND(A, B, C)$. In addition, assume that a signal arriving from the sensor D indicates the arrival of a security guard to the area, in which case no alarm should be set even if an intruder is present. Then, the structure of our pattern would become $SEQ(AND(A, B, C), NOT(D))$.

In the general case, $S_i$ is an arbitrary expression over the operators presented above. As discussed in Sections 2.5.2 and 3.5.4, such patterns can be simplified by the transition to DNF form. From the standpoint of a MCEP system, every clause of the resulting DNF expression can be considered as a separate pattern in a workload. In addition, a clause containing multiple AND/SEQ operators can be flattened to a simple expression featuring a single AND or SEQ operator with possible NEG and KL operators on single
events. Therefore, we will only discuss pattern structures of the above form from now on.

Stat, first defined in Section 4.2.1, is a set of statistical data properties that are used by the MCEP engine during evaluation plan generation. In Example 5.1.1, Stat might contain the arrival rates of all event types (that is, of signals from each sensor). As depicted in Figure 5.1, this knowledge can be utilized to produce the optimal evaluation order by sorting the arrival rates in an ascending order. In addition, we will consider as members in Stat the selectivities of the conditions defined by the patterns. The selectivity of a condition is defined as the probability of the input tuple to successfully pass the condition. More formally,

\[
Stat = \{ r_x | \exists P_i \in WL : E_x \in \mathcal{E}_i \} \cup \\
\{ sel_{x,y}^C | \exists P_i \in WL : E_x, E_y \in \mathcal{E}_i \},
\]

where \( r_x \) is the arrival rate of the event type \( E_x \), and \( sel_{x,y}^C \in [0,1] \) is the selectivity of a mutual condition between \( E_x \) and \( E_y \) in some condition set \( C \) (we set \( sel_{x,y}^C = 1 \) if no condition is defined between the event types). Our results can be trivially extended to additional parameters, such as inter-event dependencies and costs of predicate evaluation, by modifying the cost model (see below).

The general architecture of a MCEP system is depicted in Figure 5.4. The two main
components of a MCEP system are the evaluation mechanism and the optimizer. The *evaluation mechanism* is responsible for the actual processing of the input stream I. An evaluation mechanism of choice in Figures 5.1-5.3 is a lazy NFA from Section 2.4. Since NFAs discussed in previous works are only capable of tracking a single pattern, an extension for multiple patterns will be presented in Section 5.3. A different evaluation mechanism, superior to NFAs in terms of expressiveness and efficiency, will be discussed in Section 5.5.

At runtime, a plan-based evaluation mechanism follows an *evaluation plan* supplied by the optimizer. We distinguish between *local evaluation plans* applicable for single-pattern evaluation mechanisms only, and *global evaluation plans* that consider a workload of patterns. For example, the plans applied by the NFAs in Figures 5.1(a) and 5.1(b) are local evaluation plans, whereas Figures 5.2 and 5.3 illustrate global evaluation plans.

Different evaluation mechanisms support different types of evaluation plans. As described in Chapter 3, creating a Chain NFA for a single pattern requires an *order-based* local evaluation plan. For a pattern P over the event types \(E_1, \ldots, E_m\), the order-based evaluation plan is an ordering \(O = (E_{q_1}, \ldots, E_{q_m})\), where \(q_1, \ldots, q_m\) is a permutation of \([1, \ldots, m]\). The structure of global order-based evaluation plans will be discussed in the next section.

The task of the *optimizer* is to create a global evaluation plan upon system initialization. The resulting plan is then transferred to the evaluation mechanism, which subsequently launches the detection process on a stream I. The optimizer typically uses a predefined *cost function* to measure the quality of a plan subject to the given workload WL and the statistics collection Stat. We will define this function as

\[
Cost : \mathcal{P} \times \mathcal{W} \times STAT \rightarrow \mathbb{R},
\]

where \(\mathcal{P}, \mathcal{W}, STAT\) are the sets of all global evaluation plans, workloads, and combinations of statistical values, respectively. The cost assigned by this function to a global evaluation plan may reflect performance metrics such as throughput, detection latency, network communication cost, power consumption, and more.

### 5.3 Multi-Pattern CEP with Prefix Sharing

In this section, we present the core principles and algorithms behind our MCEP system. We discuss an extension to NFA for handling multiple patterns, analyze the complexity of event detection under this mechanism, and formally define the cost model and the optimization problem to be solved by the optimizer. For presentational purposes, we describe a limited version of our method, only considering prefix sharing opportunities between patterns. In Section 5.5, we extend our framework to support arbitrary subexpression sharing. The experimental study in Section 5.7 is conducted on this extended system.
5.3.1 Multi-Pattern NFA Evaluation

Our framework processes all patterns in a workload using a single NFA, which we denote as the multi-pattern NFA. It is organized in a tree-like topology formed by merging the common prefixes of the chain-structured NFAs corresponding to each pattern in the workload. The root of the tree is shared between all patterns and serves as the initial state of the automaton. Each internal node can be shared between two or more patterns.

Since different patterns may have different time windows, we augment each state of the multi-pattern NFA with a special time window attribute, set to the largest time window among the patterns sharing the state. The system uses this attribute to decide whether a specific partial match has expired.

Figure 5.5 depicts three of the possible multi-pattern NFAs for a workload of two patterns, \(P_1: SEQ(A, B, C)\) and \(P_2: SEQ(A, B, D)\), with \(W_1 = 10\) and \(W_2 = 20\). As discussed in Section 5.1, some NFAs have more shared states, while others contain more states in total but provide more efficient evaluation paths for individual patterns.

For each pattern in a workload, a dedicated final state is defined. When the final state for some pattern \(P\) is reached, a match is reported for this pattern. Note that while the final state for a pattern is typically a tree leaf, this is not always the case. For example, in a workload consisting of \(SEQ(A, B, C)\) and \(SEQ(A, B)\), the final state for \(SEQ(A, B)\) is an internal node.

The evaluation process for multiple patterns is similar to the one presented in Chapter 2 for NFA-based single-pattern detection. As a new event \(e\) of type \(T\) enters
the system, it is evaluated against existing \emph{NFA instances}. An instance is defined by a combination of a unique state identifier and a partial match. The system starts with a single instance associated with the initial state and an empty match. All instances associated with states containing an outgoing transition for \( T \) are matched with \( e \). For every instance satisfying the conditions between the events (including \( e \)), a new instance is created containing the new match resulting from \( e \)’s addition and associated with the state to which the transition leads. When an instance corresponding to some final state is created, its match is reported to the end users. An instance exceeding the time window specified by its associated state is removed from the system.

Since the number of instances in a system processing a large workload may be huge, traversing all of them on every event arrival is impractical. Instead, for each event type \( T \) we define a list \( l_T \) to contain all states with an outgoing transition accepting \( T \). The size of \( l_T \) can never exceed the number of patterns in a workload containing \( T \) in their specification and will be substantially lower under an efficient sharing strategy that aims to merge states that process interleaving event types. At runtime, NFA instances are stored in a hash table according to their associated state, and the arrival of an event of type \( T \) only triggers the traversal of instances associated with states in \( l_T \). For example, the state lists of a multi-pattern NFA in Figure 5.5(b) are \( l_A = \{q_2,q_3\}, l_B = \{q_1\}, l_C = \{q_2\}, l_D = \{q_4\} \).

### 5.3.2 Multi-Pattern Tree

Global evaluation plans utilized by multi-pattern NFAs are similarly structured in a tree-like manner. We will refer to this plan type as the \emph{multi-pattern tree (MPT)}. Given a MPT, a multi-tree NFA is constructed by simply copying the structure of the former.

As described in Section 5.2, a MPT is created by the optimizer. As we will see in Section 5.4, our optimizer proceeds by creating an initial MPT and repeatedly modifying it. Hence, efficient creation and modification operations are crucial for minimizing the optimization cost. In implementing these operations, the core principle of MPT behavior is to unconditionally share all shareable prefixes of the supplied local evaluation plans (orders). To add an evaluation order \( O \) to an existing MPT, we iterate over \( O \) and only create a new node if no equivalent one exists. Two nodes are considered equivalent if and only if they correspond to identical sequences of event types, and if their edges specify identical conditions. Similarly, a plan is removed by iterating over the respective order and only deleting states that are not shared with other patterns.

Figure 5.6 illustrates an addition and a removal of a plan from a MPT. The complexity of both operations is \( O(m) \), where \( m \) is the length of the evaluation order.

Creating a MPT from a set of orders \( \{O_1, \ldots, O_n\} \) is implemented by iteratively adding the orders to an initially empty tree. This operation requires \( O(n \cdot \max(m_i)) \) time and space, where \( m_i \) is the length of \( O_i \).

Since MPTs merge all common prefixes, we can uniquely define a MPT by the tuple
Figure 5.6: MPT modification example: (a) a MPT from Figure 5.5(a) and a local plan for a pattern $SEQ(A,C,E)$; (b) the MPT following the addition of the new evaluation plan (the path corresponding to the newly added plan is highlighted); (c) the MPT after the local evaluation plan for $SEQ(A,B,C)$ is removed.

$(O_1, \cdots, O_n)$. Forcing some nodes not to be shared is only possible by modifying the individual evaluation orders. This way, careful selection of local evaluation plans by the optimizer can achieve the perfect balance between sharing degree and local evaluation plan quality.

5.3.3 Runtime Complexity and Multi-Pattern Cost Model

We will now analyze the runtime complexity of the MCEP evaluation process described above and derive the cost function definition for multi-pattern trees.

The total cost associated with processing a single event $e$ of type $T$ is the sum of two components: 1) the cost of combining $e$ with the existing partial matches and creating new instances as a result of successful matching; 2) the cost of purging the instances created as a result of $e$’s arrival upon their expiration. We will denote the former as $CP(T)$ and the latter as $CR(T)$.

Both functions depend on the expected number of instances active at the time of an event arrival. Reducing the number of instances (or, more generally, the size of intermediate results) is a common optimization goal in multiple fields, including database query optimization [CM95, SAC+79, Swa89] and complex event processing [SMMP09, MM09]. In Section 3.4.1, we developed a cost model to estimate this metric for single-pattern lazy NFA evaluation. For an order-based plan $O = (E_{q_1}, \cdots, E_{q_m})$ detecting a pattern $P = (E, S, C, W)$, this cost function is defined as:

$$
Cost_{ord}(O, P, Stat) = |E| \sum_{k=1}^{|E|} Cost_{ord}^k (O, P, Stat),
$$

where $Cost_{ord}^k$ is the cost of the $k^{th}$ state in the chain-based NFA following $O$, calculated
as follows:

\[ \text{Cost}^k_{\text{ord}} (O, P, \text{Stat}) = W^k \cdot \prod_{i=1}^{k} r_{q_i} \cdot \prod_{i,j \leq k; i \leq j} \text{sel}_{q_i,q_j}^C, \]

where \( r_i; i \in [1, m] \) and \( \text{sel}_{i,j}^C; i, j \in [1, n] \) are as defined in Section 5.2.

We will use the above definition to calculate the expected number of instances existing simultaneously at any given moment during MPT-based multi-pattern evaluation. Given a node \( N \), let \( P_N \) denote the path from the root of the MPT to \( N \) (by definition of a tree, there is always exactly one such path). For the root, we set \( P_R = \emptyset \). The total number of instances is the sum of numbers of instances associated with each NFA state (and hence with the corresponding MPT node), calculated as follows:

\[ \#\text{inst} (\text{MPT}, \text{WL}, \text{Stat}) = \sum_{N \in \text{MPT}} \text{Cost}_{\text{ord}}^{\left| P_N \right|} (P_N, \text{WL}, \text{Stat}). \]

Thus, to calculate the number of instances to be traversed upon arrival of an event of type \( T \), we need to sum the instances associated with the states in \( l_T \):

\[ \#\text{inst}_T (\text{MPT}, \text{WL}, \text{Stat}) = \sum_{S \in l_T} \text{Cost}_{\text{ord}}^{\left| P_N(S) \right|} (P_N(S), \text{WL}, \text{Stat}), \]

where \( N(S) \) denotes a node corresponding to \( S \) in \( \text{MPT} \).

The processing cost per event is now derived as follows. Let \( C_a \) be the cost of accessing an instance, \( C_n \) the cost of creating a new instance and inserting it into the data structure, and \( C_r \) the cost of removing an instance from the system. In addition, let \( C_v (T, P_N) \) denote the cost of verifying the conditions between a new event of type \( T \) and the events preceding \( T \) in \( P_N \), and let \( \text{Sel}_v (T, P_N) \) denote the total selectivity of the above conditions. To make \( C_v \) and \( \text{Sel}_v \) well-defined, we set \( C_v = \text{Sel}_v = 0 \) if \( T \notin P_N \). Then, the expected cost of processing a single event of type \( T \) is:

\[ CP (T) = \sum_{S \in l_T} \left( \text{Cost}_{\text{ord}}^{\left| P_N(S) \right|} (P_N(S), \text{WL}, \text{Stat}) \cdot (C_a + C_v (T, P_N(S)) + \text{Sel}_v (T, P_N(S)) \cdot C_n) \right). \]

To calculate the cost of removing the expired instances, we observe that the expected number of instances created in state \( S \) after processing a new event of type \( T \) is equal to \( \text{Sel}_v (T, P_N(S)) \). Thus, the cost of eventually removing these instances upon their expiration is:

\[ CR (T) = \sum_{S \in l_T} \text{Cost}_{\text{ord}}^{\left| P_N(S) \right|} (P_N(S), \text{WL}, \text{Stat}) \cdot \text{Sel}_v (T, P_N(S)) \cdot C_r. \]

\(^1\)The presented function is the basic version of \( \text{Cost}_{\text{ord}} \) and it only applies when no negation or Kleene closure operator appears in the pattern. The reader is referred to Section 3.5 for more details.
The above analysis emphasizes two main performance objectives of a MCEP system attempting to minimize the processing cost per event. First, the sharing degree needs to be maximized to reduce the sizes of the state lists \( l_T \). Second, the cost of the local evaluation plans in terms of the expected number of simultaneously existing instances has to be as low as possible. As illustrated in Figure 5.3, there might be a conflict between these two objectives, which we will solve by defining an optimization problem later on.

The extended formula for the expected number of instances represents the same parameter dependencies as does the expression \( CP(T) + CR(T) \). In particular, both expressions are exponential in the size of the longest path in the MPT, polynomial in the time window length, and linear in the total size of the tree, the arrival rate of each primitive event type, and the selectivity of each condition. Hence, we will use \( \#\text{inst} \) as our cost function for measuring the quality of MPTs:

\[
\]

### 5.3.4 MCEP Optimization Problem

We will now formally define the problem to be solved by the MCEP optimizer. We will start with some preliminary definitions. Given an order-based plan \( O \) for a pattern \( P \) and a multi-pattern tree \( MPT \), we say that \( O \in MPT \) if and only if \( MPT \) contains a path \( P \) of length \( |O| \), starting at the root and ending at some final state, such that the event types and the conditions specified on the transitions in \( P \) are identical to those of a NFA detecting \( P \) according to \( O \). For example, a MPT in Figure 5.6(b) satisfies \( O_3 = (A, C, E) \in MPT \). Also, we will denote by \( \text{ORD}_P \) the set of all valid order-based evaluation plans for \( P \). For a pattern of size \( m \), \( |\text{ORD}_P| = m! \).

We are now ready to define our optimization problem.

**Tree-based MCEP optimization problem (T-MCEP).** Given a workload \( WL \) of \( n \) patterns and a statistics collection \( Stat \), find a multi-pattern tree \( MPT \) minimizing the value of the cost function \( \text{Cost}^{\text{multi}}_{\text{ord}}(MPT, WL, Stat) \) subject to

\[
\forall P_i, 1 \leq i \leq n : \exists O \in \text{ORD}_P \text{ s.t. } O \in MPT.
\]

We will denote the path in the MPT corresponding to the evaluation order of a pattern \( P_i \) as \( P_i \).

We will now discuss the complexity of T-MCEP. It can be noted that for \( n = 1 \) our problem is equivalent to the single-pattern CEP optimization problem (SCEP), thoroughly discussed in previous work [MM09, REG11, SMMP09]. In Section 3.4, it was shown that SCEP is NP-complete by reducing it to the problem of join evaluation order generation. The NP-completeness of this latter problem was in turn proven by [CM95, IK84] through a reduction to the maximum clique problem. The maximum clique problem is not only known to be NP-complete, but is also hard to approximate.
It was demonstrated in [Hås96] that, unless $NP = ZPP$, no polynomial-time algorithm exists that approximates the problem within the factor of $n^{1-\varepsilon}$, where $n$ is the size of the graph. By correctness of the reductions, this result applies also to the SCEP problem, and, by generalization, to T-MCEP.

5.4 Optimization Framework for T-MCEP

T-MCEP is a computationally hard optimization problem, characterized by an enormously large solution space and multiple local minima. Therefore, advanced techniques are needed in order to produce a high-quality solution under tight restrictions common for real-time MCEP systems.

The algorithms employed by our optimizer to achieve this goal implement the local search paradigm [AL97, HS04]. Local search is a well-known approach for finding approximate solutions for hard optimization problems [AL97]. Instead of building a new solution step-by-step, local search methods execute heuristically guided random walks in the solution space and search for the cheapest solution subject to a predefined cost function. The algorithm starts in the *initial state*, representing some valid solution for the problem. At each consecutive step, a similar solution referred to as a *neighbor state* is inspected and its cost is evaluated. A local search algorithm may choose to move to some neighbor state, making it the *current state*. When a predefined stopping criterion is satisfied, the search terminates and the cheapest observed solution is returned. Local search methods are successfully applied for solving a wide range of problems, from the classic traveling salesman problem to code design and VLSI layout synthesis [AL97].

To the best of our knowledge, no prior work attempted to represent the task of stream or event processing optimization as a local search problem. Instead, related research efforts focused on utilizing heuristic approaches [RSSB00], dynamic programming [AcT08, MM09], local-ratio approximation algorithms [RLR16], and branch-and-bound methods [GMAK14, ZVDH17] for similar optimization problems with hyperexponential solution spaces.

Local search methods present several important benefits for real-time streaming applications, and in particular for MCEP. Most importantly, they offer a tradeoff between the quality of the returned solution and the running time of the search. Since the local search procedure keeps a “current best” solution at any point of its execution, it can always be interrupted due to expired time limit and will return a valid solution, albeit not necessarily the cheapest. This property makes local search methods an attractive choice for targeting the MCEP optimization problem under tight real-time constraints.

In addition, local search allows for easy and straightforward incorporation of essential features of a real-life system, such as service-level agreements, on-the-fly evaluation plan adaptation and dynamic workload modification. We will discuss these advanced capabilities in Section 5.6.

We start this section by describing a data structure for managing inter-pattern
sharing opportunities, which we denote as a *multi-pattern graph* (MPG). We then present
a set of algorithms utilizing the MPG and implementing the local search paradigm to
solve T-MCEP.

### 5.4.1 Multi-Pattern Graph

We will start with some preliminary definitions. Let \( \pi_X(Y) \) denote a projection of an
expression \( Y \) on a set of variables \( X \). \( Y \) can be either a pattern structure or a condition
set as defined in Section 5.2. For example, \( \pi_{\{B,D\}}(SEQ(A,B,C,D)) = SEQ(B,D) \).
Given a pattern \( P = (E, S, C, W) \), we will say that another pattern \( P' = (E', S', C', W') \)
is a subpattern of \( P \) (marked as \( P' \subseteq P \)) if \( E' \subseteq E \), \( S' = \pi_{E'}(S) \), \( C' = \pi_{E'}(C) \), and
\( W' \leq W \).

A common subpattern \( P_{ij} = (E_{ij}, S_{ij}, C_{ij}, W_{ij}) \) of two patterns \( P_i, P_j \) is a pattern
satisfying \( (P_i \subseteq P_j) \land (P_j \subseteq P_i) \), such that \( W_{ij} = \min(W_i, W_j) \). A maximal common
subpattern of \( P_i, P_j \) is a common subpattern \( P_{ij} \), such that no other common subpattern
\( P'_{ij} \) satisfies \( P_{ij} \subseteq P'_{ij} \). We will denote it by \( MP_{ij} \) from now on. In addition, we will
denote by \( \Gamma_{ij} \) the set of all subsets of \( MP_{ij} \), that is, all common subpatterns of \( P_i \) and
\( P_j \). Obviously, \( \Gamma_{ij} = \Gamma_{ji} \) for each \( i, j \). The above definitions are trivially extended to an
arbitrary number of intersecting patterns.

To illustrate the above notations, let \( P_1 : SEQ(A, B, C, D) \) and \( P_2 : SEQ(A, E, C, D) \).
Assume that both patterns have no conditions and \( W_1 = 10, W_2 = 20 \). Then,
\( SEQ(A, D), SEQ(C, D), \) and \( SEQ(A, C) \) with \( W = 10 \) are common subpatterns of \( P_1 \) and \( P_2 \), while \( SEQ(C, A) \) is a subpattern of neither, since it has a conflicting
structure. The maximal common subpattern is \( SEQ(A, C, D) \).

The *multi-pattern graph* \( MPG = (V, E) \) is a data structure capable of efficiently
collecting, maintaining, and retrieving the information regarding the mutual subpatterns
of \( P_1, \cdots, P_n \). For each pattern \( P_i \), MPG contains a vertex \( v_i \in V \). For each pair
of distinct patterns \( P_i, P_j \) with non-empty intersection (i.e., satisfying \( \Gamma_{ij} \neq \emptyset \)), an
undirected edge \( e_{ij} = (v_i, v_j, \Gamma_{ij}) \in E \) is defined. More formally, the multi-pattern
graph for the workload \( WL = \{P_1, \cdots, P_n\} \) is defined as \( MPG = (V, E) \), where

\[
V = \{v_i | P_i \in WL\},
\]

\[
E = \{e_i = (v_i, v_j, \Gamma_{ij}) | v_i, v_j \in V, \Gamma_{ij} \neq \emptyset\}.
\]

Figure 5.7 depicts a MPG for a workload of 6 patterns. For presentation clarity,
edges with maximal common subpattern of size 1 are not shown.

In the general case, a MPG is an arbitrary, not necessarily connected graph. However,
it can be noted that any algorithm solving T-MCEP can be activated separately on each
connected component, and the results can then be combined to produce the final plan.
Not only does this observation allow us to solve the problem much more efficiently in
Figure 5.7: A multi-pattern graph for a workload of 6 patterns. Edges corresponding to maximal common subpatterns of size 1 are not shown. The triplet $P_1$, $P_2$ and $P_3$ share a maximal common pattern $SEQ(A, C)$. $P_3$ and $P_4$ have two distinct maximal common subpatterns. $P_6$ is fully contained in $P_3$.

the presence of multiple components, but it also makes it possible to limit the discussion below to connected graphs.

To guarantee an efficient local search procedure, the MPG has to occupy small space. Moreover, addition and removal operations must be fast and low-cost, and likewise for the retrieval of pattern intersection information. However, the definition of the MPG as presented above introduces potential performance issues. The remainder of this section will outline these issues and discuss the solutions. Using a combination of several advanced optimizations, we will be able to implement a MPG with near constant cost of retrieval and worst-case linear cost of addition and deletion with near linear space complexity.

The most dominant issue associated with MPG implementation is its exponential space complexity. First, explicitly storing the set of common subpatterns $\Gamma_{ij}$ requires $O(2^s)$ memory, where $s$ is the size of the maximal common subpattern. This can be solved by only storing the $MP_{ij}$ instead, as the rest of the common subpatterns can be inferred from it. Second, when $m$ patterns share the same subpattern, the MPG will contain $\binom{m}{2}$ edges representing the same subpattern set. Consequently, directly instantiating the MPG in memory would be extremely inefficient.

We address this shortcoming by compact graph representation. Rather than explicitly store the vertices and the edges, for every distinct maximal common subpattern $MP$ of some set of patterns $\Gamma$, we keep $\Gamma$ in a hash table with $MP$ as a key. In addition, a second hash table maps a single pattern $P$ to a list of maximal common subpatterns with its peers in MPG. This data structure still contains all the necessary information, additionally providing near constant cost of retrieval and worst case linear cost of addition and deletion of patterns. The space occupied by both hash tables is $O(n \cdot \gamma)$, where $\gamma$ is the total number of distinct maximal common subpatterns in the workload. While the value of $\gamma$ can reach $n^2$ in the worst case (and even exceed it in some cases that we describe shortly), the way in which the hash tables are constructed makes it...
extremely unlikely for the space complexity to surpass $O(n^2)$.

Another potential performance bottleneck associated with the MPG is the resource-consuming operation of calculating the maximal common subpatterns for all pairs of patterns. We will utilize the following simple and efficient implementation. Given $P_i = (E_i, S_i, C_i, W_i)$ and $P_j = (E_j, S_j, C_j, W_j)$, first a simple set intersection $E_{ij}$ of $E_i$ and $E_j$ is calculated. Then, we project the conditions in $C_i$ and $C_j$ on $E_{ij}$ and compare the resulting condition sets. If the sets are not equal, we calculate their intersection and reduce $E_{ij}$ accordingly. The same procedure is then performed for $S_i$ and $S_j$. Overall, the worst-case complexity of this operation is $O(\max(|E_i|, |E_j|) + \max(|C_i|, |C_j|))$.

For example, consider the above calculation for the following workload:

$$P_1 : \text{AND}(A, B, C) ; C_1 = \{ A.x < 10 \} ,$$

$$P_2 : \text{AND}(A, D, C) ; C_2 = \{ A.x \geq 10 \} .$$

The intersection of event types in this case is $E_{12} = \{ A, B \}$. In addition, due to the conflicting conditions on $A$, the maximal common subpattern is reduced to $\text{AND}(B)$.

Note that multiple maximal common subpatterns may exist. For example, both $\text{SEQ}(A, B)$ and $\text{SEQ}(A, C)$ are the maximal intersections of the sequences $\text{SEQ}(A, B, C)$ and $\text{SEQ}(A, C, B)$. In this case, the MPG will store a list of maximal common subpatterns.

The worst-case complexity of computing all maximal common subpatterns is then $O(n^2 \cdot (s_{\text{max}} + c_{\text{max}}))$, where $s_{\text{max}}$ and $c_{\text{max}}$ denote the maximum sizes of a pattern in terms of events and conditions, respectively. When the patterns in the workload are very long, we can improve the common case performance by utilizing the triangle inequality exhibited by the problem. Namely, given three patterns $P_i, P_j, P_k$, it can be observed that $P_{ik} \supseteq (P_{ij} \cap P_{jk})$. Hence, after computing $MP_{ij}$ and $MP_{jk}$, we can start the calculation of $MP_{ik}$ with the maximal common subpattern of $MP_{ij}$ and $MP_{jk}$. This is especially useful when $MP_{ij} = MP_{jk}$.

### 5.4.2 Local Search Algorithms for T-MCEP

A local search problem is specified by a pair $(\varphi, f)$, where $\varphi$ is a set of feasible problem solutions and $f : \varphi \rightarrow \mathbb{R}$ is a cost function. The goal is then to find an optimal solution $s^*$ such that $f(s^*) \leq f(s)$ for all $s \in \varphi$. In the case of T-MCEP, $\varphi$ consists of all possible MPTs and $f \equiv \text{Cost}_{\text{multi}}^{\text{ord}}$.

The search starts from some initial solution $s_{\text{init}}$. Local search algorithms traverse the search space by exploring the neighborhood of the current solution. A domain-specific neighborhood function $N : \varphi \rightarrow 2^\varphi$ maps a solution to a set of its neighbors, i.e., solutions that can be obtained by performing a slight modification. The strategy for performing the search is determined by the meta-heuristic in use. A local search algorithm for
a given problem can be uniquely defined by a combination of a meta-heuristic and a neighborhood function. When a predefined stopping criterion is satisfied, the search terminates and the cheapest observed solution is returned.

We will now provide a brief description of the most widely used meta-heuristics. The reader is referred to [AL97, HS04] for more detailed information.

The most basic meta-heuristic is known as iteractive improvement. At each step, this method selects the cheapest neighbor according to the provided cost function. The search stops when a local minimum is reached. Since for large neighborhoods traversing all neighbors is infeasible, the maximal number of inspected neighbors is limited to a predefined constant $L$. While simple and intuitive, this algorithm is expected to perform poorly in search spaces characterized by multiple local minima. Unfortunately, this is one of the key characteristics of the MCEP optimization problem due to the sharing-reordering tradeoff discussed in Section 5.1. Hence, we will apply more sophisticated methods.

Many different local search algorithms were developed, making use of sophisticated heuristics and meta-heuristics to efficiently scan the solution space. The most prominent and widely employed methods are simulated annealing and Tabu search.

Simulated annealing extends the functionality of iterative improvement by also allowing limited non-improving moves. For each step (we define a step as an inspection of a single neighbor), a threshold $c_k$ is defined. When a better neighbor solution is selected, it is chosen to replace the current solution, in a manner similar to the iterative improvement algorithm. If the neighbor solution is more expensive, it is accepted with probability $\exp\left(-\frac{\Delta f}{c_k}\right)$, where $\Delta f$ is the difference between the costs of the old and the new solutions. The thresholds are chosen such that $c_k = \alpha \cdot c_{k-1}, \alpha < 1$. The algorithm starts with a sufficiently large $c_0$ and terminates when a predefined small value $c_k$ is reached. Before the start of the actual search, $c_0$ is set to the largest difference observed during evaluation of $I$ neighbors of $s_{init}$. In our experiments in Section 5.7, we used $\alpha = 0.99$ and $I = 10^3$ neighbors for setting the initial threshold.

Tabu search explores $L$ random neighbors during each step and moves to the best of them (according to the cost function). Visiting the same state twice is prohibited. To enforce that, previously visited solutions are stored in a dedicated tabu list (hence the name of the algorithm). The tabu list has a finite capacity $C$: when the number of stored solutions reaches $C$, oldest stored solutions are removed. The best solution $s^*$ observed during the run of the algorithm is returned. We used a memory list of capacity $C = 10^4$ and $L = 100$ during our experimental evaluation.

Both algorithms stop after reaching a predefined number of steps since the last improvement to $s^*$ or when the time expires. To study the tradeoff between evaluation time and solution quality, we only implemented the timestamp-based stop condition.

The remainder of this section focuses on our problem-specific neighborhood functions utilizing the information in the MPG to create candidate solutions.

It can be noted that the solution space of our problem is enormously large. For a
workload of size \( n \), there are \( \prod_{i=1}^{n} |P_i|! \) possible MPTs, where \( |P_i| \) denotes the number of event types in the \( i^{th} \) pattern. Fortunately, closer analysis of the solution space will allow us to immediately discard the overwhelming majority of the subplan combinations.

We can observe the following regarding the possible local evaluation orders for a pattern \( P_i \) in the shared workload. If no subset of \( P_i \) can be shared with other patterns, it only makes sense to select the most efficient evaluation order. Otherwise, for every shareable subpattern \( P' \subseteq P_i \), we have to consider an order that starts with the best order \( O' \) for \( P' \), then continues with the best order for the remainder of the pattern given \( O' \) as the prefix. Note that not only the maximal common subpatterns but also their subsets must be considered, including the empty subset (which is equivalent to the case when no such \( P' \) exists).

We will formally state the above in the following theorem.

**Theorem 5.1.** Let \( MPT_{opt} \) be the optimal multi-pattern tree for some workload \( W \). Then, for each path \( \mathcal{P}_i \) in \( MPT_{opt} \) corresponding to the pattern \( P_i \) at least one of the following holds: (1) \( \mathcal{P}_i \) is the optimal evaluation order for \( P_i \); (2) \( \mathcal{P}_i \) can be divided into a non-empty prefix \( Pref_i \) that is shared with at least one additional pattern and a non-shared suffix \( Suff_i \), and it is the most efficient local evaluation order for \( P_i \) out of those starting with \( Pref_i \).

The proof is straightforward by assuming that neither (1) nor (2) hold and showing that \( MPT_{opt} \) can be improved by modifying \( Suff_i \) to make \( \mathcal{P}_i \) the most efficient order starting with \( Pref_i \), which contradicts the optimality of \( MPT_{opt} \). Since \( Suff_i \) is not shared by definition, improving it necessarily leads to an improvement of \( MPT_{opt} \).

Theorem 5.1 reduces the maximal number of potential orders for a single pattern from \( |P_i|! \) to \( \sum_{j=1}^{n} |\Gamma_{ij}| \) (this also includes \( |\Gamma_{ii}| = 1 \), which is the most efficient local order for \( p_i \)). However, to apply the above strategy, an algorithm is required to calculate local evaluation plans as described above. We will assume the existence of a deterministic local plan generation algorithm \( \mathcal{A} \), capable of the following functionality:

1. Given a pattern \( P \) and the statistical event characteristics \( Stat \), return the cheapest local order-based evaluation plan \( \mathcal{O} \) subject to \( Cost_{ord} \).

2. Given a pattern \( P \), its subpattern \( P' \), an evaluation plan \( \mathcal{O}' \) for \( P' \), and the statistics collection \( Stat \), return the cheapest (subject to \( Cost_{ord} \)) local order-based evaluation plan \( \mathcal{O} \) starting with prefix \( \mathcal{O}' \).

Many algorithms answering the above requirements have been discussed in Chapter 3. Moreover, any greedy algorithm or an algorithm based on dynamic programming satisfies both conditions. While most algorithms are not guaranteed to produce an optimal result due to the NP-hardness of local evaluation plan generation, they provide empirically accurate approximations. In the example that we discussed in Section 5.1, \( \mathcal{A} \) is a simple sorting algorithm arranging the event types in the ascending order of their expected arrival frequencies.
With the above observation in mind, we will now define neighborhood functions for T-MCEP. The first function produces a neighboring solution by selecting a random edge \((v_i, v_j)\) in the MPG and a common subpattern \(P \in \Gamma_{ij}\). We restrict \(P\) to be different from the subpattern that is shared between \(P_i\) and \(P_j\) in the current MPT (however, its subpatterns are allowed). A neighbor will be generated by invoking \(A\) to create new evaluation orders \(O_i, O_j\) sharing a common prefix \(O_P\), and replacing \(P_i, P_j\) with the resulting orders. We will denote this neighborhood as an *edge-based neighborhood* and use the notation \(N_{\text{edge}}\) to refer to it. \(N_{\text{edge}}(MPT)\) will denote the set of all solutions that can be obtained by the above procedure. The size of the neighborhood produced by \(N_{\text{edge}}\) is \(\frac{1}{2} \cdot \sum_{i=1}^{n} \sum_{j=1, j \neq i}^{n} |\Gamma_{ij}|\) (the division by two is due to counting every edge twice).

The main drawback of \(N_{\text{edge}}\) is that it can only attempt pairwise sharing. In many real-life scenarios, a single subexpression might be shared between patterns comprising a large fraction of the workload. While sharing such subexpression between all involved patterns may dramatically increase the performance, only considering two of them may fail to produce an improvement over the plan not sharing the expression at all. As a result, the sharing opportunity may be missed.

To overcome this limitation, we define *vertex-based neighborhood* \(N_{\text{vertex}}\) as follows. Let \(V_i = \bigcup_{(v_i, v_j) \in E} \Gamma_{ij}\) be called the *vicinity* of \(v_i\). Instead of an edge, the neighborhood function will select a vertex \(v_i\) and a subpattern \(P\) in the vicinity of \(v_i\). Then, let \(\Gamma_P\) denote a set of all patterns containing \(P\). This set can be efficiently retrieved from the MPG as described above. We will select \(\min(k, |\Gamma_P|)\) patterns, where \(k \geq 2\) is a predefined parameter. Then, \(A\) will be invoked to generate new evaluation orders sharing a common prefix \(O_P\). We will denote the variation of \(N_{\text{vertex}}\) using a particular value for \(k\) as \(N_{\text{vertex}}^k\). Note that \(N_{\text{vertex}}^2\) is equivalent to \(N_{\text{edge}}\). The size of the neighborhood of \(N_{\text{vertex}}^k\) is bounded by \(\sum_{i=1}^{n} \sum_{P \in V_i} \left( \frac{|\Gamma_P|}{k} \right)\).

The per-step complexity of the neighborhood functions \(N_{\text{edge}}\) and \(N_{\text{vertex}}^k\) is given by \(O(\sum_{i=1}^{n} m_i \cdot O)\), where \(O\) is the complexity of \(A\). A step is defined as a single selection of a neighbor and evaluating its cost. The average cost of a step can be decreased by introducing differential calculation of \(Cost_{\text{ord}}(MST)\). Instead of summing up the costs of all states in a tree, only states used by the patterns whose local plans were modified as a result of the step will be recalculated.

In all algorithms, the initial state is set to the MPT in which all patterns are evaluated according to the best possible individual orders, that is, \(P_i = A(P_i, Stat)\) for all \(i\).

We experimentally evaluate all combinations of meta-heuristics and neighborhood functions in Section 5.7.
5.5 MCEP with Arbitrary Subexpression Sharing

The multi-pattern plan generation method in Section 5.3 only considers prefix sharing. This introduces a significant limitation, since the optimizer is required to move common subpatterns to the MPT root in order to share their computation. This mechanism also prevents a pattern from sharing multiple distinct subexpressions with other patterns. A complete solution has to consider an option for sharing arbitrary subsets in a more flexible way. As an example, consider a workload consisting of patterns \( P_1 : SEQ(A, B, C, D) \), \( P_2 : SEQ(A, E, C, F) \), and \( P_3 : SEQ(G, B, H, D) \). In order to share the subpattern \( SEQ(A, C) \) with \( P_2 \), the evaluation order of \( P_1 \) must start with \((A, C)\) or \((C, A)\). On the other hand, it has to start with \((B, D)\) or \((D, B)\) to share the subpattern \( SEQ(B, D) \) with \( P_3 \). The optimizer will have to refrain from sharing one of the subpatterns in this case.

In this section, we extend our optimization framework to arbitrary subexpression sharing. To that end, we replace the local order-based plans with tree-based plans, shaped as binary trees. Tree-based plans \([MM09]\), extensively discussed in Chapters 3 and 4, specify the structure for tree-based single-pattern evaluation mechanisms. A leaf is defined for each event type, and the root of the tree serves as a final state. The evaluation proceeds from the leaves towards the root, with each internal node responsible for a subpattern consisting of the event types in its subtree. Figure 5.8 presents three possible tree-based plans for a pattern \( SEQ(A, B, C) \).

The tree-based evaluation process is similar to the one described for NFAs. As a new event arrives, an instance is created containing this event. Every instance corresponds to some subtree \( s \) of the tree-based plan. A new instance \( I \) is combined with previously created “siblings”, that is, instances associated with a node sharing the parent with the node of \( I \). As a result, another instance containing the unified subtree is generated. This process continues iteratively until the root of the tree is reached or no siblings are found.

Replacing order-based plans with tree-based plans allows us to share arbitrary subsets of patterns while preserving the characteristics of the order-based method. To the best of our knowledge, no previous work considers multi-pattern complex event processing based on tree-based plans.
Similarly to MPT, we define a *multi-pattern multitree* (MPM) as the global plan consisting of multiple shared tree-based plans. Each pattern in a MPM has a dedicated root, and all leaves corresponding to the same event type are shared regardless of the plan in use. Figure 5.9 depicts a possible MPM for the example above. Note that the displayed plan successfully shares both subpatterns of $P_1$ with $P_2$ and $P_3$, a result that could not be achieved using an order-based approach.

We will now formally define the cost function and the optimization problem of multitree-based multi-pattern complex event processing. We start with extending the cost function. Let $T_i$ denote a local tree-based evaluation plan for a pattern $P_i$. We use the cost function definition for tree-based plans from Section 3.4.2. For a plan $T_i$, we define:

$$\text{Cost}_{\text{tree}}(T) = \sum_{N \in \text{nodes}(T)} C(N),$$

where

$$C(N) = \begin{cases} 
W_i \cdot r_j & \text{if } N \text{ is a leaf representing } E_j \\
C(L) \cdot C(R) \cdot \text{sel}_{L,R} & \text{if } N \text{ is an internal node with child nodes } L \text{ and } R.
\end{cases}$$

Here, $\text{sel}_{L,R}$ denotes the total selectivity of all conditions defined between the event types in $L$ and $R$.

The extension of $\text{Cost}_{\text{tree}}$ for multitrees will be defined by counting the individual costs of all nodes in a multitree:

$$\text{Cost}_{\text{multi tree}}(\text{MPM}) = \sum_{N \in \text{nodes}(\text{MPM})} C(N).$$

Given a tree-based plan $T$ and a multi-pattern multitree $\text{MPM}$, we will say that $T \in \text{MPM}$ if and only if $\text{MPM}$ contains a subtree identical to $T$. We will denote a subtree of the MPM corresponding to a pattern $p_i$ as $T_i$. In addition, we will denote by $\text{TREE}_P$ the set of all tree-based plans of a pattern $P$. The extended optimization problem will be subsequently defined as follows.
Multitree-based MCEP optimization problem (M-MCEP). Given a workload $WL$ of $n$ patterns and a statistics collection $Stat$, find a multi-pattern multitree $MPM$ minimizing the value of the cost function $Cost_{multi}^{tree}(MPM, WL, Stat)$ subject to

$$\forall P_i, 1 \leq i \leq n : \exists T \in TREE_P s.t. T \in MPM.$$  

Since T-MCEP can be viewed as a particular case of M-MCEP (restricted to left-deep trees as local plans), the complexity results obtained for T-MCEP in Section 5.3.4 hold for M-MCEP by generalization.

To justify the use of $N_{edge}$ and $N_{vertex}^k$ for MPM-based solution space, we utilize an observation similar to the one presented in Theorem 5.1.

**Theorem 5.2.** Let $MPM_{opt}$ be the optimal multi-pattern multitree for some workload $W$. For each tree $T_i$ in $MPM_{opt}$ corresponding to the pattern $P_i$, let $S_i$ denote the set of subtrees that are shared with other patterns in $MPM_{opt}$. Then, $T_i$ is the most efficient local tree-based plan for $P_i$ out of those containing all the subtrees in $S_i$.

The idea of the proof is identical to the one employed for Theorem 5.1.

The MPM is created and modified similarly to the MPT. The complexity of the operations is not altered by switching to tree-based plans, as the number of nodes in a local tree-based plan is still linear in the number of the participating event types. In addition, the existence of a subtree $T$ in a MPM can be tested in constant time (and an additional $O(\sum_{i=1}^{n} m_i)$ space) by hashing the subtrees upon creation. The complexity analysis of runtime evaluation from Section 5.3.3 also remains unchanged for the multitree model, with the exception of the cost function $Cost_{ord}^{multi}$ being replaced with $Cost_{tree}^{multi}$. The structure of the multi-pattern graph and the operations performed on it do not depend on the plan structure and thus remain unchanged as well.

The local search process for MPMs functions as described for MPTs in Section 5.4.2. However, one important modification to the algorithms is necessary. In contrast to the order-based case, now it is possible for a pattern to share multiple disjoint subtrees. Consider a situation where one such subpattern $\hat{P}_1$ is already shared, and the optimizer attempts to share the second subpattern $\hat{P}_2$ during the local search step. In this case, we would like to consider two separate options: 1) the most efficient tree containing $\hat{P}_2$ regardless of the existing sharing of $\hat{P}_1$; 2) the most efficient tree containing both $\hat{P}_1$ and $\hat{P}_2$. This case can be generalized to sharing $q$ subtrees and considering the $(q + 1)^{th}$ one. Due to this extension, $A$ is required to support multiple subtrees. More formally, we require $A$ to be capable of the following:

1. Given a pattern $P$ and the statistical event characteristics $Stat$, return the cheapest local tree-based evaluation plan $T$ subject to $Cost_{tree}$.

2. Given a pattern $P$, a set of tree-based plans $\Upsilon$ for some subpatterns of $P$, and the statistics collection $Stat$, return the cheapest (subject to $Cost_{tree}$) local tree-based evaluation plan $T$ containing all trees in $\Upsilon$ as subtrees.
All algorithms for tree-based plan generation discussed in Chapter 3 satisfy the above requirements.

When selecting the next state to be returned, our neighborhood functions will randomly choose whether existing shared subtrees should be preserved for the patterns involved. For $N^k_{\text{vertex}}$, this decision will be performed independently for each of the $k$ patterns sharing a common subpattern. To apply this modification, we need to explicitly store sharing information for each pattern in the workload, which adds an $O(n \cdot \max \{|E_i|\})$ factor to the memory consumed by the MPG.

5.6 Advanced Features of a MCEP System

In previous sections, we justified our decision to adopt the local search paradigm by its applicability to systems with time-critical constraints. However, this is not the only advantage of this approach. In this section, we demonstrate how advanced system capabilities can be easily implemented on top of our MCEP optimization model.

Multiple challenges have to be solved by a practical CEP engine in addition to providing an efficient plan generation procedure. First, in many cases some patterns may be more important to efficiently detect than others. Most modern service providers allow a client to assign her pattern a high priority for an additional payment. In this situation, a service level agreement (SLA) is created between the customer and the service provider, with the latter guaranteeing a predefined level of performance for patterns belonging to the client, even when this decision may impact the performance of the total workload. This is an especially important consideration in cloud-based services [AJG+15], whose fusion with CEP-based environments is a hot area of research [BdDPP16].

Examine Figure 5.9 again. Note that the local tree-based plans for each of the patterns are not necessarily optimal. Figure 5.10(a) depicts a scenario in which the owner of $P_1$ enforces a SLA requiring the pattern to be detected according to the best possible local plan. Let this optimal plan be the one combining $A$ with $B$ and $C$ with $D$, then joining the intermediate results. In this case, enforcing the MPM to follow the optimal local plan for $P_1$ makes it impossible to utilize the opportunities of sharing $(A,C)$ with $P_2$ and $(B,D)$ with $P_3$, resulting in a suboptimal global plan.

Another practical consideration is the instability of the event stream statistical characteristics on which the plan generation is based. As we discussed in Chapter 4, in real-life scenarios the knowledge regarding these properties is rarely obtained in advance and is subject to rapid fluctuations over time. To overcome this problem, a CEP system must continuously estimate the up-to-date statistics and, if and whenever necessary, adapt itself to the changes.

Figure 5.10(b) shows a MPM resulting from an adaptive modification of the MPM from Figure 5.9 as a reaction to the drastic on-the-fly growth in the arrival rate of events of type $A$. As a result, it is no longer efficient to pair $A$ with $C$ due to the large
number of the resulting pairs. Instead, the most potent strategy is to postpone the processing of \( A \)'s to the last stage of detection, when the rest of the match is formed. The depicted MPM is optimal for the currently observed statistics, despite the reduced sharing degree between \( P_1 \) and \( P_2 \).

Finally, dynamic addition and removal of patterns in the workload is necessary in any long-running system. In a user-oriented environment, a customer may order or cancel the detection of her own patterns at any given moment. Consider the situation in Figure 5.10(c). As a fourth pattern \( P_4 : SEQ (A, B, C, X) \) is included in the workload depicted in Figure 5.9, the system is forced to reactivate the plan generation algorithm to improve the existing plan. As a result, local plans of other patterns may change. In our example, it is beneficial to share the subset \( (A, B, C) \) between \( P_1 \) and \( P_4 \), canceling previous sharing with \( P_2 \). While this functionality is commonly implemented in massive stream-based query processing systems [CCD+03, CDTW00], no similar efforts were
made for complex event processing systems.

5.6.1 SLA Requirements

Until now, our work focused on maximizing the performance of a global plan for the entire workload \( \{P_1, \cdots, P_n\} \) subject to the cost functions \( Cost_{ord} \) and \( Cost_{tree} \). We will now extend this approach to take service level agreements (SLAs) into account by also optimizing the local plans belonging to specific patterns subject to SLA-defined functions. One advantage of the proposed local search approach is the ease of incorporating SLA enforcement into the plan generation process. This can be accomplished the following simple modification.

Let \( \Omega_i \) denote the set of all valid local plans for a pattern \( P_i \). Let \( SLA_i : \Omega_i \rightarrow \{0, 1\} \) be an arbitrary Boolean function accepting a candidate local plan \( T_i \) for \( P_i \) and returning 1 if the plan adheres to the SLA defined for this pattern and 0 otherwise. One typical implementation of such a function might check whether the input plan is equal to \( A(P_i) \). This is acceptable for the most critical patterns, for which any plan yielding suboptimal performance is undesirable. A more relaxed version of the above might allow a predefined performance threshold \( \alpha < 1 \) and validate the condition \( \alpha \cdot Cost(T_i) \leq Cost(A(p_i)) \). For patterns with no SLA at all, \( SLA_i \) unconditionally returns 1. A customer may also wish the evaluation plan to satisfy any other criteria unrelated to our cost model, such as low detection latency.

The SLA verification function \( SLA_i \) will be invoked when an optimizer tries to alter the evaluation plan of \( P_i \) while creating a neighboring solution during a local search step. For a special case when only the optimal plan can be accepted, we can optimize the process by totally disallowing \( P_i \) from being selected for modification by the neighborhood function.

Increasingly complex customer-supplied functions might severely impact the per-step complexity of the local search, and hence the overall performance of the plan generation algorithm. Therefore, these expensive SLA functions should be avoided whenever possible. All common cases outlined above only introduce an additional linear factor to the complexity of a single local search step.

5.6.2 Dynamic MCEP

Most existing MCEP approaches assume that the problem is entirely static and all variables are kept intact [AcT08, DGH+06, LRG+11, RLR16, ZVDH17]. These methods commonly assume that a once created evaluation plan will remain efficient for the duration of the execution. However, as we discussed in Chapter 4, this assumption does not hold in real-life scenarios. The statistical properties of the input stream, such as the event arrival rates and predicate selectivities, can change rapidly and arbitrarily during the run. Moreover, even the initial values are not always known in advance. In addition, new patterns can be added to the workload during processing and old ones
can be removed. An obvious solution is to relaunch the search algorithm when a change like this is detected. This would result in prohibitively high costs when the changes are frequent.

In this section, we will demonstrate a general approach for efficiently handling dynamic behavior in multi-pattern CEP systems. Then, Sections 5.6.3 and 5.6.4 will show how to apply this approach to support adaptive CEP (i.e., reacting to runtime changes in the statistics) and on-the-fly workload modification respectively.

Our method is based on an extension to the traditional local search. Given an instance of T-MCEP or M-MCEP, we define a subset of patterns \( CP \subseteq W \). We modify our neighborhood function by limiting it to moves involving patterns from \( CP \). For \( N_{\text{edge}} \), this means that at least one of the endpoints of a selected edge must be in \( CP \). For \( N_{\text{vertex}}^k \) the pattern associated with the selected vertex has to be contained in \( CP \). We will refer to \( CP \) as the colored pattern set and to this new method as colored pattern search.

We start the local search with some initial set \( CP_0 \) that will be defined shortly. The colored pattern set is modified during each local search step. Specifically, following \( i^{th} \) step, all patterns whose local evaluation plans were affected are added to \( CP_{i-1} \) to produce a new set \( CP_i \). For our neighborhood functions \( N_{\text{edge}} \) and \( N_{\text{vertex}}^k \), the newly included patterns are those selected for sharing a subexpression at the current step. At some point of execution \( CP_i \) might contain all patterns in \( W \). In this case, the colored pattern search terminates and we continue with the standard local search.

The colored pattern search method allows us to mark some patterns for which we suspect a better local evaluation plan to exist (by including them in \( CP_0 \)), and to avoid modifying other patterns for which the currently employed plan is supposed to remain optimal. As the evaluation plans for the selected patterns are modified, the connections between different patterns are altered. New subpatterns may become shared and already shared subpatterns may become unshareable. By gradually expanding \( CP \), the changes made to the global plan are propagated further.

### 5.6.3 Adaptive MCEP

The problem of adaptive complex event processing has been extensively studied for single-pattern setting (see Chapter 4 for details). Our work is the first, to the best of our knowledge, to address this crucial aspect of complex event processing in a multi-pattern environment.

The general scheme for adaptive query/pattern processing consists of four stages [DIR07]: estimating the up-to-date values of the monitored statistics, deciding whether a reoptimization is required, computing the new evaluation plan, and replacing the currently used one. The first, the second, and the fourth stage employed by our solution do not differ from the state-of-the-art approaches for single-pattern scenarios. We utilize the algorithm from [DGIM02] to estimate the event arrival rates and predicate
selectivities over sliding windows. To decide whether plan recalculation is necessary, our system monitors the changes in the statistic values and detects a relative deviation exceeding a predefined threshold [MM09]. When the new plan is constructed, it replaces the currently deployed plan on-the-fly without losing intermediate results using a method described in Section 4.2.2.

The colored pattern search strategy presented above is employed during the new plan generation stage, thus avoiding plan recomputation for patterns whose statistics did not undergo a significant change. Let $\widetilde{Stat} \subseteq Stat$ denote a set of statistics whose values have shifted by a significant margin (this margin is defined by the deciding mechanism). Before calculating the plan reflecting these changes, we will derive a pattern set $\widetilde{W} \subseteq W$ consisting of patterns potentially affected as a result of the changes in $\widetilde{Stat}$. This is done by locating the patterns containing the event types whose arrival rates have changed, or the conditions whose selectivities were altered. Then, we will apply the colored pattern search with the initial state set to the currently employed global plan and $CP_0 = \widetilde{W}$.

Figure 5.11 illustrates this adaptation procedure on the multi-pattern graph shown in Figure 5.7. In 5.11(a), plan recalculation is triggered following an on-the-fly change in the arrival rate of the event type $G$. The colored pattern method is applied with $CP_0 = \{P_2\}$. At this point, only neighbors affecting the evaluation plan for $P_2$ are considered. Assume that before the change occurred, the computation of the subset $(A,C)$ was shared between $P_1$, $P_2$ and $P_3$. Figure 5.11(b) illustrates the next step of the local search, where a candidate plan was elected that does not share this subset, opting for locally optimal plans for the above patterns instead. Here, $CP_1 = \{P_1, P_2, P_3\}$ and hence the neighborhood function is now limited to producing plans that modify local plans for patterns in this set. Further assume that the algorithm chooses to modify the local plan for $P_4$ to share the subset $(D,A)$ with $P_3$, and that plan is globally optimal. Figure 5.11(c) presents the final state of the search, where all modified patterns have been colored and the cheapest plan has been found.
5.6.4 On-the-fly Addition and Removal of Patterns

Many real-life scenarios are characterized by online behavior; that is, a user can add a new pattern to the workload $W$ or remove a pattern on-the-fly at any moment. These operations require the system to apply changes to the global evaluation plan. In addition, the multi-pattern graph must be updated to reflect the modifications. Our goal is to minimize the time and the computational resources required for every such operation, making systems supporting this functionality practical in real-time setups.

We implement both operations using the colored pattern search. For insertion, an initial state for the local search is created by calculating the best possible evaluation tree $\hat{T} = \mathcal{A}(\hat{P})$ and adding it to the MPM. $CP_0$ is initialized to only contain $\hat{P}$. Then, colored pattern search is applied. Additionally, a new node corresponding to $\hat{P}$ is added to the MPG and edges are initialized with the information regarding its maximum common subpatterns with other patterns, as we described in Section 5.4.1.

For removal, the reverse action is done. Let $P_i$ be the removed pattern. First, we remove $T_i$ from the MPM, and also delete the node $v_i$ and all its edges from the MPG. The resulting plan will be our initial solution for the local search. We initialize $CP_0$ to contain all patterns that shared non-leaf states (that is, common subpatterns of size greater than 1) with $P_i$. The local plans for these patterns might now become inefficient, since the overall gain from sharing has decreased as a result of removal of $P_i$.

The complexity of an addition or a removal has three components: modifying the MPM to create an initial solution, modifying the MPG, and executing the local search. The complexity of the first stage is $O(m_P)$ for insertion and removal, where $m_P$ is the size of the pattern being added or removed. The second stage is executed at $O(n \cdot (s_{\text{max}} + c_{\text{max}}))$ in the case of insertion, where $s_{\text{max}}$ and $c_{\text{max}}$ denote the maximum pattern sizes in terms of events and conditions, respectively. For removal, MPG modification requires $O(n)$ time. Overall, the most dominant part is the execution of the local search, whose complexity was analyzed in Section 5.4.2. We attempt to optimize the average case using the colored pattern search strategy.

5.7 Experimental Evaluation

In this section, we report the results of our experimental evaluation. We assess the overall system performance achieved by our approach as compared to the state-of-the-art methods for MCEP, and analyze the impact of the various parameters on the quality of the generated global plans.

5.7.1 Experimental Setup

Two datasets from the previous empirical studies were used in the experiments, both described in detail in Chapters 2-4. The first was taken from the NASDAQ stock market historical records [EOD]. Each data record represents a single update to the price of a
stock, spanning a 1-year period. The second dataset contains the vehicle traffic sensor data, provided by the city of Aarhus, Denmark [AGM15] and collected over a period of 4 months. Each event represents an observation of traffic at the given point.

The structure of the patterns in the workloads generated for the stock dataset was motivated by the problem of monitoring the relative changes in stock prices. Each pattern represented either a sequence or a conjunction of a number of event types and included a number of predicates, roughly equal to half the pattern size, comparing the difference attributes of two of the involved event types. In addition, about 20% of the patterns contained either a negation or a Kleene closure operator on some event type. As discussed in Section 5.2, the aforementioned combinations of pattern operators are sufficient to cover the whole spectrum of pattern structures. For example, a typical sequence pattern of size 3 is as follows: $P_1 : SEQ(MSFT, Kleene(GOOG), APPL); C_1 = \{MSFT.diff < APPL.diff\}$.

The patterns created for the traffic dataset followed the rules specified above and were motivated by normal driving behavior, where the average speed tends to decrease with the increase in the number of vehicles on the road. We requested to detect the violations of this model, that is, combinations of three or more observations with either an increase or a decline in both the number of vehicles and the average speed.

Unless stated otherwise, all arrival rates and predicate selectivities were calculated in advance during the preprocessing stage. The measured arrival rates varied between 2 and 47 events per second, and the selectivities ranged from 0.003 to 0.92.

The workloads were created by grouping the patterns generated as described above based on a set of parameters, including the number of patterns in a workload, average pattern size (number of event types in a pattern), and pattern time window. Unless stated otherwise, the default values used for workload generation were set to 100 patterns per workload, an average pattern size of 5 event types, and the time window of 15 minutes. Two additional parameters of interest, multi-pattern graph density and normalized arrival rate difference, will be discussed later.

Unless stated otherwise, all experiments were conducted on the full version of our MCEP optimizer presented in Section 5.5. The default local search time limit for all algorithms was set to 180 seconds. We used the exhaustive algorithm based on dynamic programming described in Section 3.7.1 (DP-LD) as our local plan generation algorithm $A$.

We selected throughput, defined as the number of events processed per second during pattern detection, as our main performance metric. We believe that similar results could be obtained for algorithms targeting any other optimization goal, such as minimizing latency, power consumption, or communication cost.

All experiments presented below were repeated on 10 generated workloads, and the displayed results are averaged among all trials. All models and algorithms were implemented in Java. The experiments were run on a machine with 2.20 Ghz CPU and 16.0 GB RAM.
Figure 5.12: Throughput gain as a function of the workload size for different configurations: (a) stocks dataset, simulated annealing; (b) stocks dataset, Tabu search; (c) traffic dataset, simulated annealing; (d) traffic dataset, Tabu search.

Figure 5.13: Throughput gain as a function of the workload size for different configurations (prefix-only version): (a) stocks dataset, simulated annealing; (b) stocks dataset, Tabu search; (c) traffic dataset, simulated annealing; (d) traffic dataset, Tabu search.
5.7.2 Experimental Results

Impact of Input Parameters on System Performance

In our first experiment, we evaluated the performance of the local search algorithms described in Section 5.4 as a function of the workload size (Figure 5.12). Here and in all subsequent experiments, the graphs show the relative throughput gain over the trivial global evaluation plan, utilizing no sharing and no rewriting techniques. The neighborhoods $N_{\text{edge}}$, $N_{\text{vertex}}^4$, and $N_{\text{vertex}}^8$ were tested in conjunction with simulated annealing and Tabu search meta-heuristics on stock (Figures 5.12(a)-5.12(b)) and traffic (Figures 5.12(c)-5.12(d)) datasets.

Overall, all combinations demonstrated more significant performance gains for larger workloads, ranging from a factor of 21 to over 72. Despite being the simplest, $N_{\text{edge}}$ neighborhood showed the best results, finding evaluation plans that outperformed the trivial plan by a factor of up to 72.7 for the stock dataset and up to 50.7 for the traffic dataset. This can be explained by the overwhelming size of the neighbor spaces explored by $N_{\text{vertex}}^4$ and $N_{\text{vertex}}^8$. Tight time constraints prevent the system from locating the best optimization opportunities in huge neighborhoods. Thus, although $N_{\text{vertex}}$ neighborhoods contain all of the moves in $N_{\text{edge}}$, the better moves are statistically harder to reach before the time expires. Comparable performance was observed for both meta-heuristics, with simulated annealing slightly outperforming Tabu search for the stock dataset and vice versa for the traffic dataset.

Next, we repeated the above experiment for the prefix-only version of our framework presented in Section 5.3, as opposed to the default arbitrary-subset version considered for the rest of this section. The goal of this study was to determine the impact of a subexpression sharing strategy on the system performance.

Figure 5.13 illustrates the results. While the plots exhibit the same trends as those depicted in Figure 5.12, comparing the figures reveals the choice of a sharing strategy as a major factor in optimizing a MCEP engine. When the optimizer was restricted to only consider sharing prefixes, the generated evaluation plans resulted in up to 5 times lower throughput as compared to the plans produced using an identical setup without the above limitation. This observation fully matches our prior analysis. As we discussed in Section 5.5, a prefix-only approach ignores a significant fraction of the space of possible optimizations and limits a pattern to only sharing a single subexpression by utilizing order-based local plans as opposed to tree-based ones.

We further assessed the scalability of our optimizer subject to various parameters. Figures 5.14 and 5.15 summarize the results for the stock and the traffic datasets, respectively. Simulated annealing (marked as ‘SA’ in the graph) and Tabu search (marked as ‘TS’) were again evaluated in conjunction with $N_{\text{edge}}$ and $N_{\text{vertex}}^4$. Figures 5.14(a) and 5.15(a) depict the throughput gain as a function of the average length of a pattern in a workload. Our approach seems to improve even more for longer patterns, speeding up the event processing by up to two orders of magnitude. This is not
Figure 5.14: Throughput gain of the local search algorithms applied on the stock dataset as a function of: (a) average pattern size; (b) local search running time; (c) pattern timestamp-based window; (d) pattern count-based window.

Figure 5.15: Throughput gain of the local search algorithms applied on the traffic dataset as a function of: (a) average pattern size; (b) local search running time; (c) pattern timestamp-based window; (d) pattern count-based window.
surprising, as longer patterns introduce more opportunities for sharing and reordering. The combination of simulated annealing and $N_{\text{edge}}$ (marked as ‘SA-EDGE’) was the most effective, closely followed by ‘TS-EDGE’.

Unsurprisingly, the output plan quality also improves with increased time limit of the local search algorithm (Figures 5.14(b) and 5.15(b)). Interestingly, the performance of simulated annealing seems to converge to a constant value, while Tabu search keeps improving for longer time limits. This can be explained by the distinctive behavior of the former after a large number of iterations, when the current threshold becomes small enough for the algorithm to converge to a local minimum.

The results obtained for different time window sizes (Figures 5.14(c) and 5.15(c)) demonstrate similar trends. Since our cost function and the overall system throughput strictly depend on the value of this parameter, increasing it leads to bigger differences in plan qualities, both calculated and empirically observed.

Finally, we experimented with patterns utilizing count-based windows. As opposed to specifications based on time-based windows that we defined in Section 5.2, count-based patterns require a match to appear within the last $W$ arrived events rather than within $W$ time units.

For the stock dataset, even bigger performance boost was observed for larger windows (Figure 5.14(d)) as compared to the time-based scenario. This can be explained by the highly fluctuating event arrival rates exhibited by this dataset. When time-based windows are used, the peak load is only experienced during brief ‘bursts’, whereas large count-based windows cause the system to be constantly overloaded. Since the performance gain achieved by an efficient evaluation plan is proportional to the average system load, the latter case demonstrates a more significant increase in total throughput. In contrast, the results for the traffic dataset (Figure 5.15(d)) were extremely similar to those obtained for time-based windows due to much less skew in event distribution over the input stream.

Our next experiment explores the influence of the workload statistical characteristics on the performance of our optimizer. From now on, we only evaluate the best performing combinations SA-EDGE and TS-EDGE.

We control the statistical characteristics of workload generation using a pair of configurable parameters, multi-pattern graph density and normalized arrival rate difference. The multi-pattern graph density is defined as an average relative number of neighbors of a given pattern in a MPG. For example, in a workload of 100 patterns with MPG density equal to 0.5, each pattern will have 50 neighbors on average. This parameter is used to control the sharing sensitivity of a workload.

The arrival rate difference, defined as the maximal difference in rates of two event types within a single pattern, allows us to manipulate the reordering sensitivity of a workload. For example, for an unconditional conjunction of 5 event types arriving at an identical rate, each of the possible 5! evaluation orders will have the same cost and will yield the same throughput. However, if one of the types appears 100 times more
Figure 5.16: Throughput gain of the local search algorithms as a function of workload statistical properties: (a) stock dataset, sharing sensitivity; (b) traffic dataset, sharing sensitivity; (c) stock dataset, reordering sensitivity; (d) traffic dataset, reordering sensitivity.

frequently than the rest, the gain obtained by postponing the costly event type to the last state is considerably high. Patterns with varying degrees of reordering sensitivity are produced by limiting the selection of the event types for a pattern accordingly. The values of this parameter were normalized with respect to the maximal observed difference of 45.

Figure 5.16 depicts the achieved throughput gain as a function of the sharing sensitivity (Figures 5.16(a) and 5.16(b)) and the reordering sensitivity (Figures 5.16(c) and 5.16(d)) of the workload. The plots also show the performance of the basic reordering (RE) and the basic sharing (SH) methods. The basic reordering method utilizes the greedy cost-based algorithm from Section 3.7.1, building the event sequence by picking the event type maximizing the cost function at each step. The basic sharing method refers to the maximal subexpression sharing technique mentioned in Section 5.3.2 and used in many previous studies (e.g., [AcT08, DGH+06, DAF+03, HRK+09]).

The high gains of the local search methods do not exhibit dominant dependencies on either of the two parameters. While larger graph densities and rate difference limits introduce more sharing and reordering opportunities, they also increase the search space size and the number of potential local minima. Nevertheless, our approach consistently outperforms the better of SH and RE for every attempted experimental configuration. At the extremes, local search tends to resort to an almost pure sharing plan for low
arrival rate differences (since virtually no improvement can be achieved by reordering), whereas for sparse multi-pattern graphs the solution assigning the best local plan to all patterns is often preferred.

The basic reordering method becomes more efficient with increasing differences in arrival rate and is almost unaffected by the changes in graph density (barring the insignificant noise factor). The performance of the basic sharing method increases monotonically with graph density. It also decreases with the rate difference due to the smaller number of participating event types in more restricted workloads. Given a pair of workloads of the same size containing patterns of the same length, the workload with fewer event types will have more events of the same type on average, and is expected to offer more sharing opportunities.

State-of-the-art Comparison

We repeated the experiments summarized in Figures 5.12, 5.14 and 5.15 for the basic SH and RE methods discussed above, as well as for two recent state-of-the-art MCEP methods [RLR16, ZVDH17]. SPASS [RLR16] selects the subpatterns to share according to a metric called ‘redundancy ratio’. This metric represents the potential gain in sharing its computation. Each subexpression is assigned a score, and the winners are chosen by approximating the well-known minimal substring cover problem. MOTTO [ZVDH17] utilizes a combination of techniques referred to as MST (merge sharing technique), DST (decomposition sharing technique), and OTT (operator transformation technique). The system solves the directed Steiner minimum tree problem to select the best global plan produced by applying the above techniques.

Figures 5.17 and 5.18 present the results. The redundancy ratio method and the merge-decomposition technique are marked as SH-RR and SH-MDT respectively. While both SH-RR and SH-MDT scale well with growing workload size (Figures 5.17(a) and 5.18(a)) and average pattern length (Figures 5.17(b) and 5.18(b)), our optimizer achieves the best overall speedup, in some cases up to three times better than that of the runner-up solution. This result follows from utilizing the reordering opportunities, which were shown in previous chapters to boost CEP evaluation by up to several orders of magnitude. On the other hand, our approach also attempts to exploit sharing opportunities when possible, which allows it to outperform the pure reordering algorithm (RE) for large pattern sizes. The gaps were closer for time window evaluation (Figures 5.17(c) and 5.18(c)), with SA-EDGE still achieving an advantage of at least 25% over the second-best method. The results for count-based windows (Figures 5.17(d) and 5.18(d)) strictly follow the trends described for Figures 5.14 and 5.15.

In all experiments, SH-MDT performed slightly better than SH-RR due to traversing a larger solution space by considering non-continuous subexpression sharing (as exemplified in Figure 5.3(d)).
Figure 5.17: Throughput gain of the state-of-the-art and the local search algorithms applied on the stock dataset as a function of: (a) workload size; (b) average pattern size; (c) pattern time window; (d) pattern count-based window.

Figure 5.18: Throughput gain of the state-of-the-art and the local search algorithms applied on the traffic dataset as a function of: (a) workload size; (b) average pattern size; (c) pattern time window; (d) pattern count-based window.
Figure 5.19: Throughput gain as a function of the percentage of the patterns with SLAs requiring optimal local plans.

**Advanced System Features**

We extensively tested the advanced system features described in Section 5.6. The experiments presented below were only conducted on the stock dataset.

Figure 5.19 presents the results of evaluating pattern workloads with SLA constraints. In this experiment, some of the patterns were associated with SLAs disabling suboptimal local plans (Section 5.6.1). The graph depicts the relative throughput gain as a function of the percentage of such patterns in the workload. As expected, the global performance drops drastically as more patterns with SLA are introduced. When the ratio of such patterns exceeds 40%, the optimizer almost exclusively remains in the initial state when the search is completed. As we described in Section 5.4.2, the initial scheme assigns the locally optimal plan to each participating pattern.

In our next experiment, we evaluated the performance of our system in the presence of a dynamically changing input stream (Section 5.6.3). For this experiment alone, semi-synthetic input was used. We implemented a component that accepts a parameter $x$ and randomly and independently transforms every $x$ incoming events before they are received by the evaluation mechanism. A transformation is performed by picking $y$ event types at random, creating their random permutation $P$ and then replacing the type attribute of every affected event with the one following its value in $P$. This modification allows us to simulate rapid and drastic changes in the arrival rates of all types of events.

We repeated the experiment for $y = 5$ and $x$ ranging between 10 and 1000 on different variations of our framework: 1) a static system generating an evaluation plan on startup and using it exclusively regardless of input changes (marked as SA-STATIC); 2) an adaptive platform, restarting the initial plan calculation process when a drastic change in the statistics is detected (SA-DYNAMIC); 3) same as (2), but with the basic local search algorithm replaced with the colored search method from Sections 5.6.2-5.6.3 (SA-CSM).

The results are depicted in Figure 5.20. Unsurprisingly, the initially generated plan fails to perform adequately when the input characteristics overcome on-the-fly changes.
The colored search method outperforms the basic adaptive strategy by 19% to 41%. The biggest improvement is achieved when there is no more than a single modification per 100 events. When the transformations are extremely frequent, the high number of modified statistics results in almost all nodes being colored, essentially making the two adaptive methods nearly identical.

Finally, we experimentally assessed the scenario of dynamic workload modification (Section 5.6.4). Following the processing of every $x$ events by the evaluation mechanism, either a newly generated pattern was added to the workload, or an existing one removed at random. We compared the basic modification approach, restarting the local search procedure upon each such modification, with the colored search method. Figure 5.21 displays the throughput gain of the both strategies arranged by the number of workload modifications per 1000 processed events. The colored search method performs significantly better than in the previous experiment (up to 3.4 times better than the basic approach). This is not surprising, as the invocation of the basic method often leads to a complete change of plan for all patterns, when only slight local modifications could have been made to reflect an addition or a removal of a single pattern.

Figure 5.20: Throughput gain as a function of the number of input stream event reshufflings per 1000 incoming events.

Figure 5.21: Throughput gain as a function of the number of dynamic additions and/or removals of patterns to/from the workload per 1000 incoming events.
Chapter 6

Related Work

6.1 Complex Event Processing Systems

Scalable systems for real-time complex event detection have been the focus of much research in recent years, as a result of the rising demand for technologies of this type [CM12b, FGD+17, DP18]. The earliest solutions designed for solving this problem fall under the category of Data Stream Management Systems. Those systems are based on SQL-like specification languages and focus on processing data coming from continuous, usually multiple input streams. Examples include TelegraphCQ [CCD+03], NiagaraCQ [CDTW00], Aurora/Borealis [ABB+05, BBC+04], Stream [ABB+16, Gro03], OpenCQ [LPT99], Gigascope [CGJ+02], and Stream Mill [BTW+06].

Later, the need to analyze event notifications of interesting situations – as opposed to generic data – was identified. Then, a broad variety of complex event processing systems were introduced. SASE/SASE+ [ADGI08, WDR06, ZDI14], Cayuga [BDG+07, DGH+06, DGP+07], Amit [AE04], CEDR [BGAH07], T-Rex [CM12a], DistCED [PSB03], NextCEP [SMMP09], PB-CED [AcT08] and SPADE [GAW+08] are some of the examples of advanced CEP systems.

Apart from SASE [WDR06] and SASE+ [ADGI08], thoroughly discussed above, many other event specification languages were proposed. CEL [DGH+06, DGP+07] is a declarative language used by the Cayuga system. It supports patterns with Kleene closure and event selection strategies. CQL [ABW06] made it possible to create transformation rules with a unified syntax for processing both information flows and stored relations. TESLA [CM10] combines high expressiveness with a relatively small set of operators. It offers completely programmable per-event selection policies. CEDR [BGAH07] presents a complex and flexible temporal model, based on three different timings. Our lazy evaluation framework is based on SASE+ [GADI08], an expressive language, including constructs such as iterations and aggregations. Nevertheless, the operators it supports serve as basic blocks for most of the described languages.

The majority of CEP approaches incorporate NFAs or other form of a finite state machine as their primary evaluation mechanism [ADGI08, AcT08, CGM10, CM12a,
DGH+06, PSB03, SMMP09, WDR06]. ZStream [MM09] utilizes tree-based detection plans for the representation of event patterns. Event processing networks [EN10] is another conceptual model, presenting a pattern as a network of simple agents.

Multiple open-source CEP libraries are available, such as Esper [Esp], Siddhi [SGN+11], and Cayuga [DGH+06]. Moreover, large-scale CEP engines are on the rise, examples being IBM System S [AAB+06], TIBCO [Bro13], WSO2 CEP [RD15], and CHAOS [GWA+09].

### 6.2 Single-Pattern Optimizations in CEP

CEP systems implement a broad variety of optimization techniques aimed at minimizing processing time and resource consumption of a single pattern [HSS+14]. In [REG11], a rewriting framework is described, based on unifying and splitting patterns. A method for efficient Kleene closure evaluation based on sharing with postponed operators is discussed in [ZDI14], while in [PLAR17] the authors solve the above problem by maintaining a compact graph encoding of event sequences and utilizing it for effective reuse. RunSAT [DCR+08] utilizes another approach, preprocessing a pattern and setting optimal points for termination of the detection process. SASE [WDR06], Cayuga [DGP+07] and T-Rex [CM12a] design efficient data structures to enable smart runtime memory management. These NFA-based mechanisms do not support out-of-order processing, and hence are still vulnerable to the problem of large intermediate results.

Multiple works have addressed the broad range of optimization opportunities for order-based CEP evaluation arising when the statistical characteristics of the primitive events are taken into account. In [AcT08] “plan-based evaluation” is described, where the arrival rates of events are exploited to reduce network communication costs for distributed CEP. The authors of NextCEP [SMMP09] propose a framework for pattern rewriting in which operator properties are utilized to assign a cost to every candidate evaluation plan. Then, a search algorithm (either greedy or dynamic) is applied to select the lowest cost detection scheme. Unlike our proposed framework, none of these works takes the selectivities of the event constraints into account.

ZStream [MM09], thoroughly discussed above, presents an optimization framework for optimal tree generation, based on a complex cost model. A set of algebraic rule-based transformations is applied on a given pattern, and then the operators are reordered to minimize the cost of a plan. As was shown in Section 3.2.3, since the leaves of an evaluation tree cannot be reordered, it only searches through a partial solution space.

### 6.3 Lazy Evaluation

The concept of lazy evaluation has been widely applied in multiple research fields. Related work was also conducted in the context of complex event processing. ZStream [MM09] uses tree-based query plans to represent complex patterns. As the most
efficient plan is chosen, the pattern detection order is dynamically adjusted by internal buffering. In [AcT08], the authors describe “plan-based evaluation”, where temporal properties of primitive events are exploited to reduce network communication costs. [CFGR02] presents an XPath-based mechanism for filtering XML documents in stream environments. This method postpones costly operations as long as possible. While many of the ideas discussed by the aforementioned studies are close to ours, none considers the nondeterministic finite automata environment, which is our primary innovation in Chapter 2.

Several authors propose postponing and preprocessing mechanisms for NFA-based evaluation frameworks. In [ZDI14], the authors present an optimization method based on sharing common operations between instances and postpone per-instance part. The focus of that technique is on improving the performance of Kleene closure patterns under the skip-till-any-match selection policy. [DM07] discusses a mechanism similar to ours, including the concept of storing incoming events and evaluating more selective events before more frequent ones. However, the authors do not consider complex pattern types, such as negation and iteration. [CGB12] proposes a matching strategy including a preprocessing step, which is similar to our input buffering. It is used for applying pruning techniques, such as filtering and partitioning, rather than for event reordering. APAM [YLW16] is a hybrid eager-lazy evaluation method. For each event type, it determines which method to use for minimizing detection latency.

The concept of lazy evaluation has also been proposed in the related research field of online processing of XML streams. [CFGR02] describes an XPath-based mechanism for filtering XML documents in stream environments. This mechanism postpones costly operations as long as possible. However, the goal in this setting is only to detect the presence or absence of a match, whereas our focus is on finding all possible matches between primitive events. In [GGM+04], a technique for lazy construction of a DFA (Deterministic Finite Automaton) on-the-fly is discussed. This work is motivated by the problem of exponential growth of automata for XPath pattern matching. Our work solves a different problem of minimizing the number of runtime NFA instances rather that the size of the automaton itself. In addition, while there is some overlap in the semantics of CEP and XPath queries, they were designed for different purposes and allow different types of patterns to be defined.

6.4 Join Query Plan Generation

Estimating an optimal evaluation plan for a join query has long been considered one of the most important problems in the area of query optimization [SMK97]. Multiple authors have shown the NP-completeness of this problem for arbitrary query graphs [CM95, IK84], and a wide range of methods were proposed to provide either exact or approximate close-to-optimal solutions [KS00, KBZ86, LSC97, MN06, SAC+79, SMK97, Swa89].
Methods for join query plan generation can be roughly divided into two main categories. The heuristic algorithms, as their name suggests, utilize some kind of heuristic function or approach to perform efficient search through the huge solution space. They are often applied in conjunction with combinatorial [IK90, SMK97, Swa89] or graph-based [KBZ86, LSC97] techniques. The heuristic algorithms produce fast solutions, but the resulting execution plans are often far from the optimum.

The second category of JQPG algorithms, the exhaustive search algorithms, provide provable guarantees on the optimality of the returned solutions. These methods are often based on dynamic programming [MN06, SAC79] and thus suffer from worst-case exponential complexity. To solve this issue, hybrid techniques were proposed, making it possible to set the trade-off between the speed of heuristic approaches and the precision of DP-based approaches [KS00].

Incorporating join optimization techniques from traditional DBMSs was already considered in the related fields, such as XPath [GRT07] and data stream processing [CDTW00]. For the best of our knowledge, our work in Chapter 3 is the first to address the CEP-specific challenges and to provide a formal reduction.

6.5 Adaptive CEP

Adaptive query processing (AQP) is the widely studied problem of adapting a query plan to the unstable data characteristics [DIR07]. Multiple solutions consider traditional data-bases [AVL11, IHW04, BBD05, MRS04, KD98, SLMK01]. The mid-query reoptimization mechanism [KD98], one of the first to possess adaptive properties, collects statistics at the predefined checkpoints and compares them to the past estimates. If severe deviation is observed, the remainder of the data is processed using a new plan. The methods described in [BBD05] and [MRS04] are the closest in spirit to our work. Rather than executing reoptimization on a periodic basis or upon a constant change, the authors compute an individual range for each monitored value within which the current plan is considered close-to-optimal.

The field of stream processing has developed adaptive techniques of its own. A-Greedy [BMM04] is an algorithm for adaptive ordering of pipelined filters, providing strong theoretical guarantees. Similarly to our method, it detects violations of invariants defined on the filter drop probabilities. The authors of [LIL16] describe “incremental reoptimization,” where the optimizer constantly attempts to locate a better plan using efficient search and pruning techniques. Eddy [AH00, BBDW05, MSHR02] presents stateless routing operators, redirecting incoming tuples to query operators according to a predefined routing policy. This system discovers execution routes on-the-fly in a per-tuple manner. Query Mesh [NWL13] is a middle-ground approach, maintaining a set of plans and using a classifier to select a plan for each data item. Large DSMSs have also incorporated adaptive mechanisms [TeZ03, BW04].

The majority of the proposed CEP techniques are deprived from adaptivity consider-
ations [FGD+17]. One notable exception, ZStream [MM09] was covered in detail above. Additional works labeled as ‘adaptive’ refer to on-the-fly switching between several detection algorithms [SJ14, YLW16] or dynamic rule mining [CMM12, LYHJ15].

To the best of our knowledge, no CEP-based method was proposed for adaptive multi-pattern event detection.

6.6 Multi-Pattern CEP

Multi-query optimization (MQO) is a well-known problem in database query processing [Sel88]. Various methods have been proposed for sharing common subexpressions between queries in a workload. Notable examples include Volcano [RSSB00], MQJoin [MGAK16], and Monet [MPK00]. Multi-query sharing techniques were incorporated in large-scale engines, such as SPARQL [LKDL12] and Microsoft SQL Server [ZLFL07]. SWO [GMAK14] is the closest in spirit to our work. Rather than scanning the input queries for common subexpressions, the global optimal execution plan is calculated using an optimization algorithm that employs the branch-and-bound method.

Query sharing mechanisms were also developed for stream processing [AW04, DGGR16, HFAE03, HRK+09, JRSA16]. NiagaraCQ [CDTW00] is a large-scale system for processing multiple continuous queries over streams. To the best of our knowledge, this is the only stream processing system to consider a multi-query setting with dynamic workload modification. TelegraphCQ [CCD+03] is remarkable for introducing CACQ [MSHR02], a technique based on per-tuple dynamic routing (also known as Eddies [AH00]) for sharing arbitrary subsets among queries. Similarly to our approach, this method combines adaptive sharing and query rewriting. Dedicated MQO solutions were also proposed for processing streams of XML documents [DAF+03, HDG+07].

Pattern sharing techniques for complex event processing are being actively researched. Numerous advanced methods have been proposed for intra-pattern (sharing of subexpressions inside a nested pattern) [LRD+11, RRL+13] and inter-pattern scenarios (sharing between different patterns in a MCEP system) [AcT08, DGH+06, LRG+11, RLR16, ZVDH17]. Some solutions consider aggregations over event streams [QCRR14].

To the best of our knowledge, PB-CED [AcT08] and MOTTO [ZVDH17] are the only CEP systems to consider a combination of sharing and pattern rewriting techniques. However, the solution provided by [AcT08] only considers a single shared subpattern. In [ZVDH17], arbitrary subset sharing is achieved by transforming a sequence pattern to a conjunction (rather than by reordering methods), which was shown in Section 2.4.2 to severely diminish the performance of NFA-based event detection.
Chapter 7

Conclusions and Future Work

This thesis presented a novel approach for overcoming the scalability limit of modern CEP engines caused by the worst-case exponential complexity of detecting arbitrarily complex patterns over event streams.

In Chapter 2, we discussed a lazy evaluation mechanism for efficient detection of complex event patterns. Unlike previous solutions, our proposed method does not process the events in order of their arrival, but rather according to their ascending order of frequency. Two lazy NFA topologies were introduced to implement the above concept. The Chain NFA requires the arrival rates of the events in the pattern to be known in advance. The Tree NFA utilizes an adaptive approach by computing the actual frequency order on-the-fly. Our experimental results showed that both Chain NFA and Tree NFA achieve significant improvement over the eager evaluation mechanism in terms of throughput, memory consumption, and runtime complexity.

Chapter 3 targeted the problem of generating efficient evaluation plans for plan-based evaluation mechanisms, such as the aforementioned lazy NFA. To that end, we studied the relationship between two important and relevant problems, CEP Plan Generation and Join Query Plan Generation. It was shown that the CPG problem is equivalent to JQPG for a subset of pattern types, and reducible to it for other types. We discussed how close-to-optimal solutions to CPG can be efficiently obtained by applying existing JQPG methods. CEP-related challenges, such as detection latency and event selection strategies, were addressed. The presented experimental study supported our analysis by demonstrating how the evaluation plans created by some of the well-known join algorithms outperform those produced by the methods traditionally used in CEP systems.

In Chapter 4, we discussed efficient adaptation of a CEP system to on-the-fly changes in the statistical properties of the data. A new method was presented to avoid redundant reoptimizations of the pattern evaluation plan by periodically verifying a small set of simple conditions defined on the monitored data characteristics, such as the arrival rates and the predicate selectivities. We proved that validating this set of conditions will only fail if a better evaluation plan is available. We applied our method on two
real-life algorithms for plan generation and experimentally demonstrated the achieved performance gain.

Finally, in Chapter 5 we studied the problem of optimizing multi-pattern CEP performance using a combination of sharing and reordering techniques. We formally defined the optimization problem of finding an optimal global evaluation plan for a pattern workload. We then presented an optimization framework for solving this computationally hard problem under tight real-time conditions. Extensions to the basic mechanism, such as SLAs, adaptive evaluation and dynamic workload modification, were discussed. Our experimental evaluation demonstrated a significant performance boost as compared to currently employed state-of-the-art techniques for MCEP.

Complex event processing is a constantly evolving and developing research area. Therefore, there is no shortage of related open problems. While an exhaustive list is beyond the scope of this work, we believe that the methods and approaches presented in this thesis could provide a strong starting point for addressing many of the current drawbacks and open research questions in the field. Some of the important examples include:

1. **Distributed complex event processing.** Simultaneous pattern detection at multiple, possibly geographically distant sites is a common scenario in large CEP frameworks. In this case, communication cost and latency become the most important concerns. Monitoring complex events in this environment is a challenging task and a hot research topic [AcT08, CGLPN16, CM09, Hir12, LRE09, LWZ15, PSB03, REG11, SMMP09]. It has been demonstrated by previous works that plan-based methods utilizing statistical properties of the data can achieve significant performance improvements in distributed setups [AcT08, SMMP09]. However, the main challenge in deploying these methods in real-life systems follows from the high degree of synchronization required to maintain global consistency, especially when evaluation plans are generated adaptively using dynamically estimated data characteristics.

2. **Complex event processing with uncertain data.** Many CEP application domains involve input streams which consist of incomplete, imprecise, or uncertain data. As an example, events may contain wrong timestamps or attribute values, or the probability of the occurrence of a certain event may be less than one. The research area of uncertain complex event processing is devoted to finding computationally efficient solutions for handling this type of streams. In addition, ‘pattern uncertainty’ refers to scenarios where the patterns to be detected are either incomplete or not fully known in advance. While many different approaches have recently been proposed [ASAP17, CMMT15, FGD+17, RLBS08, CSLJ10, WCZ13, ZDI10], we believe that lazy statistic-aware methods and related concepts, such as input stream buffering and data distribution estimation, introduce opportunities for novel evaluation mechanisms providing better support for complex event
processing under uncertainty.

3. **Complex event processing on multi-core platforms.** In the modern era of massive utilization of multi-core servers and GPUs for computationally hard problems, a practical CEP solution cannot be considered complete without addressing the aspect of parallelization. However, parallelizing NFA-based event detection is not trivial due to the need to efficiently share partial matches. Existing methods for parallel CEP [BDWT13, Hir12, MKR15] showcase limited scalability, as only data parallelism is exploited. Exploring a possibility of implementing a parallelized lazy evaluation mechanism based on a fusion of data-based and state-based parallelism is a promising research direction.
Appendix A

Equivalence of the Eager and the Lazy Chain NFA

In this appendix, we formally prove the correctness of the Chain NFA construction. It is shown that, for any pattern over the presented operators, a Chain NFA or a Lazy Multi-Chain NFA constructed according to Section 2.4 or 2.5 respectively is equivalent to the corresponding eager NFA in terms of the language it accepts.

Theorem A.1. Let \( P \) be a complex event pattern, as defined in Section 2.2.1, over the event types \( e_1, \ldots, e_n \). Let \( A_{\text{eager}} = (Q_{\text{eager}}, E_{\text{eager}}, q_{1\text{eager}}, F_{\text{eager}}, R_{\text{eager}}) \) be an eager NFA derived from \( P \) as described in Section 2.2.2 and let \( A_{\text{lazy}} = (Q_{\text{lazy}}, E_{\text{lazy}}, q_{1\text{lazy}}, F_{\text{lazy}}, R_{\text{lazy}}) \) be a Chain NFA or a Multi-Chain NFA derived from \( P \). Then \( A_{\text{eager}} \) and \( A_{\text{lazy}} \) are equivalent, i.e., \( A_{\text{eager}} \) accepts a stream of events if and only if \( A_{\text{lazy}} \) accepts it.

We will start with the outline of the proof. It will proceed in several steps.

First, we examine the case where \( P \) is a pure sequence pattern. We show that swapping two adjacent states in the NFA detecting \( P \) does not affect its correctness when the input buffer is used. From there, the statement is proven by induction for an arbitrary detection order. Next, the case of a sequence with negation is shown separately for both lazy mechanisms introduced above. In both parts, NFA construction properties are used to infer correctness. We prove the case of a sequence with iteration by induction on the length of an iterated pattern match, viewing the pattern as a regular sequence during the induction step.

Afterwards, we continue to more complex pattern types. We start by observing that conjunctions and partial sequences can be represented as disjunctions of full sequences. Then, we prove that two branch-structured NFAs (i.e., built of several chains starting at the initial state) are equivalent if their sub-automata are equivalent. Similarly, we demonstrate that a branch-structured NFA is equivalent to a chain-structured NFA, if all of the branches are equivalent to this chain-structured NFA. On the basis of these three statements, we derive the correctness for conjunctions and partial sequences as well. From there, the case of disjunctions is shown in a similar manner. We complete
the proof by generalizing this case for an arbitrary composite pattern, using DNF form conversion.

**Lemma A.0.1.** Let $P$ be a sequence pattern over $e_1, \cdots, e_n$, let $seq$ be the order of the events in $P$, and let $freq$ be an ascending frequency order. Let $A_{eager}$ be an eager NFA and let $A_{lazy}$ be a Chain NFA derived from $P$. Then $A_{eager}$ and $A_{lazy}$ are equivalent.

We first address the case where the order $freq$ is identical to $seq$, i.e., we show that a lazy NFA for $P$ where the frequency order is identical to the sequence order is equivalent to the eager NFA for $P$. Next we show that if $freq'$ is an order received by swapping the event types at location $i$ and $i+1$ ($1 \leq i \leq n-1$) in $freq$, then $A_{lazy}$ is equivalent to $A_{lazy'}$, which uses $freq'$ instead of $freq$. This will conclude the proof, since we can obtain $freq$ by a set of swaps of adjacent event types on $seq$.

For the case $seq = freq$, the ordering filters of any edge will point to the end of the input buffer, and all events will be taken from the input stream; hence, the conditions on edges will become the same as in the eager NFA. In addition, the order of states will be the same as in the eager NFA. Consequently, the transition function between states is the same, making the two automata identical.

Now assume that $freq'$ is received by swapping the event types at location $i$ and $i+1$. Let $A_{lazy'}$ be a corresponding Chain NFA derived from $P$ and $freq'$.

Note that $A_{lazy}$ and $A_{lazy'}$ differ only in the definitions of the outbound edges (including the self-loops) from the $i^{th}$ and $(i+1)^{th}$ states, in particular in the ordering filters on their take edges. Let $s_1, \ldots, s_m$ be a stream of events. We examine the processing of this stream by both $A_{lazy}$ and $A_{lazy'}$. We focus on instances of $A_{lazy}$ that have reached the state $q_i^{lazy}$ and $q_i^{lazy'}$. We show that for every such instance, there is a corresponding instance in $A_{lazy'}$ that has reached $q_i^{lazy'}$ with the same input and match buffers. It is sufficient to show one direction (that instances in $A_{lazy}$ have corresponding instances in $A_{lazy'}$), since the proof for the other direction is symmetric. Proving such a mapping will show that $A_{lazy}$ and $A_{lazy'}$ are equivalent, since the processing of these instances will be identical in both NFAs from this point on.

Let us examine an instance $I$ that has reached $q_i^{lazy}$ in $A_{lazy}$. We will denote the event that caused $I$ to transition into $q_i^{lazy}$ by $s_l$. In addition, let us denote the event that caused $I$ to transition into $q_i^{lazy'}$ by $s_k$. Let $t_k$ denote the arrival time of $s_k$. Since the definitions of $A_{lazy}$ and $A_{lazy'}$ are identical up to the $i^{th}$ state, at time $t_k$ there is an instance $I'$ of $A_{lazy'}$ with match and input buffers identical to those of $I$. Furthermore, $s_k$ will also cause $I'$ to transition into $q_i^{lazy'}$. From this point on, the processing of the two instances may diverge, but we show that after the event $s_l$ is processed, both instances will have transitioned into $q_{i+2}$ with the same input and match buffers.

Any event corresponding to an event type already located in the match buffer will be ignored by both $I$ and $I'$. Similarly, any event corresponding to an event type requested by one of the later states ($q_{i+2}$ and beyond) will be stored to respective input buffers.
of both instances. Thus, we are left to consider the events that trigger the transitions $q_i^{\text{laz}} \rightarrow q_i^{\text{laz}+1}$ and $q_i^{\text{laz}+1} \rightarrow q_i^{\text{laz}+2}$.

We will examine the four possible scenarios for the above transitions in $A^{\text{laz}}$:

1. The transition $q_i^{\text{laz}} \rightarrow q_i^{\text{laz}+1}$ occurred due to a corresponding event $s_u$ arriving from the input stream, and the transition $q_i^{\text{laz}+1} \rightarrow q_i^{\text{laz}+2}$ occurred due to a corresponding event $s_v$ taken from the input buffer. Then we may conclude that $s_v$ arrived before $s_u$. Hence, during evaluation in $A^{\text{laz}}$, after time $t_k$ the event $s_v$ is either already located in the input buffer, or will arrive eventually before $s_u$, triggering the transition $q_i^{\text{laz}'} \rightarrow q_i^{\text{laz}'}$, adding itself to the match buffer and removing itself from the input buffer if taken from there. Then, the event $s_v$ will eventually be received from the input stream and trigger the transition $q_i^{\text{laz}'} \rightarrow q_i^{\text{laz}'}$, adding itself to the match buffer. Thus, at time $t_l$, both $A^{\text{laz}}$ and $A^{\text{laz}'}$ contain $s_u$ and $s_v$ in their respective match buffers and do not contain $s_v$ in their input buffers.

2. The transition $q_i^{\text{laz}} \rightarrow q_i^{\text{laz}+1}$ occurred due to a corresponding event $s_u$ taken from the input buffer, and the transition $q_i^{\text{laz}+1} \rightarrow q_i^{\text{laz}+2}$ occurred due to a corresponding event $s_v$ arriving from the input stream. Then, during evaluation in $A^{\text{self}}$, the event $s_u$ is already located in the input buffer by the time $s_v$ arrives from the input stream. When $s_v$ arrives, it triggers the transition $q_i^{\text{laz}'} \rightarrow q_i^{\text{laz}'}$ and adds itself to the match buffer. Then, the event $s_u$ will be immediately received from the input stream and will trigger the transition $q_i^{\text{laz}'} \rightarrow q_i^{\text{laz}'}$, adding itself to the match buffer and removing itself from the input buffer. Thus, at time $t_l$, both $A^{\text{laz}}$ and $A^{\text{laz}'}$ contain $s_u$ and $s_v$ in their respective match buffers and do not contain $s_v$ in their input buffers.

3. Both $q_i^{\text{laz}} \rightarrow q_i^{\text{laz}+1}$ and $q_i^{\text{laz}+1} \rightarrow q_i^{\text{laz}+2}$ were triggered by the corresponding events $s_u, s_v$ arriving on the input stream. Then we may conclude that $s_u$ arrived before $s_v$. During evaluation in $A^{\text{laz}'}$, the event $s_u$ will be inserted to the input buffer upon its arrival, which will occur after $t_k$. Later, the event $s_v$ will trigger the transition $q_i^{\text{laz}'} \rightarrow q_i^{\text{laz}'}$, adding itself to the match buffer. Immediately afterwards, the event $s_u$ located in the input buffer will trigger $q_i^{\text{laz}'} \rightarrow q_i^{\text{laz}'}$, adding itself to the match buffer and removing itself from the input buffer. Thus, at time $t_l$, both $A^{\text{laz}}$ and $A^{\text{laz}'}$ contain $s_u$ and $s_v$ in their respective match buffers and do not contain $s_u$ in their input buffers.

4. Both $q_i^{\text{laz}} \rightarrow q_i^{\text{laz}+1}$ and $q_i^{\text{laz}+1} \rightarrow q_i^{\text{laz}+2}$ were triggered by the corresponding events taken from the input buffer. Then, during evaluation in $A^{\text{laz}'}$, at time $t_k$ the events which triggered both those transitions will already be located in the input buffer (by the assumption above) and will trigger the transitions $q_i^{\text{laz}'} \rightarrow q_i^{\text{laz}'}$ and $q_i^{\text{laz}'} \rightarrow q_i^{\text{laz}'}$ respectively, removing themselves from the input buffer and adding themselves to the match buffer exactly as occurred during evaluation in
Thus, at time \( t_l \), both \( A_{\text{lazy}} \) and \( A_{\text{lazy}}' \) contain \( s_u \) and \( s_v \) in their respective match buffers and do not contain \( s_u \) and \( s_v \) in their input buffers.

Thus, we have shown that in all cases the states of both \( A_{\text{lazy}} \) and \( A_{\text{lazy}}' \) are identical at \( t_l \). As stated above, this claim completes the proof.

**Lemma A.0.2.** Let \( P \) be a sequence pattern over event types \( e_1, \ldots, e_k, \ldots, e_n \), with a single negated event \( e_k \). Let \( A_{\text{eager}} \) be an eager NFA and let \( A_{\text{lazy}} \) be a Chain NFA derived from \( P \). Then \( A_{\text{eager}} \) and \( A_{\text{lazy}} \) are equivalent.

We prove this lemma separately for two possible types of \( A_{\text{lazy}} \), Post-Processing Chain NFAs and First-Chance Chain NFAs.

Let us first consider the case in which \( A_{\text{lazy}} \) is implemented as a Post-Processing Chain NFA. Let \( S = s_1, \ldots, s_m \) be a stream of events. Assume without loss of generality that all events in \( S \) are within the time window \( W \). If \( S \) does not contain an instance of a negated event \( e_k \), then, by Lemma A.0.1, it is accepted by \( A_{\text{eager}} \) if and only if it is accepted by \( A_{\text{lazy}} \). Otherwise, we examine the two possible scenarios:

1. An event \( s_l \) of type \( e_k \) exists in \( S \) and satisfies the conditions required by the pattern. In this case, \( A_{\text{eager}} \) will discard all intermediate results and no match will be reported. In \( A_{\text{lazy}} \), by construction, a state \( r \) exists, which is responsible for detecting \( e_k \). If the execution for all matches never reaches \( r \), then, since \( r \) precedes \( F \), no instance will reach the accepting state and thus no match will be reported. Otherwise, if \( r \) is reached by some instance after \( s_l \) has arrived, then \( s_l \) will be located in the input buffer, by construction of a Post-Processing Chain NFA. Then, the edge \( e_r^{\text{take}} \) will take \( s_l \) and proceed to the rejecting state, thus discarding the instance. Finally, if \( r \) is reached by some instance before \( s_l \) has arrived, then \( e_k \) must appear at pattern end, after all positive events have already arrived. By construction, \( r \) will only have an outgoing edge \( e_r^{\text{timeout}} \) to the accepting state, and the resulting search failed event will not be processed. Following the arrival of \( s_l \), the edge \( e_r^{\text{take}} \) will proceed to the rejecting state, since conditions for \( s_l \) are satisfied. Thus, in any case, \( A_{\text{lazy}} \) will discard all its partial matches.

2. An event \( s_l \) of type \( e_k \) exists in \( S \) and does not satisfy the conditions required by the pattern. In this case, \( A_{\text{eager}} \) will ignore \( s_l \) and report all matches. In \( A_{\text{lazy}} \), by construction, the edge \( e_r^{\text{take}} \) of a state \( r \) will never be traversed, since its conditions do not hold for \( s_l \). If \( r \) is reached by some instance after \( s_l \) has arrived, then \( s_l \) will be located in the input buffer, and a search will trigger the search failed event, which will cause the instance to proceed to \( F \). Otherwise, if \( r \) is reached by some instance before \( s_l \) has arrived, then \( e_k \) must appear at pattern end, after all positive events have already arrived. By construction, \( r \) will only have an outgoing edge \( e_r^{\text{timeout}} \) edge to the accepting state. When \( s_l \) arrives, the edge \( e_r^{\text{take}} \) will not be traversed,
and the instance will only be affected by the *timeout* event, which will cause a transition to $F$. Thus, in any case, $A_{\text{lazy}}$ will report all its positive matches.

To summarize, we have shown that in all cases $A_{\text{eager}}$ and $A_{\text{lazy}}$ report the same matches when applied on a stream $S$.

Now, let $A_{\text{lazy}}$ be implemented as a First-Chance Chain NFA. Again, let $S = s_1, \ldots, s_m$ be a stream of events and assume w.l.o.g. that all events in $S$ are within the time window $W$. If $S$ does not contain an instance of a negated event $e_k$, then, by Lemma A.0.1, it is accepted by $A_{\text{eager}}$ if and only if it is accepted by $A_{\text{lazy}}$. Otherwise, we examine two possible scenarios:

1. An event $s_l$ of type $e_k$ exists in $S$ and satisfies the conditions required by the pattern. In this case, $A_{\text{eager}}$ will discard all intermediate results and no match will be reported. In $A_{\text{lazy}}$, by construction, a *take* edge $e_k^{\text{take}}$ exists, detecting an event of type $e_k$ and proceeding to the rejecting state $R$. Let $q$ denote the source state of $e_k^{\text{take}}$. If the execution for all matches never reaches $q$, then, since $q$ precedes $F$, no instance will reach the accepting state and no match will be reported. Otherwise, if $q$ is reached by some instance after $s_l$ has arrived, then $s_l$ will be located in the input buffer, by construction of $A_{\text{lazy}}$. Then, $e_k^{\text{take}}$ will take $s_l$ and proceed to $R$. Note that $q$ cannot be reached before $s_l$ has arrived, since in this case $e_k$ must appear at the pattern’s end (by construction, requiring the positive event succeeding $e_k$ to be taken prior to entering $q$), and this type of pattern is not supported by First-Chance Chain NFAs. Thus, in any case, $A_{\text{lazy}}$ will discard all its partial matches.

2. An event $s_l$ of type $e_k$ exists in $S$ and does not satisfy the conditions required by the pattern. In this case, $A_{\text{eager}}$ will ignore $s_l$ and report all matches. In $A_{\text{lazy}}$, by construction, $e_k^{\text{take}}$ will never be traversed, since its conditions do not hold for $s_l$. With the exception of $e_k^{\text{take}}$, the structure of $A_{\text{lazy}}$ is identical to that of a lazy sequence Chain NFA accepting a pattern $P$ with $e_k$ ignored. Therefore, by Lemma A.0.1, all matches reported by $A_{\text{eager}}$ will also be reported by $A_{\text{lazy}}$.

Thus, we have shown that in all cases $A_{\text{eager}}$ and $A_{\text{lazy}}$ report the same matches when applied on a stream $S$, which completes the proof.

**Corollary A.2.** Let $P$ be a sequence pattern over event types $e_1, \ldots, e_n$, with an arbitrary number $1 \leq m < n$ of negated events. Let $A_{\text{eager}}$ be an eager NFA and let $A_{\text{lazy}}$ be a Chain NFA derived from $P$. Then $A_{\text{eager}}$ and $A_{\text{lazy}}$ are equivalent.

The proof is by induction on $m$, with Lemma A.0.2 serving as an induction basis. The induction step for $m = i + 1$ is proven in the same way as Lemma A.0.2, using the induction hypothesis instead of Lemma A.0.1 for assuming the correctness of the pattern without the $m^{\text{th}}$ negated event.
Lemma A.0.3. Let $P$ be a sequence pattern over event types $e_1, \ldots, e_k, \ldots, e_n$, with a single iterated event $e_k$. Let $A_{\text{eager}}$ be an eager NFA and let $A_{\text{lazy}}$ be a Chain NFA derived from $P$. Then $A_{\text{eager}}$ and $A_{\text{lazy}}$ are equivalent.

We will prove this lemma by double inclusion, i.e., by showing that, given a potential match $M = s_1, \ldots, s_m$ for pattern $P$, $A_{\text{eager}}$ accepts $M$ if and only if $A_{\text{lazy}}$ does. This, in turn, will be proven by induction on the number of events of type $e_k$ in $M$.

For the induction basis, assume that $M$ contains only a single event of type $e_k$. Then, let $P'$ be a pattern identical to $P$, but with $e_k$ as a non-iterated type. By Lemma A.0.1, if $M$ is a match for $P$, then it is also a match for $P'$. By iteration definition, both $A_{\text{eager}}$ and $A_{\text{lazy}}$ also accept $P'$. Hence, either both NFA accept the match $M$, or both reject it.

For the induction step, assume that the condition holds for up to $i$ events of type $e_k$, and let $M$ contain $i+1$ events of type $e_k$: $s_1^k, s_2^k, \ldots, s_{i+1}^k$.

Let $M$ be accepted by $A_{\text{eager}}$. Let $t_{i+1}$ be the arrival time of $s_{i+1}^k$. Then, a state $q$ exists in $Q_{\text{eager}}$, such that at time $t_{i+1}$ the NFA instance of $A_{\text{eager}}$ is in $q$. Since $i \geq 1$, $q$ is a state containing a self-loop taking $e_k$. Hence, before and after $t_{i+1}$, the current state was $q$. Since $M$ is accepted, we know that all events that arrived after $t_{i+1}$ caused the NFA instance to reach $F$ from $q$. Let $M'$ be a match identical to $M$, but without $s_{i+1}^k$. By the observation above, $M'$ is also accepted by $A_{\text{eager}}$. By the induction hypothesis, $M'$ is thus accepted by $A_{\text{lazy}}$. Now, since $M$ is accepted by $A_{\text{eager}}$, it follows that $s_{i+1}^k$ satisfies the conditions with other events in $M$. Therefore, when an iterate edge in $A_{\text{lazy}}$ fetches a subset $s_1^k, \ldots, s_{i+1}^k$, it is added to the match buffer. By construction of a Lazy Iteration Chain NFA, the edge is traversed, reaching $F$. Hence, $M$ is accepted by $A_{\text{lazy}}$.

Now, let $M$ be accepted by $A_{\text{lazy}}$. Consequently, when a subset $s_1^k, \ldots, s_{i+1}^k$ is fetched by an iterate edge of $A_{\text{lazy}}$, all events in this subset satisfy the conditions with other events. In particular, it follows that a subset $s_1^k, \ldots, s_{i}^k$ satisfies the conditions as well. It also follows that during evaluation of $M$ some NFA instance reaches the state with an outgoing iterate edge. Hence, during the same traversal which retrieves $s_1^k, \ldots, s_{i+1}^k$, a subset $s_1^k, \ldots, s_{i}^k$ is also generated and causes the transition to occur. That is, the match $M'$ containing $s_1^k, \ldots, s_{i}^k$ without $s_{i+1}^k$ is also accepted by $A_{\text{lazy}}$. By the induction hypothesis, $M'$ is thus accepted by $A_{\text{eager}}$. By definition of the eager NFA, after $s_{i}^k$ arrives during evaluation, an instance exists whose current state is a state $q$ with a self-loop $e_{\text{loop}}$ taking $e_k$. When $s_{i+1}^k$ arrives, this instance will thus attempt a traversal of $e_{\text{loop}}$. By assumption, $s_1^k, \ldots, s_{i+1}^k$ satisfies all the conditions required by the pattern, hence this transition will succeed. Since $M'$ is accepted, we know that all events that arrived after $s_i^k$ in $M'$ caused the NFA instance to reach $F$ from $q$. Consequently, the same event sequence will cause the same transitions in $M$, i.e., $M$ is accepted by $A_{\text{eager}}$.

To summarize, we have shown that, given a sequence pattern $M$ containing an iterated event, $A_{\text{eager}}$ accepts $M$ if $A_{\text{lazy}}$ does, and vice versa, which proves their equivalence.
Corollary A.3. Let $P$ be a sequence pattern over event types $e_1, \ldots, e_n$, with an arbitrary number $1 \leq m < n$ of iterated events. Let $A_{\text{eager}}$ be an eager NFA and let $A_{\text{lazy}}$ be a Chain NFA derived from $P$. Then $A_{\text{eager}}$ and $A_{\text{lazy}}$ are equivalent.

The proof is identical to that of Corollary A.2, using Lemma A.0.3 instead of Lemma A.0.2.

Definition A.0.4. Let $A = (Q, E, q_1, F, R)$ be an NFA. We will call $A$ a composite NFA of size $K$, if there exist automata $A_1, \cdots, A_k$, such that $A$ consists of the union of them, with their respective initial states merged into $q_1$, the accepting states merged into $F$, and the rejecting states merged into $R$.

Lemma A.0.5. Let $A_1, A_2$ be two composite NFAs of size $K$, i.e., $A_1$ consists of $A^1_1, \cdots A^1_k$ and $A_2$ consists of $A^2_1, \cdots A^2_k$. Then, if the sub-automata of $A_1$ and $A_2$ are pairwise equivalent, i.e., $(A^1_1 \equiv A^2_1) \land \cdots \land (A^1_k \equiv A^2_k)$, then $A_1$ and $A_2$ are equivalent.

The proof is by induction on $K$. For $K = 1$, the correctness follows immediately, since $A_1 = A^1_1 \equiv A^2_1 = A_2$. For the induction step, assume the claim to hold for $K = i$ and let $A_1, A_2$ be two composite NFAs of size $i + 1$. Now, assume that all sub-automata of $A_1$ and $A_2$ are equivalent. We will show the equivalence of $A_1$ and $A_2$ by mutual inclusion.

Let $M = s_1, \ldots, s_m$ be a potential match for $A_1, A_2$. Examine the following possible situations:

1. $M$ is accepted by $A^1_{i+1}$. Then, by construction of $A_1$, it also accepts $M$. On the other side, by assumption of equivalence of $A^1_{i+1}$ and $A^2_{i+1}$, $M$ is also accepted by $A^2_{i+1}$. By construction of $A_2$, it also accepts $M$.

2. $M$ is accepted by $A_1$, but not accepted by $A^1_{i+1}$. Let $\bar{A}_1, \bar{A}_2$ be constructed by removing all the internal states of $A^2_{i+1}$ and $A^1_{i+1}$, respectively. $M$ is accepted by $\bar{A}_1$, according to the assumption. Then, by the induction hypothesis, $M$ is also accepted by $\bar{A}_2$, since $\bar{A}_1, \bar{A}_2$ are composite NFAs of size $i$. By construction of $\bar{A}_2$, $M$ is also accepted by $\bar{A}_2$.

3. $M$ is not accepted by $A_1$. Then, in particular, it is not accepted by any of its sub-automata $A^1_1, \cdots A^1_k, A^1_{i+1}$. By assumption of equivalence, $M$ is also not accepted by any of $A^2_1, \cdots A^2_k, A^2_{i+1}$. By construction of $A_2$, it also does not accept $M$.

To summarize, we have shown that, for any potential match $M$, it is either accepted by both $A_1$ and $A_2$ or by neither of them, which completes the proof.

Lemma A.0.6. Let $P$ be a partial sequence pattern over event types $e_1, \cdots, e_n$, with possible negated or iterated events. Let $A_{\text{chain}}$ be a Chain NFA for $P$, as presented in Section 2.4.3. Let $\text{SEQ} = \{\text{seq}_1, \cdots, \text{seq}_k\}$ be a set of all sequence patterns whose orders are allowed by $P$ (i.e., $P$ is a union of patterns in $\text{SEQ}$). Finally, let $A_{\text{multi-chain}}$
be a Lazy Multi-Chain NFA (Section 2.5.1), with each sub-chain being a Chain NFA for a sequence pattern in SEQ (Section 2.4.1). Then $A_{\text{chain}}$ and $A_{\text{multi-chain}}$ are equivalent.

The proof is by mutual inclusion. Let $M = s_1, \ldots, s_m$ be a potential match for $A_1, A_2$. Examine the following scenarios:

1. $M$ is accepted by $A_{\text{multi-chain}}$. Then, in particular, it is accepted by at least one of its sub-chains. By construction, this Chain NFA $A_i$ and $A_{\text{chain}}$ are identical except for $A_i$ having stricter ordering constraints. Consequently, $M$ is also accepted by $A_{\text{chain}}$.

2. $M$ is rejected by $A_{\text{multi-chain}}$. Then, it is also rejected by each of its sub-chains. Observe that the only difference between $A_{\text{chain}}$ and each of the sub-chains is the ordering constraint definition. Hence, the reason for the rejection can only be related to violating temporal conditions. Assume w.l.o.g. that an event $s_p$ of type $e_u$ in $M$ precedes an event $s_r$ of type $e_v$, which is forbidden by all chains. Also, assume that $e_v$ precedes $e_u$ in the frequency order (the opposite case is symmetrical). Then, as $A_{\text{chain}}$ reaches a state responsible for taking $e_u$, $s_r$ is already located in the match buffer. By definition of an ordering filter $\text{succ}_i$ for Chain NFA for partial sequences, $\text{succ}_i$ will contain $e_v$, hence $s_p$ will not be taken by $A_{\text{chain}}$, discarding $M$.

**Corollary A.4.** Let $P$ be a conjunction pattern over event types $e_1, \ldots, e_n$, with possible negated or iterated events. Let $A_{\text{chain}}$ be a Chain NFA for $P$. Let us define $\text{SEQ} = \{ \text{seq}_1, \ldots, \text{seq}_k \}$ as a set of all sequences over $e_1, \ldots, e_n$. Let $A_{\text{multi-chain}}$ be a Lazy Multi-Chain NFA with each sub-chain being a Chain NFA for a sequence pattern in $\text{SEQ}$. Then $A_{\text{chain}}$ and $A_{\text{multi-chain}}$ are equivalent.

The proof immediately follows from Lemma A.0.6, since a conjunction pattern is an edge case of a partial sequence, with no ordering constraints between the primitive events.

**Corollary A.5.** Let $P$ be a partial sequence pattern over $e_1, \ldots, e_n$. Let $A_{\text{eager}}$ be an eager NFA and let $A_{\text{lazy}}$ be a Chain NFA derived from $P$. Then $A_{\text{eager}}$ and $A_{\text{lazy}}$ are equivalent.

Assume w.l.o.g. that $A_{\text{eager}}$ is implemented as a composite NFA, containing a sub-automaton for each valid sequence in $P$. An NFA with this structure necessarily exists, since each partial sequence can be represented as a disjunction of sequences. Let $A_{\text{multi-chain}}$ be a Lazy Multi-Chain NFA with each sub-chain being a Chain NFA for each valid sequence in $P$. By Lemma A.0.5, $A_{\text{eager}}$ and $A_{\text{multi-chain}}$ are equivalent. By Lemma A.0.6, $A_{\text{multi-chain}}$ and $A_{\text{lazy}}$ are equivalent. Then, by transitivity, $A_{\text{eager}}$ and $A_{\text{lazy}}$ are equivalent.
Corollary A.6. Let $P$ be a conjunction pattern over event types $e_1, \cdots, e_n$. Let $A_{\text{eager}}$ be an eager NFA and let $A_{\text{lazy}}$ be a Chain NFA derived from $P$. Then $A_{\text{eager}}$ and $A_{\text{lazy}}$ are equivalent.

The proof immediately follows from Corollary A.5.

Lemma A.0.7. Let $P$ be a disjunction pattern over $e_1, \cdots, e_n$, i.e., $P = OR(p_1, \cdots, p_m)$, where each $p_i$ is a pattern over sequence, conjunction, partial sequence, negation and iteration operators. Let $A_{\text{eager}}$ be an eager NFA and let $A_{\text{lazy}}$ be a Lazy Multi-Chain NFA derived from $P$. Then $A_{\text{eager}}$ and $A_{\text{lazy}}$ are equivalent.

Assume w.l.o.g. that $A_{\text{eager}}$ is implemented as a composite NFA, containing a sub-automaton for each sub-pattern $p_i$. Then, $A_{\text{eager}}$ and $A_{\text{lazy}}$ are both composite NFA of size $m$. By definition of $p_i$, by Lemma A.0.1, and by Corollaries A.2, A.3, A.5, and A.6, each sub-automaton for $p_i$ in $A_{\text{eager}}$ is equivalent to a corresponding sub-chain in $A_{\text{lazy}}$. Then, by Lemma A.0.5, $A_{\text{eager}}$ and $A_{\text{lazy}}$ are equivalent.

We will now complete the proof of Theorem A.1. If $P$ does not contain the disjunction operator, the equivalence of $A_{\text{eager}}$ and $A_{\text{lazy}}$ follows from Lemma A.0.1 and from Corollaries A.2, A.3, A.5, and A.6. Otherwise, let $P'$ be the DNF form of pattern $P$. It is sufficient to prove that the claim holds for $P'$. By definition of DNF, $P'$ is a disjunction pattern. Thus, by Lemma A.0.7, the corresponding eager and lazy NFA for $P'$ are equivalent. ■
Appendix B

ASI Property of the Order-Based Cost Functions

In this appendix, we will formally prove that the order-based CPG cost functions $\text{Cost}_{\text{trpt}}^{\text{ord}}$ and $\text{Cost}_{\text{lat}}^{\text{ord}}$ presented in Sections 3.4.1 and 3.6.1 respectively have the adjacent sequence interchange (ASI) property, defined in [MS79]. As we discussed in Section 3.4.3, polynomial-time algorithms were developed for join ordering of acyclic queries subject to cost functions that have this property [IK84, KBZ86]. Since all JQPG algorithms demonstrated in Chapter 3 are executed subject to the left-deep tree cost function $\text{Cost}_{\text{LDJ}}$, we do not use this result directly for solving the CPG problem. However, the throughput- and latency-related functions can potentially be employed directly for solving the join ordering problem in streaming database systems, and hence it is important to show their ASI property as a part of our work.

We will start with the definition of the ASI property.

**Definition B.0.1.** A cost function $C$ has the adjacent sequence interchange (ASI) property, if and only if there exists a rank function $\text{rank}(s)$ for sequences $s$, such that for all sequences $a, b$ and for all non-empty sequences $v, u$ the following holds:

$$C(auvb) \leq C(avub) \iff \text{rank}(u) \leq \text{rank}(v).$$

We will first provide the proof for the throughput-related cost function $\text{Cost}_{\text{ord}}^{\text{trpt}}$. To that end, we will utilize the idea from a similar proof in [CM95].

**Theorem B.1.** The cost function $\text{Cost}_{\text{ord}}^{\text{trpt}}$ has the ASI property.

Let $P$ be a pure conjunctive pattern over the event types $T_1, \cdots, T_n$ with an acyclic query graph. Recall that the cost function is defined as follows:

$$\text{Cost}_{\text{ord}}^{\text{trpt}}(O) = \sum_{k=1}^{n} \left( W^k \prod_{i=1}^{k} r_{p_i} \prod_{i,j \leq k, i \leq j} \text{sel}_{p_i,p_j} \right),$$
where $O = (T_{p_1}, T_{p_2}, \cdots, T_{p_n}) ; p_i \in [1, n]$.

Due to the acyclicity of the pattern, each event type $T_{p_i}$ will only have one predicate with the event types preceding it in $O$. Further, if we set the root of the query tree at some event type $T_R$, this predicate can be uniquely determined for any other type, as follows from the uniqueness of a path between two nodes in a tree. For each $T_i \neq T_R$, we will denote this predicate as $c^R_i$, and its selectivity will be thus denoted as $sel^R_i$. For the root event type, we set $sel^R_R = 1$. Rewriting the cost function definition accordingly, we get:

$$Cost^{trpt}_ord (O) = \sum_{k=1}^{n} \prod_{i=1}^{k} (W \cdot r_{p_i} \cdot sel^R_i).$$

We will now define the following auxiliary functions, defined on any sequence $s$ of size $m$, consisting of events $T_1, \cdots, T_n$:

$$C(s) = \sum_{k=1}^{m} \prod_{i=1}^{k} (W \cdot r_{p_i} \cdot sel^R_i) ; C(\varepsilon) = 0$$

$$T(s) = \prod_{i=1}^{m} (W \cdot r_{p_i} \cdot sel^R_i) ; T(\varepsilon) = 1.$$

Note that the above functions only depend on the selection of the root $R$.

It can be observed that $C(O) = Cost^{trpt}_ord (O)$ for any evaluation order $O$. In addition, the following holds:

$$C(s_1 s_2) = C(s_1) + T(s_1) \cdot C(s_2).$$

We will now define the rank function as follows:

$$rank(s) = \frac{T(s) - 1}{C(s)}$$

We will demonstrate the ASI property for the rank function $rank$. Let $a, b$ arbitrary sequences and $v, u$ arbitrary non-empty sequences. Then:

$$C(auvb) \leq C(auvb) \Leftrightarrow$$

$$C(a) + T(a) \cdot C(ub) \leq C(a) + T(a) \cdot C(vb) \Leftrightarrow$$

$$C(u) + T(u) \cdot C(vb) \leq C(v) + T(v) \cdot C(ub) \Leftrightarrow$$

$$C(u) + T(u) \cdot (C(v) + T(v) \cdot C(b)) \leq C(v) + T(v) \cdot (C(u) + T(u) \cdot C(vb)) \Leftrightarrow$$

$$C(u) + T(u) \cdot C(v) + T(u) \cdot T(v) \cdot C(b) \leq C(v) + T(v) \cdot C(u) + T(v) \cdot T(u) \cdot C(b) \Leftrightarrow$$

$$T(u) \cdot C(v) - C(v) \leq T(v) \cdot C(u) - C(u) \Leftrightarrow$$

$$\frac{T(u) - 1}{C(u)} \leq \frac{T(v) - 1}{C(v)} \Leftrightarrow$$

$$rank(u) \leq rank(v).$$
The above transitions are possible due to the positivity of $T(s)$ and the positivity of $C(s)$ for non-empty sequences.

We will now proceed to the latency-related cost function $\text{Cost}_{\text{ord}}^{\text{lat}}$.

**Theorem B.2.** The cost function $\text{Cost}_{\text{ord}}^{\text{lat}}$ has the ASI property.

Let $P$ be a pure conjunctive pattern over the event types $T_1, \cdots, T_n$ with an acyclic query graph. Recall that the cost function is defined as follows:

$$\text{Cost}_{\text{ord}}^{\text{lat}}(O) = \sum_{T_i \in \text{Succ}_O(T_n)} W \cdot r_i,$$

where $O = (T_{p_1}, T_{p_2}, \cdots, T_{p_n}); p_i \in [1, n]$, $T_n$ is the last event type in the order induced by the pattern, and $\text{Succ}_O(T_n)$ denotes the event types succeeding $T_n$ in $O$.

For each sequence $s$ over the above types, we will define the following function accepting an event type $T_i$ as a parameter:

$$g_s(T_i) = \begin{cases} W \cdot r_i & \text{if } T_i \in \text{Succ}_s(T_n) \\ 0 & \text{otherwise.} \end{cases}$$

$\text{Cost}_{\text{ord}}^{\text{lat}}$ can be now rewritten in the following form:

$$\text{Cost}_{\text{ord}}^{\text{lat}}(O) = \sum_{i=1}^{n} g_O(T_i).$$

The rank function will be defined as follows:

$$\text{rank}(s) = \begin{cases} \sum_{T_i \in s} g_s(T_i) & \text{if } T_n \in s \\ 0 & \text{otherwise.} \end{cases}$$

We will now demonstrate the ASI property of $\text{Cost}_{\text{ord}}^{\text{lat}}$ by examining the following cases for non-empty sequences $u, v$:

1. $T_n \notin u$ and $T_n \notin v$: in this case, $\text{rank}(u) = \text{rank}(v) = 0$. Examine the expressions $\text{Cost}_{\text{ord}}^{\text{lat}}(auvb)$ and $\text{Cost}_{\text{ord}}^{\text{lat}}(avub)$ for some sequences $a, b$. If $T_n \notin a$, then obviously $u$ and $v$ do not contribute any event type with a non-zero value of $g_{auvb}(T_i)$ (or $g_{avub}(T_i)$, symmetrically), hence $\text{Cost}_{\text{ord}}^{\text{lat}}(auvb) = \text{Cost}_{\text{ord}}^{\text{lat}}(avub) = \text{Cost}_{\text{ord}}^{\text{lat}}(ab)$. Otherwise, since $u, v, b \subseteq \text{Succ}_{auvb}(T_n)$, the cost function for $u, v, b$ is commutative by definition and thus $\text{Cost}_{\text{ord}}^{\text{lat}}(auvb) = \text{Cost}_{\text{ord}}^{\text{lat}}(avub)$.

2. $T_n \in v$: in this case, we have $\text{rank}(v) \geq 0$ by non-negativity of $g_s$ and $\text{rank}(u) = 0$ since $u$ and $v$ must be disjoint. Consequently, $\text{rank}(u) \leq \text{rank}(v)$. Now, examine
the first cost expression:

$$\text{Cost}_{\text{ord}}(auvb) = \sum_{T_i \in auvb} g_{auvb}(T_i) =$$

$$= \sum_{T_i \in a} g_{auvb}(T_i) + \sum_{T_i \in u} g_{auvb}(T_i) + \sum_{T_i \in v} g_{auvb}(T_i) + \sum_{T_i \in b} g_{auvb}(T_i) =$$

$$= 0 + 0 + \text{rank}(v) + \sum_{T_i \in b} (W \cdot r_i).$$

For the second cost expression, we obtain the following:

$$\text{Cost}_{\text{ord}}(avub) = \sum_{T_i \in avub} g_{avub}(T_i) =$$

$$= \sum_{T_i \in a} g_{avub}(T_i) + \sum_{T_i \in v} g_{avub}(T_i) + \sum_{T_i \in u} g_{avub}(T_i) + \sum_{T_i \in b} g_{avub}(T_i) =$$

$$= 0 + \text{rank}(v) + \sum_{T_i \in u} (W \cdot r_i) + \sum_{T_i \in b} (W \cdot r_i).$$

The difference between these expressions is:

$$\text{Cost}_{\text{ord}}(avub) - \text{Cost}_{\text{ord}}(auvb) = \sum_{T_i \in u} (W \cdot r_i).$$

By non-negativity of time windows and arrival rates, we thus get $\text{Cost}_{\text{ord}}(auvb) \leq \text{Cost}_{\text{ord}}(avub)$.

3. $T_n \in u$: this case is symmetrical to case 2, and identical steps are applied to show that $\text{rank}(v) \leq \text{rank}(u)$ and $\text{Cost}_{\text{ord}}(avub) \leq \text{Cost}_{\text{ord}}(auvb)$.

To summarize, we have demonstrated that in all cases the condition of the ASI property holds, which completes the proof. ■
Appendix C

Additional Experimental Results - Performance of the CPG and JQPG Methods by Pattern Type

Figures C.1-C.10 present the results discussed in Section 3.7.3 separately for each of the five pattern categories described in Section 3.7.2. Each graph demonstrates either throughput or memory consumption obtained for the specified pattern type as a function of the pattern size (in terms of the number of the participating event types).

Figures C.1, C.3, C.5, C.7, and C.9 depict the throughput measured during each of the respective experiments. Although the performance of all methods degrades drastically as the pattern size grows, the relative throughput gain for JQPG methods over native CPG methods is consistently higher for larger patterns. As an example, examine the experimental results for the tree-based evaluation of the conjunction patterns over traffic data (Figure C.5(d)). Here, the most efficient JQPG algorithm (DP-B) achieves up to 16 times higher throughput than the native CPG framework (ZSTREAM) for patterns of length 7, compared to a speedup of only 1.7 times for patterns of 3 events. The results for memory consumption follow the same trend (Figures C.2, C.4, C.6, C.8, and C.10). We can thus conclude that, at least for the pattern sizes considered in this study, the JQPG methods provide a considerably more scalable solution.
Figure C.1: Throughput as a function of the sequence pattern size: (a) stock dataset, order-based methods; (b) stock dataset, tree-based methods; (c) traffic dataset, order-based methods; (d) traffic dataset, tree-based methods.

Figure C.2: Memory consumption as a function of the sequence pattern size: (a) stock dataset, order-based methods; (b) stock dataset, tree-based methods; (c) traffic dataset, order-based methods; (d) traffic dataset, tree-based methods.
Figure C.3: Throughput as a function of the negation pattern size: (a) stock dataset, order-based methods; (b) stock dataset, tree-based methods; (c) traffic dataset, order-based methods; (d) traffic dataset, tree-based methods.

Figure C.4: Memory consumption as a function of the negation pattern size: (a) stock dataset, order-based methods; (b) stock dataset, tree-based methods; (c) traffic dataset, order-based methods; (d) traffic dataset, tree-based methods.
Figure C.5: Throughput as a function of the conjunction pattern size: (a) stock dataset, order-based methods; (b) stock dataset, tree-based methods; (c) traffic dataset, order-based methods; (d) traffic dataset, tree-based methods.

Figure C.6: Memory consumption as a function of the conjunction pattern size: (a) stock dataset, order-based methods; (b) stock dataset, tree-based methods; (c) traffic dataset, order-based methods; (d) traffic dataset, tree-based methods.
Figure C.7: Throughput as a function of the Kleene closure pattern size: (a) stock dataset, order-based methods; (b) stock dataset, tree-based methods; (c) traffic dataset, order-based methods; (d) traffic dataset, tree-based methods.

Figure C.8: Memory consumption as a function of the Kleene closure pattern size: (a) stock dataset, order-based methods; (b) stock dataset, tree-based methods; (c) traffic dataset, order-based methods; (d) traffic dataset, tree-based methods.
Figure C.9: Throughput as a function of the disjunction pattern size: (a) stock dataset, order-based methods; (b) stock dataset, tree-based methods; (c) traffic dataset, order-based methods; (d) traffic dataset, tree-based methods.

Figure C.10: Memory consumption as a function of the disjunction pattern size: (a) stock dataset, order-based methods; (b) stock dataset, tree-based methods; (c) traffic dataset, order-based methods; (d) traffic dataset, tree-based methods.
Appendix D

Additional Experimental Results
- Performance of the Different Adaptation Methods by Pattern Type

This appendix extends the experimental study of the adaptive mechanisms discussed in Section 4.5 by organizing and presenting them by pattern type.

Five distinct set of patterns were used throughout the experiments as specified below:

1. Sequence patterns - this set contains patterns with a single SEQ operator, similar to the one demonstrated in Example 4.1.1. The results for sequence pattern set follow the same trends as those discussed in Section 4.5.2 and are displayed in Figures D.1-D.4.

2. Conjunction patterns - contains patterns with a single AND operator. Each pattern in this set can be obtained by taking the pattern of the same size from

![Figure D.1: Comparison of the adaptation methods applied on the traffic dataset in conjunction with the greedy algorithm (sequence patterns): (a) throughput; (b) relative throughput gain over the non-adaptive method; (c) total number of plan reoptimizations; (d) computational overhead.](image)
Figure D.2: Comparison of the adaptation methods applied on the traffic dataset in conjunction with ZStream algorithm (sequence patterns): (a) throughput; (b) relative throughput gain over the non-adaptive method; (c) total number of plan reoptimizations; (d) computational overhead.

Figure D.3: Comparison of the adaptation methods applied on the stocks dataset in conjunction with the greedy algorithm (sequence patterns): (a) throughput; (b) relative throughput gain over the non-adaptive method; (c) total number of plan reoptimizations; (d) computational overhead.

Figure D.4: Comparison of the adaptation methods applied on the stocks dataset in conjunction with ZStream algorithm (sequence patterns): (a) throughput; (b) relative throughput gain over the non-adaptive method; (c) total number of plan reoptimizations; (d) computational overhead.
Figure D.5: Comparison of the adaptation methods applied on the traffic dataset in conjunction with the greedy algorithm (conjunction patterns): (a) throughput; (b) relative throughput gain over the non-adaptive method; (c) total number of plan reoptimizations; (d) computational overhead.

Figure D.6: Comparison of the adaptation methods applied on the traffic dataset in conjunction with ZStream algorithm (conjunction patterns): (a) throughput; (b) relative throughput gain over the non-adaptive method; (c) total number of plan reoptimizations; (d) computational overhead.

Figure D.7: Comparison of the adaptation methods applied on the stocks dataset in conjunction with the greedy algorithm (conjunction patterns): (a) throughput; (b) relative throughput gain over the non-adaptive method; (c) total number of plan reoptimizations; (d) computational overhead.
Figure D.8: Comparison of the adaptation methods applied on the stocks dataset in conjunction with ZStream algorithm (conjunction patterns): (a) throughput; (b) relative throughput gain over the non-adaptive method; (c) total number of plan reoptimizations; (d) computational overhead.

Figure D.9: Comparison of the adaptation methods applied on the traffic dataset in conjunction with the greedy algorithm (negation patterns): (a) throughput; (b) relative throughput gain over the non-adaptive method; (c) total number of plan reoptimizations; (d) computational overhead.

Figure D.10: Comparison of the adaptation methods applied on the traffic dataset in conjunction with ZStream algorithm (negation patterns): (a) throughput; (b) relative throughput gain over the non-adaptive method; (c) total number of plan reoptimizations; (d) computational overhead.
Figure D.11: Comparison of the adaptation methods applied on the stocks dataset in conjunction with the greedy algorithm (negation patterns): (a) throughput; (b) relative throughput gain over the non-adaptive method; (c) total number of plan reoptimizations; (d) computational overhead.

Figure D.12: Comparison of the adaptation methods applied on the stocks dataset in conjunction with ZStream algorithm (negation patterns): (a) throughput; (b) relative throughput gain over the non-adaptive method; (c) total number of plan reoptimizations; (d) computational overhead.

Figure D.13: Comparison of the adaptation methods applied on the traffic dataset in conjunction with the greedy algorithm (Kleene closure patterns): (a) throughput; (b) relative throughput gain over the non-adaptive method; (c) total number of plan reoptimizations; (d) computational overhead.
Figure D.14: Comparison of the adaptation methods applied on the traffic dataset in conjunction with ZStream algorithm (Kleene closure patterns): (a) throughput; (b) relative throughput gain over the non-adaptive method; (c) total number of plan reoptimizations; (d) computational overhead.

Figure D.15: Comparison of the adaptation methods applied on the stocks dataset in conjunction with the greedy algorithm (Kleene closure patterns): (a) throughput; (b) relative throughput gain over the non-adaptive method; (c) total number of plan reoptimizations; (d) computational overhead.

Figure D.16: Comparison of the adaptation methods applied on the stocks dataset in conjunction with ZStream algorithm (Kleene closure patterns): (a) throughput; (b) relative throughput gain over the non-adaptive method; (c) total number of plan reoptimizations; (d) computational overhead.
Figure D.17: Comparison of the adaptation methods applied on the traffic dataset in conjunction with the greedy algorithm (composite patterns): (a) throughput; (b) relative throughput gain over the non-adaptive method; (c) total number of plan reoptimizations; (d) computational overhead.

Figure D.18: Comparison of the adaptation methods applied on the traffic dataset in conjunction with ZStream algorithm (composite patterns): (a) throughput; (b) relative throughput gain over the non-adaptive method; (c) total number of plan reoptimizations; (d) computational overhead.

Figure D.19: Comparison of the adaptation methods applied on the stocks dataset in conjunction with the greedy algorithm (composite patterns): (a) throughput; (b) relative throughput gain over the non-adaptive method; (c) total number of plan reoptimizations; (d) computational overhead.
set 1 and removing the temporal constraints. For this set, the relative gain of the considered adaptive methods was higher than for other (non-composite) pattern sets. We can attribute this result to the lower total selectivity of the inter-event conditions (due to the lack of sequence constraints) and hence larger number of intermediate partial matches, resulting in higher importance of the correct adaptation decisions (Figures D.5-D.8).

3. Negation patterns - this set was produced from set 1 by adding a negated event (an event under the negation operator) in a random position in the pattern. Surprisingly, the introduction of this operator did not significantly affect the experimental results, exhibiting nearly identical relative throughput gains for all adaptive methods (Figures D.9-D.12).

4. Kleene closure patterns - consists of sequence patterns containing a single event under Kleene closure. This pattern demonstrated a significant deviation from the rest in terms of the throughput measured for the various adaptation methods. Due to the substantial complexity and high cost of the Kleene closure operator regardless of its position in the evaluation plan, the overall impact of the adaptation methods was considerably lower as compared to other pattern sets. Still, the invariant-based method was superior to the other algorithms in all scenarios (Figures D.13-D.16).

5. Composite patterns - each pattern in this category is a disjunction of three independent sequences. As each of the subsequences was evaluated independently, the obtained results were very similar to those observed for the sequence pattern set (Figures D.17-D.20).

Each set contained 6 patterns varying in length from 3 to 8. For sets 1-4, the pattern size was defined as the number of events in a pattern. Note that, while the events under the Kleene closure operator (set 4) are included in size calculation, while the negated events (set ) are excluded. For set 5, the definition was altered to reflect the number of events in each subpattern.
Bibliography


ACM.


על בסיס המודל הנ"ל ותוכו שימש ביבוב התuded המודל של הוותה אזוריות מונסציה של סדר שריוט, ונא
מתאימים נייש דחים בחישוב תכנית הערכה האופטימלית. אנו מראים כי קיים מידע רב בק הניה
שלני ליבות דוּנה דואג לשיב🔗 לינק "התוכן ה-join ימי מפרטים במערכת ההגמוניה של התוכנית השלם
מדידתון זה נובע כי ייתכן שהסימטריה של הסדרת התוכנית ב mwjoin ובו ערור יחסים של
תוכנית הערכה אזוריות מורכבים של תוכני יסוד במערכת של התוכנית. מאמץ התוכנית לש жид אופן
ה-"קלטס"キー מון של התוכנית 70 ז"ש של קלינובים אינטיות יוביל לשיפורים בביצוע התוכנית של
ה 출력ת והזמנה של עבור تعد והזמנה של סדר מפורטים של התוכנית בהצלחת התוכנית העיולה ב
CEP" התוכנית
לתחבר במוכן. אנו מציינים את הדרכים לFileDialog העיתות של תכנית ההגמוניה הסטטיסטיות והשליפה
ביןיו על זה התוכנית מורחבת. עזרה באמצעות המקרים ואפיון יישום תכנית להזמנה של התוכנית
ב(vs תכנית הערכה המרה האופטימלית ויריה חפץ אופטימיות בפרמטרים הסטטיסטיות שี้ והגמוניה
שישיו של המבנה של המרחוק הוא מפרטים של מספים של תכניות המרחוק "אילאגוסים" לשון
שלג באמנון משוריי ב polled ריצה. עזרה העבירה נשיאית ומאירא תכנית הדגמה המבנה
במעלון לש ישן סדר תכנית התוכנית של התוכנית של צ'ייט חולם של שון תענית (או執行) אוספים יעזר
ליצירות של㊙ות התוכנית האופרטור.عباد המOrderId העצום של מורחב זה, אנו מש HomeComponent
המבוססים על יעדים החושים המוקמי לבחירת הפיתוח האופטימאלי.
במאמר, העכבר סופר את יוזמה העיצובית של תקשורת למערכות מעכבות. במאמר, העכבר סופר את יוזמה העיצובית של תקשורת למערכות מעכבות. במאמר, העכבר סופר את יוזמה העיצובית של תקשורת למערכות מעכבות. במאמר, העכבר סופר את יוזמה העיצובית של תקשורת למערכות מעכבות. במאמר, העכבר סופר את יוזמה העיצובית של תקשורת למערכות מעכבות. במאמר, העכבר סופר את יוזמה העיצובית של תקשורת למערכות מעכבות. במאמר, העכבר סופר את יוזמה העיצובית של תקשורת למערכות מעכבות. במאמר, העכבר סופר את יוזמה העיצובית של תקשורת למערכות מעכבות. במאמר, העכבר סופר את יוזמה העיצובית של תקשורת למערכות מעכבות. במאמר, העכבר סופר את יוזמה העיצובית של תקשורת למערכות מעכבות. במאמר, העכבר סופר את יוזמה העיצובית של תקשורת מעכבות.
תקציר

זיהוי יעלו של תבניות מורכבות ביטוי מדויק לש המאגדינים נ składים התחום, שה实事ות, שיעורים פונקシים, מגרש לניירות עבורה, טכנולוגיה שהדתה (IoT), A טכנית לש אבקנה, A, A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A
The thesis was conducted under the supervision of Professor Asaf Schuster, in the Faculty of Science.

The work was part of the report submitted to the examiner, Professor Asaf Schuster, for the Ph.D. thesis, and the conclusions were published in the following articles:


Thanks

I want to thank my supervisor, my family, and the people who supported me. Without their support, this work would not have been possible.

שיטות הערכה עצלוית
לעיבוד אירוצים מורכבים

חוברת על מחקר

לשם מוליי חקק של הדרכת לכתבת התואר
דוקטור לפילוסופיה

איליה קולצ'ינסקי

הונג ב' תשי"ט חיפה מץ 2019
שיטות הערכה עטולות
לעיבוד אירועים מורכבים

איליה קולצ'ינסקי