Enhancing Stream Processing and Complex Event Processing Systems

Abderrahmen Kammoun

To cite this version:

Abderrahmen Kammoun. Enhancing Stream Processing and Complex Event Processing Systems. Networking and Internet Architecture [cs.NI]. Université de Lyon, 2019. English. NNT : 2019LYSES012 . tel-02468246

HAL Id: tel-02468246
https://theses.hal.science/tel-02468246v1

Submitted on 5 Feb 2020

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

No d'ordre NNT: 2019LYSE012

THESE DE DOCTORAT DE L'UNIVERSITE DE LYON

opérée au sein de

L'UNIVERSITE JEAN MONNET

Ecole Doctoral N° 488

Sciences, Ingénierie et Santé

Spécialité de doctorat:

Discipline : Informatique

Soutenue publiquement le 08/07/2019, par:

Abderrahmen Kammoun

Enhancing Stream Processing and Complex Event Processing Systems

---

Devant le jury composé de :

Jean-Marc Petit, PR, *Institut National des Sciences Appliquées de Lyon*, **Rapporteur**
Yann Busnel, PR, *École Nationale Supérieure Mines-Télécom Atlantique Bretagne Pays de la Loire*, **Rapporteur**
Frédérique Laforest, PR, *Institut National des Sciences Appliquées de Lyon*, **Examinatrice**
Jacques Fayolle, PR, *Université Jean Monnet*, **Directeur de thèse**
Kamal Singh, MCF, *Université Jean Monnet*, **Co-Directeur**

Je dédie cette thèse :
À mon père Habib Kammoun. Merci pour tous les sacrifices consentis pour moi.
c'est grâce à toi que je suis ce que je suis.
À la mémoire de ma mère Chédia Kammoun et mon grand-père Hedy Kammoun,
qui seront contents d'apprendre que leur fils a enfin terminé ces études. que leurs
âmes reposent en paix.
Aux familles KAMMOUN, BOUGHARIOU et GARGOURI.

Acknowledgements

Il me sera très difficile de remercier tout le monde car c'est à l'aide de nombreuses personnes que j'ai pu mener cette thèse à son terme.

Je voudrais tout d'abord remercier mes encadrants, Jacques Fayolle et Kamal Singh, pour leur aide et leurs conseils tout au long de ce travail de recherche.

Je remercie également Jean-Marc Petit et Yann Busnel d'avoir accepté de relire cette thèse et d'en être les rapporteurs

Je tiens à remercier Frédérique Laforest pour avoir accepté de participer à mon jury de thèse et pour sa participation scientifique ainsi que le temps qu'elle a consacré à ma recherche.

Je remercie toutes personnes avec qui j'ai partagé mes études, notamment pendant ces années de thèse.

Je tiens à remercier particulièrement Syed Gillani pour toutes nos discussions et ses conseils qui m'ont accompagné tout au long des recherches et de la rédaction de cette thèse. Il m'est impossible d'oublier Christophe Gravier, Julien Subercaze et Tanguy Raynaud que j'ai régulièrement côtoyées pendant ces années, pour leurs conseils précieux et leur aide.

Mer derniers remerciements vont à ma famille: Soutouty, Chouchou, Mi7ou, Kouka et Fattouma, qui a tout fait pour m'aider, qui m'a soutenu et surtout supporté pendant les moments les plus difficiles.

Abstract

As more and more connected objects and sensory devices are becoming part of our daily lives, the sea of high-velocity information flow is growing. This massive amount of data produced at high rates requires rapid insight to be useful in various applications such as the Internet of Things, health care, energy management, etc. Traditional data storage and processing techniques are proven inefficient. This gives rise to Data Stream Management (DSMS) and Complex Event Processing (CEP) systems, where the former employs stateless query operators while the later uses expressive stateful operators to extract contextual information.

Over the years, a large number of general purpose DSMS and CEP systems have been developed. However,(i) they do not support complex queries that push the CPU and memory consumption towards exponential complexity, and (ii) they do not provide a proactive or predictive view of the continuous queries. Both of these properties are of utmost importance for emerging large-scale applications.

This thesis aims to provide optimal solutions for such complex and proactive queries. Our proposed techniques, in addition to CPU and memory efficiency, enhance the capabilities of existing CEP systems by adding predictive feature through real-time learning. The main contributions of this thesis are as follows:

- We proposed various techniques to reduce the CPU and memory requirements of expensive queries with operators such as Kleene+ and Skip-Till-Any. These operators result in exponential complexity both in terms of CPU and memory. Our proposed recomputation and heuristic-based algorithm reduce the costs of these operators. These optimizations are based on enabling efficient multidimensional indexing using space-filling curves and by clustering events into batches to reduce the cost of pair-wise joins.

- We designed a novel predictive CEP system that employs historical information to predict future complex events. To efficiently employ historical information, we employ an N-dimensional historical matched sequence space. Hence, prediction can be performed by answering the range queries over the historical sequence space. We transform N-dimensional space into 1-dimension using space filling Z-order curves, enabling us to exploit 1-dimensional range search algorithms. We propose a compressed index structure, range query processing techniques and an approximate summarizing technique over the historical space.

- To showcase the value and flexibility of our proposed techniques, we employ them over multiple real-world challenges organized by the DEBS conference. These include designing a scalable system for the Top-k operator over non-linear sliding windows and a scalable framework for accelerating situation prediction over spatiotemporal event streams.

The applicability of our techniques over the real-world problems presented has produced further customize-able solutions that demonstrate the viability of our proposed methods.

Résumer

Alors que de plus en plus d'objects et d'appareils sensoriels connectés font partie de notre vie quotidienne, la masse d'information circulante à grande vitesse ne cesse d'augmenter. Cette énorme quantité de données produites à des débits élevés exige une compréhension rapide pour être utile dans divers domaines d'activité telles que l'internet des objets, la santé, la gestion de l'énergie, etc. Les techniques traditionnelles de stockage et de traitement de données se sont révélées inefficaces ou inadaptables pour gérer ce flux de données.

Cette thèse a pour objectif de proposer des solutions optimales à deux problèmes de recherche sur la gestion de flux de données. La première concerne l'optimisation de la résolution de requêtes continues complexes dans les systèmes de détection d'évènements complexe (CEP). La deuxième aux problèmes liées à la prédiction des événements complexes fondée sur l'apprentissage de l'historique du système.

Premièrement, nous avons proposé un modèle de recalcule pour le traitement de requêtes complexes, basé sur une indexation multidimensionnelle et des algorithmes de jointures optimisés. Deuxièmement, nous avons conçu un CEP prédictive qui utilise des informations historiques pour prédire des événements complexes futurs. Pour utiliser efficacement l'information historique, nous utilisons un espace de séquences historiques à N dimensions. Par conséquence, la prédiction peut être effectuée en répondant aux requêtes d'intervalles sur cet espace de séquences.

La pertinence des résultats obtenus, notamment par l'application de nos algorithmes et approches lors de challenges internationaux démontre la viabilité des méthodes que nous proposons.

Contents

List of Figures xiii

List of Tables xv

I Introduction & Background 1

1 Introduction 3

1.1 Motivation 3
1.2 Research Challenges and Contributions 5
1.2.1 Optimizing Expensive Queries for Complex Event Processing 5
1.2.2 Predictive complex event processing 7
1.2.3 Applying and testing our event processing solutions: challenging queries from the research community 8
1.3 Structure 9
1.4 List of publications 9

2 Background 11

2.1 Introduction 12
2.2 Stream processing 14
2.2.1 Active Database 14
2.2.2 Data Stream Management System 15
2.3 Complex event processing 16
2.3.1 Complex event processing Architectures 17
2.3.2 Event Processing models and definitions 19
2.3.3 Selection Policy and Query Operator 21
2.3.4 Event Detection models 24
2.4 Query Optimization in CEP Systems 29
2.4.1 Optimizations according to predicates 30
2.4.2 Optimization of query plan generation 31
2.4.3 Optimization of memory 34
2.5 Predictive Analytics & Complex Event Processing 35
2.5.1 Predictive Analytics for optimal decision making 37

2.5.2 Predictive Analytics for automatic rules generation 37
2.5.3 Predictive Analytics for complex events prediction 38
2.6 Processing Top-k queries 39
2.6.1 Definitions 39
2.6.2 Top-K Algorithms 39
2.7 Conclusion 41

II Enhancing Complex Event Processing 43

3 Enabling Efficient Recomputation Based Complex Event Processing for Expensive Queries 45
3.1 Introduction 46
3.2 Motivating examples 46
3.3 Related Works 48
3.4 Preliminaries and definitions 51
3.4.1 Definitions 51
3.4.2 Query Tree 53
3.5 Baseline Algorithm 55
3.6 RCEP: Recomputational based CEP 57
3.6.1 The Event Tree 57
3.6.2 Creating the Complex Matches 59
3.7 RCEP: General Recomputation Model 65
3.7.1 Multidimensional Events 65
3.7.2 Multidimensional Event Tree 66
3.7.3 Joins between Z-addresses 68
3.7.4 Handling Sliding Windows 72
3.7.5 Optimising Z-address Comparison 74
3.8 Experimental Evaluation 74
3.8.1 Setup and Methodology 74
3.8.2 Performance of Indexing and Join Algorithms 76
3.8.3 Performance of Sliding Windows 78
3.8.4 CEP Systems' Comparison 79
3.9 Conclusion 83

4 A Generic Framework for Predictive Complex Event Processing using Historical Sequence Space 85
4.1 Introduction 85
4.2 Contribution 87
4.3 Our Approach 88

4.3.1 Querying Historical Space for Prediction 89
4.3.2 Summarisation of Historical Space Points 91
4.4 Implementation 93
4.4.1 System Architecture 93
4.4.2 User Interface 93
4.5 Experimental Evaluation 94
4.5.1 Experiment Setup 94
4.5.2 Datasets and CEP Queries 96
4.5.3 Accuracy Metrics 96
4.5.4 Precision of Prediction with Summarisation 96
4.5.5 Comparison with other Techniques: 96
4.6 Conclusion 97

III Real World Event Processing Challenges 99

5 High Performance Top-K Processing of Non-Linear Windows over Data Streams 103
5.1 Introduction 103
5.2 Input Data Streams and Query definitions 105
5.2.1 Input Data Streams 105
5.2.2 Query definitions 105
5.3 Architecture 106
5.4 Query 1 Solution 107
5.4.1 Data structures 108
5.4.2 Algorithms 110
5.5 Query 2 Solution 112
5.5.1 Data Structures 112
5.5.2 Algorithms 115
5.6 Evaluation 117
5.6.1 Experimental Settings 118
5.6.2 Queues implementation 119
5.6.3 Analysis of Query 1 Performance 119
5.6.4 Analysis of Query 2 Performance 121
5.7 Conclusion 122

6 A Scalable Framework for Accelerating Situation Prediction over Spatio-temporal Event Streams **123**
6.1 Introduction 123
6.2 Input Data Streams and Query definitions 125
6.2.1 Input Data Streams 125
6.2.2 Query 1: Predicting destinations of vessels 126
6.2.3 Query 2: Predicting arrival times of vessels 126
6.3 Preliminaires 127
6.4 The Framework 127
6.5 Experimental Evaluation 129
6.5.1 Evaluation [Gul+18b] 129
6.5.2 Results and GC Benchmark 130
6.6 Conclusion 131

IV Conclusion **133**

7 Conclusion and Future Works **135**
7.1 Enhancing CEP performance 135
7.2 Predictive CEP 136
7.3 Real world use cases and challenges 137

Appendices

A `Upsortable` an Annotation-Based Approach **141**
A.1 Introduction 141
A.2 The Case For `Upsortable` 142
A.3 Upsortable solution 143
A.3.1 AST modifications 144
A.3.2 Bookkeeping 144
A.3.3 Garbage Collection 146
A.4 Discussion 147

B DETAILED ANALYSIS **149**
B.1 Evaluating Kleene+ Operator 149
B.2 Proof Sketches 150
B.3 Optimising Z-address Comparison 153
B.4 Operations over Virtual Z-addresses 154
B.4.1 Correctness of Comparing Virtual Z-addresses 154
B.4.2 NextJumpIn and NextJumpOut Computation 155

Works Cited **157**

List of Figures

2.1 A high-level view of a CEP system [CM12b] 17
2.2 Typical Complex Event Processing Architecture 18
2.3 Sliding Window and Tumbling Window 21
2.4 Results found based on the selection strategies 25
2.5 Non-deterministic Finite Automaton example from [ZDI14a] 26
2.6 Finite State Machine example from [AÇT08] 27
2.7 A left-deep tree plan for Stock Trades Query [MM09] 28
2.8 Sequence Scan and Construction Optimazation [WDR06] 31
2.9 Postponing with early filters Optimazation [ZDI14b] 32
2.10 shared versioned buffer Optimization [Agr+08] 35

3.1 (a) Events stream S_1 for the Stock Market, (b) matches for Query 1 over S_2, (c) Events stream S_2 of a patient's heart beat, (d) matches for Query 3 over S_2 and three windows. 49
3.2 (a) Left-deep and (b) Right-deep Query tree for Query 1 in Example 1 53
3.3 An Event Tree for the stock market events (price and timestamps) within a window ω 58
3.4 Hybrid-Join execution between two events sequences $\vec{E}_i$ and $\vec{E}_j$ . . 62
3.5 Execution of the Kleene+ operator using the Banker's sequence and generated binary numbers 64
3.6 (a) Two-dimensional Z-order curve, (b) Event Tree indexing of Z-order Curve 66
3.7 The dominance test between different Z-regions 69
3.8 Insertion and Query Time: Comparative Analysis of Event Tree and RTree 77
3.9 Analysis of Join Algorithms over different Data Distributions 78
3.10 Sliding Window Comparison 79
3.11 Memory Performance. 80
3.12 Throughput Performance. 81
3.13 Average Detetion Latency 82

4.1 Z-value encoding and calculation of g_b^s over a point in $\mathcal{H}_{space}$ (left) and the range query point (right) with two known dimensions. . . . 90

4.2 Range Query Processing with Summarisation 91
4.3 System Architecture of Pi-CEP 94
4.4 Interface of Pi-CEP 95
4.5 (a) Credit Card Dataset (b) Activity Dataset: Accuracy comparison of prediction for the number of matched sequence; 97
4.6 (a) Accuracy comparison of prediction for the number of matched 1-dimensional sequences (Credit Card Dataset); (b) Execution time in seconds for the insertion and prediction 97

5.1 Abstract Architecture of the System 107
5.2 Computational complexity for the different operations on the various data structures of Query 1. (DESC: Descendent) 108
5.3 Storage of the comments related to a post. The offset gives the index of the first valid comment. 110
5.4 A sample array of sorted dense comment ids for a user. On the left is the state of the array before an insertion. On the right, is the new state of the array after an insertion occurs at a timestamp for which the older comment in the window has a dense id value of 9. 115
5.5 Buffered Linked Blocking Queue. 120
5.6 Linked Blocking Queue. 120
5.7 Lock Free Blocking Queue. 121

6.1 labelInTOC 126
6.2 The use of thescore and historical index to predict the destination of a vessel 128
6.3 System Design for predicting vessels' destinations and arrival time . 128
6.4 Comparing average earliness rate and general accuracy with the percentage of training dataset used 131

A.1 labelInTOC 143
A.2 labelInTOC 145

B.1 Mapping of the Banker's sequences and a bitvector 150

List of Tables

2.1 CEP Optimisation Strategies & Detection Models 36

3.1 Definition of notations 54

5.1 Main data structure used in Query 2 113

5.2 Performance of our solution on the large dataset for Query 2 only provided by the organizers with respect to variation of window size (d) in minutes, number of elements in top-k, and different event passing queues (LinkedBlockingQueue, BufferedLinkedBlockingQueue and LockFreeQueue). Throughput values (T column) are expressed in kilo-events per seconds, latencies (L column) in 10^{-4} seconds, and execution time (time column) in miliseconds. 118

6.1 DEBS GC Results from Hobbit platform (ns: nano seconds) 131

B.1 Banker Sequence for an events sequence $\langle b_1, b_2, b_3 \rangle$ 150

Part I

Introduction & Background

1
Introduction

Contents

1.1 Motivation **3**
1.2 Research Challenges and Contributions **5**
1.2.1 Optimizing Expensive Queries for Complex Event Processing 5
1.2.2 Predictive complex event processing 7
1.2.3 Applying and testing our event processing solutions: challenging queries from the research community 8
1.3 Structure **9**
1.4 List of publications **9**

1.1 Motivation

In recent years, we have seen an unprecedented increase in continuously collecting data from social networks, telecommunication networks, the stock market, sensor networks, etc. The proliferation of data collected from these domains paves the way towards building various smart services. These emerging smart services will be enablers of the integration of the physical and the digital world. This will help us to make more informative decisions by analyzing large volumes of data streams.

The integration of smart services provides tangible value to the day-to-day lives of people by providing detailed and real-time observations of the physical world. However, it also leads to various novel and challenging research problems. One of such problems is the management of data with high volume and velocity. Existing Database Management systems (DBMS) do not provide an efficient solution to this

problem since such solutions assume that data is static and rarely updated. From a DBMS point of view: (i) the data processing stage comes after storing and indexing data, (ii) data querying is initiated each time by the user, i.e., in a pull manner.

In order to address these shortcomings, Data Stream Management Systems (DSMSs) were introduced. Their primary goal is to process arriving data items continuously by employing the main memory and running continuous queries. The differentiating characteristic of this system class is the notion of data itself. Data is assumed to be a continuous and possibly infinite resource, instead of resembling static facts organized and collected together for future pull-based queries. DSMSs have been an active research field for decades and a large number of highly-scalable solutions have been proposed. However, DSMSs do not extract the situational context from the processed data since they were initially considered for monitoring applications: the main aim of DSMSs is to process query operators originating from SQL such as selection, joins, etc. They leave the heavy task of extracting contextual information to the end users. Conversely, in today's increasingly connected world, businesses require expressive tools to intelligently exploit highly dynamic data with various different contexts.

The shortcomings of DSMS leads to a new class of systems referred as Complex Event Processing (CEP). These systems are based on the notion of events, i.e., each incoming data item represent the happening of an event. These events depict progressively minor changes about situations that when combined with others can be turned into meaningful contextual knowledge. CEP aims to detect complex events from incoming primitive events based on temporal and stateful operators. The examples include temporal pattern matching of credit-card transaction streams to detect fraudulent transactions [AAP17; Art+17a]; detecting Cardiac Arrhythmia disease by analyzing heart beats [Pop+17]; monitoring Hadoop cluster logs to detect cluster bottlenecks and unbalanced load distributions [Agr+08], etc.

In the last decade or so, a large number of industrial and academic CEP systems have been proposed [WDR06; ZDI14b; MZZ12; Bre+07; MM09; ESP; Ani+12; Agr+08; RLR16]. Due to the complexity of the patterns to be matched, the aim of these systems is to reduce the time and space complexity over high volume and velocity event streams. Since the complexity of CEP systems can become exponential in terms of time and space, most of the provided solutions only target the simplest query patterns. The existing optimization of these systems do not handle the complex query patterns that are frequent in the aforementioned domains well. For example, the space and time complexity of query execution for detecting Cardiac Arrhythmia disease is exponential in nature. This leads to various challenges to be addressed. Another functionality, which is missing in these existing solutions, is the predictive or proactive matching of complex events. That is, these system

are not able to predict if a complex pattern can happen in the future based on historical behaviour: such advanced analysis is at the core of many smart services and can provide a reactive insight to the problem at hand.

1.2 Research Challenges and Contributions

This section summarizes the research challenges and contributions of this thesis. The main questions focus on whether CEP systems can handle higher amounts of data for detecting complex patterns and if such systems can integrate more intelligent predictive functionality. Based on the aforementioned shortcomings of existing CEP systems, this thesis is focused on two main research challenges: high throughput CEP for detecting complex patterns and predictive CEP. Finally, we adapted and tested our algorithms on real-world problems and challenges proposed in international conferences. This, in turn, generated ideas which helped us to refine our algorithms further. We provide an overview of our contributions in the following text.

1.2.1 Optimizing Expensive Queries for Complex Event Processing

Complex Event Processing (CEP) systems incrementally extract complex temporal matches from an incoming event stream using a CEP query pattern. The general idea of these systems is to incrementally produce query matches while first constructing partial matches with the arrival of events. Hence, when a new event arrives, the system (i) creates a partial match and (ii) checks with all existing partial matches if a full match can be constructed. For example, let us consider an application that monitors the incoming event stream of temperature and energy consumption of server racks in a datacenter to detect anomalies. Let us assume the pattern which we want to detect is as follows: two temperature events that exceed a certain predefined threshold plus an increase in energy consumption. To detect that existing CEP systems create a set of partial matches to store all high temperature events that can later be matched with an energy consumption event when it arrives to complete the match. The number of partial matches grow over time as a function of query expressive and complexity. The query complexity depends on the number of attributes and relations defined between these attributes. This renders the management of partial matches as costly in terms of memory and computation resources.

Challenge: *The CEP engines show poor performance in terms of CPU and memory consumption when executing expressive queries, referred to as expensive*

queries. Thus, the needs are as follows: How to design efficient algorithms for efficient CPU and memory utilisation while processing expensive queries?

Traditional approaches compress partial matches to minimize redundancies and commonalities between partial matches. First, the system tracks the commonalities between partial matches and compresses them using an additional data structure. Later, it constructs complete matches while decompressing the set of common partial matches. This can reduce memory consumption at the added cost of the compression or decompression operations and redundant computation of partial matches. Thus, existing CEP solutions fail to scale over real-world datasets due to their space and time complexity.

In this thesis, we advocate a new type of model for CEP where instead of storing the partial matches, we just store the events. Then, we compute matches only when there is a possibility of finding a complete match. A system based on such techniques reduces the memory and CPU complexity. This is because, contrary to the incremental model, we do not have to keep all possible combinations of the matches and only useful matches are produced. However, to materialise these points, we require efficient indexing and querying techniques. Hence, our journey to provide efficient techniques led us to explore various diverse fields such as theta-joins (which allow the comparison of relationships such as $\leq$ and $\geq$), multidiemsnional indexing and range query processing. Our provided techniques are generic in nature and can be employed in general streaming-based systems.

Our key contributions are as follows:

- We provide a novel recomputation model for processing expressive CEP queries. Our algorithms reduce space and time complexity for expensive queries.
- To efficiently employ a recomputation model, we propose a heuristic algorithm to produce final matches from events in a batch mode.
- We provide multiple optimisation techniques for storing events using space-filling curves enabling us to use efficient multidimensional indexing in streaming environments.
- We experimentally demonstrate the performance of our approach against state-of-the-art solutions in terms of memory and CPU costs under heavy workloads.

1.2.2 Predictive complex event processing

Analytic systems are evolving towards proactive systems, where machines will simply do the work and leave strategy and vision to us. Proactive systems need predictive capabilities, but existing CEP systems lack them. Predictive Complex Event Processing (CEP) could become the next generation of CEP systems and it could provide future complex events. That is, given a partial match, predictive CEP should be able to provide future possible matches. The predictive CEP problem resembles that of sequence pattern mining and prediction. However, on the shelf data mining algorithms are use case specific. Moreover, they also require considerable efforts to model each dataset. This is unlike CEP which are supposed to be generic and can extract matches from any type of dataset.

Challenge: *The questions are as follows: how to add predictive capabilities to CEP systems, how to design algorithms to handle complex multi-dimensional sequences, how to manage such complex data structures in a dynamic environment and how to ensure accuracy and efficiency?*

This thesis designs a novel predictive CEP system and shows that this problem can be solved while leveraging existing data modelling, query execution and optimisation frameworks. We model the predictive detection of events over an N-dimensional historical matched sequence space. Hence, a predictive set of events can be determined by answering the range queries over the historical sequence space. We design range query processing techniques and an approximate summarisation technique over the historical space.

Our design is based on two ideas:

- History can be a mirror of future and, thus, we can use historical matches and events to predict future matches.
- Simply storing historical events and matches while operating in the main memory is not scalable and therefore not possible for data stream settings. On the other hand, discarding historical points may degrade prediction accuracy. Thus, we summarised the older matches according to their importance as will be defined in the later chapters.

Our experimental evaluation over two real-world datasets shows the significance of our indexing, querying and summarising techniques for prediction. Our system outperforms competitors by a considerable margin in terms of prediction accuracy and performance.

1.2.3 Applying and testing our event processing solutions: challenging queries from the research community

The research community has proposed a series of challenges to test stream event processing techniques on real use cases. Such challenges allow researchers to demonstrate the performance of their event processing systems and algorithms on real datasets. Each year the tasks evolve considerably. They allow us to test and apply some of our systems and algorithms to real data and use-cases.

Challenge: *Are our algorithms generic enough to solve varying and challenging real tasks, can our systems be adapted to different specific tasks and how do we fair as compared to other competitive researchers, their systems and algorithms?*

We participated in the challenges proposed by the Distributed and Event-Based System (DEBS) community. The first challenge that we tackled was proposed in 2016. The DEBS Grand Challenge (GC) 2016 [Gul+16b] focused on processing social networking data in real time. The aim was to design efficient top-k algorithms for analysing social network data consisting of live content, evolving users and their interactions.

While designing our solution, we carefully paid attention on optimizing every part of the system. Fast data parsing, efficient message-passing queues, as well as devising efficient fast lower and upper bounds to avoid costly computation, were key to the success of our approach. Similarly, the choice and design behind the most common data structures largely contributed to overall system performance. In addition, we developed ***Upsortable***, a data structure that offers developers a safe and time-efficient solution for developing top-k queries on data streams while maintaining full compatibility with standard Java[Sub+17].

Another challenge in which we participated was the DEBS GC 2018 [Gul+18a]. The DEBS community was interested in processing spatio-temporal streaming data. The goal was to predict the destination of a vessel and its arrival time. We provided a novel view of using historical data for the prediction problem in the streaming environment. Our solution was based on predictive CEP system, efficient indexing, querying and summarising techniques in a streaming environment.

These two challenges had an important impact on the work in this thesis. The first challenge gave us the idea of using a lazy approach for efficient processing of event streams. The idea of using lazy approach was carried forward in the recomputation based approach and it will be seen in Chapter 3. The second challenge helped us to further enhance our predictive CEP solution described in Chapter 4.

1.3 Structure

The remainder of this thesis is organized as follows. In Chapter 2, we introduce the necessary background on stream processing and complex event processing. In Chapter 3, we present the techniques used to optimize expensive queries in complex event processing. In Chapter 4, we present the design of our predictive CEP system and show that this problem can be solved while leveraging existing data modelling, query execution and optimisation frameworks. In Chapter 5 and 6, we demonstrate that the proposed techniques, along with some additional ones, can be applied to solve different challenges. We conclude in Chapter 7 and discuss future work.

1.4 List of publications

Parts of the work presented herein have been published in various international workshops and conferences. We briefly introduce them as follows:

- At EDBT 2018, we presented a comparison of different event detection models and concluded that traditional approaches, based on partial matches' storage, are inefficient for expressive queries. We advised a simple yet efficient approach that experimentally outperforms traditional approaches on both CPU and memory usage.
- At ICDM 2017 Workshops, we presented the design of our novel predictive CEP system and showed that this problem can be solved while leveraging existing data modelling, query execution and optimisation frameworks.
- We have also participated in different challenges in which we were inspired from our proposed techniques to find solutions. Highlights include a paper presented at ACM DEBS, providing a scalable system for top-k operator over non-linear sliding windows using incremental indexing for relational data streams. Furthermore, another paper proposed a Scalable Framework for Accelerating Situation Prediction over Spatio-temporal Event Streams. Our solution provides a novel view of the prediction problem in streaming settings. Hence, the prediction is not just based on recent data, but on the whole useful historical dataset.
- Moreover, much of the work will be submitted for review for the VLDB conference, where we advocate a new type of model for CEP wherein we reduce the memory and CPU cost, by providing efficient join techniques, multidimensional indexing and range query processing.

Below is the complete list of related publications:

- Abderrahmen Kammoun, Tanguy Raynaud, Syed Gillani, Kamal Singh, Jacques Fayolle, Frédérique Laforest: A Scalable Framework for Accelerating Situation Prediction over Spatio-temporal Event Streams. DEBS 2018: 221-223

- Abderrahmen Kammoun, Syed Gillani, Julien Subercaze, Stéphane Frénot, Kamal Singh, Frédérique Laforest, Jacques Fayolle: All that Incremental is not Efficient: Towards Recomputation Based Complex Event Processing for Expensive Queries. EDBT 2018: 437-440

- Julien Subercaze, Christophe Gravier, Syed Gillani, Abderrahmen Kammoun, Frédérique Laforest: Upsortable: Programming TopK Queries Over Data Streams. PVLDB 10(12): 1873-1876 (2017)

- Syed Gillani, Abderrahmen Kammoun, Kamal Singh, Julien Subercaze, Christophe Gravier, Jacques Fayolle, Frédérique Laforest:Pi-CEP: Predictive Complex Event Processing Using Range Queries over Historical Pattern Space. ICDM Workshops 2017: 1166-1171

- Abderrahmen Kammoun, Syed Gillani, Christophe Gravier, Julien Subercaze: High performance top-k processing of non-linear windows over data streams. DEBS 2016: 293-300

2

Background

Contents

2.1 Introduction **12**
2.2 Stream processing **14**
2.2.1 Active Database 14
2.2.2 Data Stream Management System 15
2.3 Complex event processing **16**
2.3.1 Complex event processing Architectures 17
2.3.2 Event Processing models and definitions 19
2.3.3 Selection Policy and Query Operator 21
2.3.4 Event Detection models 24
2.4 Query Optimization in CEP Systems **29**
2.4.1 Optimizations according to predicates 30
2.4.2 Optimization of query plan generation 31
2.4.3 Optimization of memory 34
2.5 Predictive Analytics & Complex Event Processing . . **35**
2.5.1 Predictive Analytics for optimal decision making 37
2.5.2 Predictive Analytics for automatic rules generation . . . 37
2.5.3 Predictive Analytics for complex events prediction . . . 38
2.6 Processing Top-k queries **39**
2.6.1 Definitions 39
2.6.2 Top-K Algorithms 39
2.7 Conclusion **41**

2.1 Introduction

Our society has unequivocally entered the era of data analysis. Used exclusively in large companies as well as institutional and scientific organisations for decades, data analysis is now common in many disciplines. Moreover, new paradigms are emerging which highly benefit from data analysis such as the Internet of Things (IoT). As mentioned in the introduction to this dissertation, IoT reinforces the presence of data that changes regularly over time and must be processed continuously, due to the massive presence of sensors. To this end, it is a realistic step to consider the representation of data in the form of flows and to use processing models corresponding to such information. Moreover, according to some estimates, the flow of information through the Internet of Things will be very significant and will involve complex operations on large flows and high throughput.

In a general context, an increasing number of applications require continuous processing of streaming data sensed in different locations, at different times and rates, in order to obtain added value in their business and service domains. Real time analytics is nowadays at the centre of companies' concerns. This is an essential practice to significantly increase turnover, but also to remain competitive in most industries. Real time Analytics is the science of examining raw data in real time in order to draw conclusions from this information without delay. Analytics tools are used to enable companies and organisations to make better decisions. Analytics over big data is one of the important tasks for success in many business and service domains. Some examples of these domains include health [Blo+10], finance[Adi+06], energy [Hil+08], security and emergency response[RED12], where several big data applications in these domains rely on fast and timely analytics based on available data to make quality decisions.

Regarding IoT scenarios, you will find that it is deeply embedded in the real time analytics world.

- Businesses can send targeted incentives when prospective customers are nearby, by tracking data from their locations sensed by their mobile devices.

- Financial institutions can monitor stock market fluctuations in real time and rebalance investments based on carefully and precisely measured market valuations to the nearest second. In addition, they could set up this same capability, as an added-value service, for clients who want a more powerful way to manage their investments.

- E-commerce companies can detect fraud the moment it happens by defining patterns that might be suspicious and watching machine-driven algorithms.

From the above examples, we can glean some key benefits of real time analytics:

- It can open the way for more advanced data analysis.
- It can work in addition to machine learning techniques to provide further guidance for all kinds of companies.
- It can help companies improve their profitability, thereby reducing their costs and increasing production.
- It can support brands to provide more relevant consumer experiences, which is a key to success in the age of digital everything.
- It can provide new skills when it comes to fraud detection and management, both in and outside the financial industry.

The domain of data analysis and data management has seen significant evolution. Database management systems (DBMS) based on the OLTP (On Line Transaction Processing) model have emerged to provide optimal management of this data. The objective of OLTP is to allow insertion and modification in a fast and safe way. DBMSs are especially suited to responding to requests for a small portion of the data stored in the database. In this perspective, it is standard practice to state that database systems allow basic data processing. In contrast, this model has limitations in terms of providing an efficient and consistent timely analysis of the collected data. To this end, *Information Flow Processing* (IFP) [CM12b] paradigm has been developed through the work of several communities moving from active databases [MD89] to complex events processing [LF98]. Information Flow Processing aims to extract new knowledge as soon as new information is being reported. The idea is to move the active logic from the application layer to the database layer. This avoids redundancy of monitoring functionality in case of a distributed application and a wrong verification frequency in the case of a stand-alone monitoring application[MD89].

Real time data analysis can mean applying some simple or complex operations on data, it could also mean extracting the most significant events, etc. Simple operations can use basic stream processing techniques, whereas complex operations like detecting patterns use complex events processing techniques. For tracking most significant events, top-k monitoring systems can be used. Thus, the research aspect of Information Flow Processing may be approached from several points of view. To more suitably align the state of the art with my research goals, the following areas were explored and are thus discussed in the following text: Real-time Analytics, the Stream processing domain, Complex Event Processing, Optimization trends for CEP, Predictive CEP, and Top-k queries.

2.2 Stream processing

Stream processing is about handling continuous information as it flows to the engine of a stream processing system. In this paradigm, the user inserts a persisting query in the form of a rule in the main memory that will be executed by the engine to retrieve incrementally significant information from the stream. Indeed, the event processing system requires that information be processed, asynchronously with respect to its arrival, before it is stored and indexed if necessary. Both aspects contrast with the requirements of the traditional database management system (DBMS). DBMSs store and index data so they can be queried by users on demand. While this is useful in case of low data update frequencies, in the context of IoT real-time applications, it is not necessarily efficient. These applications require continuous and timely processing of information in order to efficiently extract new knowledge as soon as relevant information flows from the periphery to the centre of the system. For example, water leak detection applications should be capable of detecting the existence of a leak as fast as possible so timely actions can be taken. Therefore, indexing and then requesting sensor data from a database, to detect if a leak has happened or not, may not only delay the repair, but also burden the workload and complicate the tasks.

2.2.1 Active Database

The roots of Stream Processing or Event processing can be traced back to the area of Active Database Management Systems[MD89] (ADBMS). Active database management systems were developed to overcome the passive nature of the database management systems, in the sense that DBMSs are explicitly and synchronously invoked by users or application layer initiated operations. In other words, using only a DBMS, it is not possible to automatically take an action or send a notification after the occurrence of a specific situation. Active database is a DBMS endowed with an active behaviour, that has been moved from the application layer. The application layer uses polling techniques to determine changes to data that must be fine-tuned so as not to flood the DBMS with too frequent queries that mostly return the same answers or, in the case of too infrequent polling, that the application not miss important changes to data. The active behaviour of database management systems supports rules with three components, listed below:

- Event: an event is an occurence that stimulates an action. This stimulator can be an internal operator to the database, like a tuple update or insertion, or an external stimulator such as the new value of an attribute or a clock notification.

- Condition: a condition defines the context in which the event occurs, for example if the new value exceeds a pre-defined threshold.

- Action: an action is a list of tasks that should be carried out when both the stimulator events and conditions have taken place. The action can be internal to the database, such as the deletion of a tuple, or an external one, like sending an alert to an external application.

Even though a series of applications such as tracking financial transactions [CS94] and identifying unusual patterns of activity [SS04] have been made possible thanks to the presence of active database. The active database is built around persistent storage, like traditional DBMS, where irrelevant data can be kept and it can suffer from poor performance in the case of a high rate of events and a significant number of rules [Ter+92].

2.2.2 Data Stream Management System

As mentioned before, the massive presence of sensors reinforces the presence of an unbounded high rate of events which must be processed continuously. It is not optimal to load arriving data (event) into a traditional database management system (DBMS) and operate on it from there. Traditional DBMSs do not support continuous queries, and they are not designed for rapid and continuous loading of individual data items. To achieve this, the database community has developed a new class of system to handle continuous and unbounded data items in a timely way: Data Stream Management Systems (DSMSs) [BW01; Bab+02] comes with an orthogonal query processing mechanism compared to DBMS, where it processes continuously arriving data items in the main memory and persists only defined queries.

To excel at a variety of real-time stream processing applications, the DSMS brings new requirements [SÇZ05], listed below:

- To achieve low latency, DSMSs must keep data moving to avoid the expensive cost of storage before initiating the knowledge extraction process.

- For expressive processing on continuous streams of data, a high level query language expanding upon the SQL language with extended streaming operations must be supported.

- To provide resiliency against stream imperfections, frequently present in the real world data, a mechanism must be provided to handle missing and out-of-order data.

- To guarantee predictable and repeatable outcomes, meaning the system must compute the same results for equivalent streams, a deterministic processing must be maintained throughout the entire processing pipeline.

- To provide interesting features by integrating stateful operations, such as aggregates and sequences.

- To ensure high availability and data safety. Where failure or loss of information can be too costly, one must use a high-availability (HA) solutions with hot backup and real-time failover schemes. [BGH87].

- To achieve incremental scalability by distributing processing across multiple processors and machines.

- To enable a real-time response, the DSMS core employs optimized query plans and minimizes processing overheads.

The above features are required for any system that will be used for high-volume low-latency stream processing applications. We can see that most of the requirements for generic stream processors are handled by [Aba+03; ABW06; Ara+03], with a varying degree of success. In addition, it is necessary to deal with complex scenarios by integrating dynamic operators (5th requirement), which give rise to the definition of Complex Event Processing (CEP) [Luc08]. This requires better management of CPU and memory to maintain the requirements of a flow processing system.

2.3 Complex event processing

A subset of event processing applications may involve detecting complex events such as anomaly detection, pattern detection, etc. They are termed as complex event processing (CEP) systems. The incoming events have specific semantics. They describe something that happens in the real world, and CEP systems are in charge of filtering, correlating and combining them. CEP systems detect patterns and extract higher knowledge from raw events and then notify to event consumers (figure2.1). Thus, it is quite different from the data processing done by simpler event processing systems, where the operations are as light as filtering and transformation.

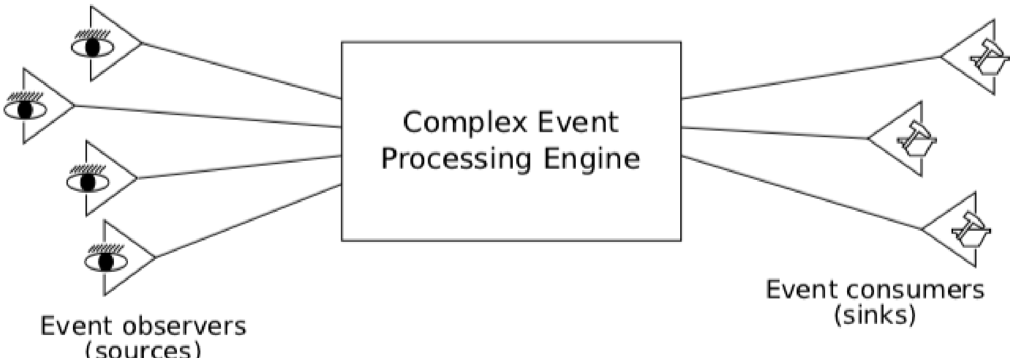


Figure 2.1: A high-level view of a CEP system [CM12b]

The high-level view of a CEP system shares some similarities with the popular publish/subscribe systems. The latter focuses on filtering incoming events from publishers and produces notifications (events) consumed by subscribers. Therefore, publishers and subscribers exchange information asynchronously, through publish/subscribe systems. To organize and present information to the subscribers, two approaches can be clearly established: Topic-based [Cas+02; Zhu+01] and content-based [CDF01; BFP10], explained below (figure2.2):

- Topic-based model: limits choice for subscribers using a list that the publishers have pre-defined. A subscriber can express its interest in one or more topics, and then they receive all published messages related to these topics.

- Content-based model: provides more flexibility to subscribers to define their event interests, by specifying constraints over the values of events' attributes. While remaining limited to accessing the history of the received events and the relationship between them.

Publish/subscribe only processes and filters each event separately from the others to decide on their importance to subscribers. CEP can be seen as a content-based publish/subscribe system endowed with a pattern based language that provides logic, sequence, window and iteration operators to capture the occurrence of high level relevant events.

2.3.1 Complex event processing Architectures

The goal of complex event processing is to extend activity monitoring through inferences from source events, and to send alerts pertaining to such inferences for further action or analysis via dashboards, in order to respond to them as quickly as

possible. CEP is an enrichment system that reports occurrences as they happen. In a service-oriented architecture, a typical CEP architecture would be as follows:

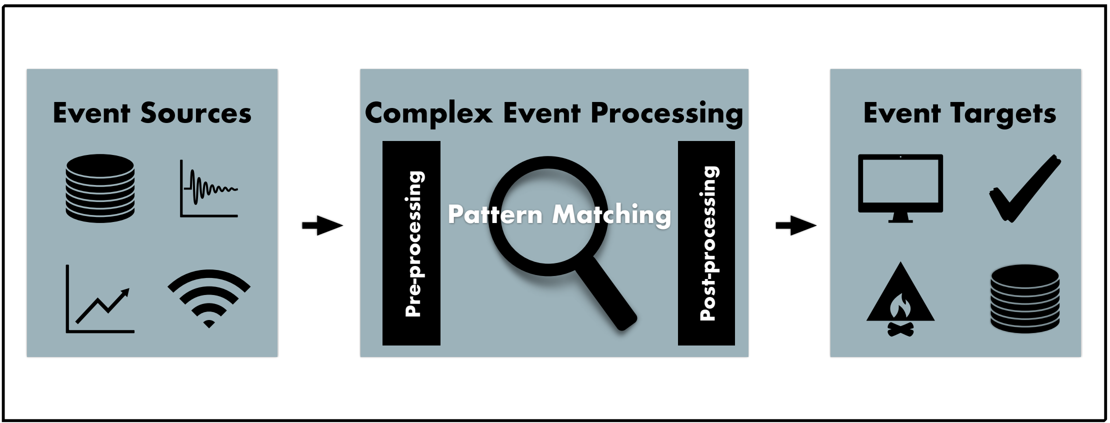


Figure 2.2: Typical Complex Event Processing Architecture

- Event sources send or produce events that describe or report everything happening in a particular situation. They act like publishers.
- Pre-processing includes converting the source-specific event format to an understandable CEP engine format.
- CEP uses events-correlation and pattern-matching techniques to "infer" complex events, which will be forwarded to event targets. Some post-processing may supsequently be applied.
- Post-processing includes converting the complex event CEP format to the target-specific format. Thus, the event could then be consumed by event targets.
- Event targets are the consumers in the form of a monitoring application or database, where the indexed results could be used for a prediction task.

To express the set of relevant events in the pattern matching step, CEP imposes a window of valid events on the incoming event stream. The shifting of a window over the event stream is defined by a *Window Policy* and the selection of relevant events in the window is determined by a *Selection Policy*. These are defined in more detail in the next sections.

2.3.2 Event Processing models and definitions

This section discusses data and processing models for CEP and illustrates their usage by referring to the SASE+ language [DIG07]. A CEP system essentially aims to detect predefined sequences of events in an infinite flow of events [Cugola and Margara, 2012b]. When a predefined sequence is detected by the CEP system, it generates a combination of events, which may trigger certain associated actions like generating an alert. The most central concept in the CEP field is therefore an event.

2.3.2.1 Event definition

An *event* is represented by a data element that encodes interesting occurrences representing a change or a condition in the domain of analysis [CM15; May18]. Events can be simple events emitted from a sensor representing low-level information, such as a temperature value update in a room, or can be a complex event emitted from a CEP system that represents high level information, such as a fire alert. Each event is characterized by *meta data* and *content*. The former is composed of a timestamp and the event type: -*timestamp* defines ordering relationships among events -*the event type* defines the abstract structure of a similar set of events, for example the type Alert for all alert occurrences. The latter contains relevant information, such as for an Alert, there are three fields: the first and second are of type *Double* to geolocate the alert (latitude and longitude), the third of type *String* to describe the Alert (Fire, Leak...).

Definition 1: Event

An *event* e is a tuple (A, t), where $A = \{A_1, A_2, \ldots, A_m\}$ $(m \geq 1)$ is a set of attributes and $t \in \mathbb{T}$ is an associated timestamp that belongs to a totally ordered set of time points $(\mathbb{T}, \leq)$.

2.3.2.2 Event Stream definition

An event stream includes a set of events. At a high level, a stream can be seen as an infinite queue of events, to which new events are continuously added.

Definition 2: Event Stream

An *event stream* $\mathcal{S}$ is a possibly infinite set of events such that for any given timestamps t and t', there is a finite amount of events occurring between them.

2.3.2.3 Event sequence definition

An Event sequence is a chronologically ordered *sequence of events*, based on timestamps given by $\mathbb{T}$, is represented as $\vec{E} = \langle e_1, e_2, \dots, e_n \rangle$ with e_1 referring to the first event and e_n to the last.

2.3.2.4 Window Policy definition

Window is a crucial concept in Data Stream Management System because an application cannot store an infinite number of events. A window is an operator that sets the validity of incoming events to the window size w. A window size can depend on the timestamp or the number of events [WDR06; CM12a; DM17]. Furthermore, it can also depend on the occurrence of a particular event or content where an event can be used to define the beginning or the end of a window, known as value-based windows [Bab+02]. Note that other types of windows are system-specific such as jumping-windows and non-linear windows. Herein, we define the most commonly used time-based *sliding window* and *tumbling window* for complex events processing.

Sliding Window

The time-based sliding window $\omega^{time} : S^l_\tau \times T \rightarrow S^l_\tau$ takes a logical stream S and the window size as arguments. The window size $w \in T, w > 0$, represents a period of time and w indicates the amount of time units captured by the window. The operator continually shifts the time interval of size w time units over its input stream to define the involved tuples. The default amount of window shift corresponds to the finest granularity of the time domain [KS09].

Definition 3: Sliding Window

$$w^{time}_w(S) := \{(e, \hat{t}, \hat{n}))| \exists X \subseteq S.X \neq 0 \wedge$$
$$X = \{\ (e, t, n) \in S \mid max\ \{\ \hat{t} - w + 1, 0\ \} \leq t \leq \hat{t}\ \} \wedge \hat{n} = \textstyle\sum_{(e,t,n) \in X} n\ \}$$

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

It's also possible that the sliding window takes an optional SLIDE clause defined by x, where $x \in \mathbb{N}^+$. The slide defines the progression step at which the window advances. In this case the window moves forward only once every x time units by an amount of x time units and the set of valid events change at time instance $x - 1, 2.x - 1, 3.x - 1$, and so on (if we assume that the window starts at the earliest time instant 0).

Definition 4: Sliding Window with Slide x

$$w^{time}_{w,x}(S) := \{(e,\hat{t},\hat{n}))|\hat{t} \geq x-1 \wedge \exists X \subseteq S.X \neq 0 \wedge X = \{\ (e,t,n) \in S \mid max\ \{\ \left\lfloor \frac{\hat{t}+1}{x} \right\rfloor\ .\ x - w + 1, 0\ \} \leq t \leq \left\lfloor \frac{\hat{t}+1}{x} \right\rfloor\ .\ x - 1\ \} \wedge \hat{n} = \sum_{(e,t,n)\in X} n\ \}$$

Tumbling Window

The Sliding window degenerates to a tumbling window, if $x = w$, in which case all the events within a window expire at the same time.

Definition 5: Tumbling Window

$$w^{time}_{w,x}(S) := \{(e,\hat{t},\hat{n}))|\hat{t} \geq x-1 \wedge \exists X \subseteq S.X \neq 0 \wedge X = \{\ (e,t,n) \in S \mid max\ \{\ (\left\lfloor \frac{\hat{t}+1}{w} \right\rfloor\ - 1)\ .w, 0\ \} \leq t \leq \left\lfloor \frac{\hat{t}+1}{w} \right\rfloor\ .\ w - 1\ \} \wedge \hat{n} = \sum_{(e,t,n)\in X} n\ \}$$

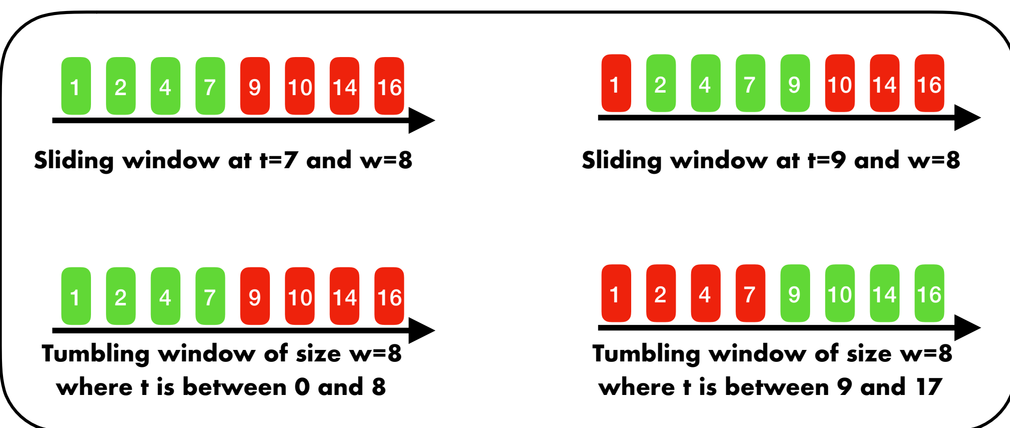


Figure 2.3: Sliding Window and Tumbling Window

Figure 2.3 shows a sliding window where the optional *SLIDE* is omitted (i.e., $x = 1$) so multiple sliding windows can share the same data items. It also shows a tumbling window where all the data items expire at the same time.

2.3.3 Selection Policy and Query Operator

Most CEP systems such as SASE [WDR06], define an Event Query Language (EQL)[Bui09] to match candidate events to their described types in the query languages. Complex query defines specific combinations of events using multiple

event-queries and the conditions to describe the correlation between them. Considering the example of water leak detection, **LowPressure(subregion(r1))**$\mapsto$ **LowConsumption(subregion(r1))** is a high-level event that may represent a leak event. That is, if the low-pressure event is followed by ($\mapsto$) low consumption in the sub-region r1 then it may point to a water leak. Typically, the expressivity of EQLs is measured by their capabilities to detect patterns. Therefore, event queries are a feature group that consists of:

- Conjunction operator: two or more events occur at the same time or during the same period.
- Disjunction operator: one of two or more events occurs without having any order constraints.
- Negation operator: the non-existence of an event.
- ANY operator: any event may occur.
- Kleene Closure ($A^{*}/A^{+}/A^{num}$): an event may occur one or more times (+) or zero or more times(*).
- Sequence Operator: a sequence of two events (SEQ(E1; E2)), means that E1 occurs before E2.

SASE [Agr+08; ZDI14a] has provided various selection strategies that overlap the sequence operator with some constraints to mix relevant and irrelevant event occurrence. In the following, the provided selection strategies are described and compared with each other in the context of the example of a sensor data stream in Figure 2.4. An event selection policy defines how a query submitted to a CEP system will specify which of the candidate events will be selected to build the corresponding complex event match based on a given pattern. Event selection policy adds a certain functionality in the detection of high-level events, rather than being limited to a regular expression's operators.

- Strict Contiguity (SC): requires the two selected events within a sequence to be contiguous in the events stream. This means that the input stream cannot contain any other events in between, similar to regular expression matching against strings.
- Partition Contiguity (PC): is a relaxation of strict contiguity strategy, to remove SC where the sequence of events is portioned based on a condition, For example when the partition condition is based on the region R1 (Region==R1), all the events that do not hold this condition will be skipped.

- Skip-Till-Next (STN): In this strategy, the two relevant events do not need to be contiguous in a relevant partition. STN is a relaxation of the strict contiguity strategy, where certain events considered as noise in the input stream are ignored.

- Skip-Till-Any (STA): is a relaxation of Skip-Till-Next to compute transitive closure over relevant events allowing non-determinism on relevant events. Such that for a sequence (E1;E2) all the patterns where E2 follows E1 are output.

Consider the Water Sensor event stream in Figure 2.4(a) that records water pressure, volume and occurrence time in different regions. For region $r1$, $e1$ represents, a *pressure* of 3 bar and a *volume* of 270 cubic metres at time 1. To analyze water sensors' data, consider the following pattern query: within a period of 10ms, find three or more events with strictly increasing volume followed by pressure of less than or equal to ($\leq$) 1 bar in the same region. Figure 2.4(b) shows five matches and one partial match for the defined query, plotted over time. Events are displayed as circles and labeled with sensor region, water volume and pressure.

Figure 2.4 (b.1) shows a Strict Contiguity Match (coloured in Red) and Partition Contiguity Match (coloured in Black). The Strict Contiguity Match consists of the events e3, e4, e5, e6 and e7. Event e3 is te start of the match, e4, e5 and e6 are subsequent events with strictly increasing volumes; and e7 is the closing event that has a pressure value equal to 1. Event e0 could have started the match if the event e1 did not occur. With the relaxation of the partition contiguity strategy, event e1 is skipped for the Partition Contiguity Match based on the regional partition condition. Thus, the Partition Contiguity Match contains all events except the second one (e2) which is a region 2 (r2) event.

Figure 2.4 (b.2) shows a Partition Contiguity Match (coloured in Black) and a Skip-Till-Next Match (coloured in Orange). The Partition Contiguity Match consists of events e11, e12, e13, e14 and e15. Events e8 and e9 could have been included in the match if event e10 did not occur. In the absence of contiguity requirements brought by the relaxation of the Partition Contiguity strategy, event e10 is considered as noise and skipped for the Skip-Till-Next Match, which will contain all the events except e10.

Figure 2.4 (b.3) shows a Skip-Till-Next partial match (coloured in Orange) and Skip-Till-Any match (coloured in Green). The Skip-Till-Next strategy finds a partial match (e16, e17, e18) that will be deleted when the window ends, but misses the e16, e17, e19, e20 match since only the partial match is considered to decide whether an event is matched or ignored as noise. In this case, e18 is a blocking noise for the Skip-Till-Next strategy. Skip-Till-Any brings more flexibility

in selecting the next relevant event by allowing a non-deterministic decision between including the event into the sequence and skipping it.

However, one important challenge is that the more flexible the selection strategy, the more complex the event processing becomes, resulting in the over-consumption of memory and CPU. Thus, we propose optimization techniques that will be discussed in the later chapters.

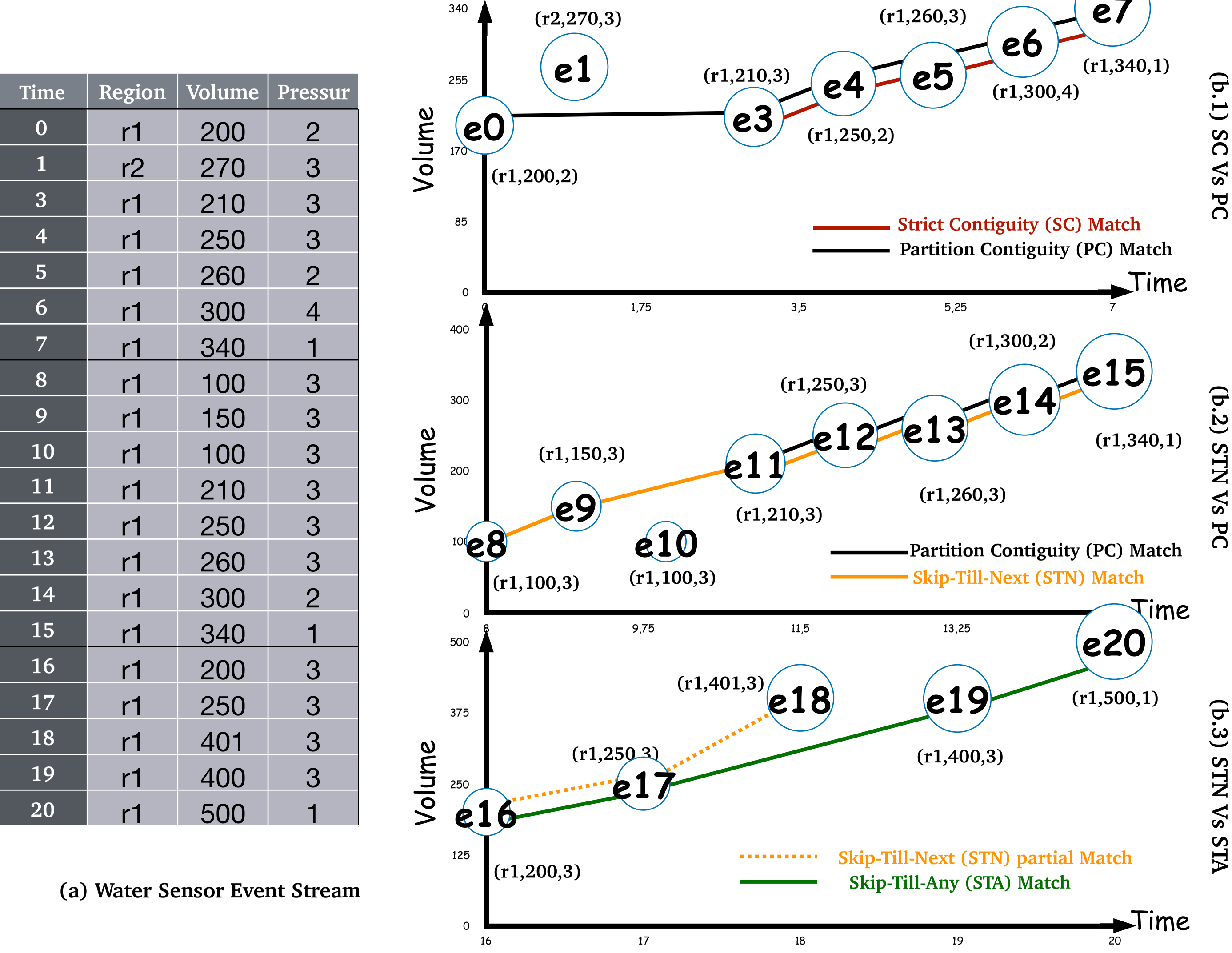
Herein, we describe the query language operators and, for brevity, we omit some operators which are not directly related to our topic such as Periodic and Aperiodic operators. In the next section we introduce the CEP query matching process.

2.3.4 Event Detection models

A large number of CEP systems, both academic and industrial, have been proposed in the literature. These systems share the goal of compiling high-level languages (Pattern P) into a logical or physical plan in some form of an automaton or a tree (reflecting the structure of P) to form the semantics and executional framework. Those plans aim to evaluate query patterns progressively by monitoring partial matches, which in turn express the potential for an imminent complex match. The choice of a plan representation is mainly motivated by the performance metrics which each approach tries to enhance. In this section, we review the main types of models used in the literature.

2.3.4.1 Automata-based Models

These systems compile CEP languages into some form of deterministic or non-deterministic automata. An Automaton consists of states set and conditional transitions between them. The transitions are triggered by the arrival of primitive events to verify conditions between events. Examples of these systems include Cayuga [Bre+07], SASE+ [Agr+08], SASE++ [ZDI14b] and Apache Flink [Fli]. With the arrival of an event, an instance or *run* of the automaton is initiated or older runs are processed. Runs that reach a final state generate the corresponding matches and are removed from the set of active runs. Since each candidate run needs to be inspected and updated for each new event, numerous systems store the set of candidate runs in a compressed form. Whenever a candidate run reaches a final state, this representation is partially decompressed to construct the output. Multiple forms of compression have been investigated in the literature. For example, SASE+ and Apache Flink factorize commonalities between runs that originate from the same ancestor run. That is, each state has a stack structure with events stored in each stack. To speed up the stack traversal, each event is augmented with a pointer to its previous event in a match to share common events. These pointers are



Time	Region	Volume	Pressur
0	r1	200	2
1	r2	270	3
3	r1	210	3
4	r1	250	3
5	r1	260	2
6	r1	300	4
7	r1	340	1
8	r1	100	3
9	r1	150	3
10	r1	100	3
11	r1	210	3
12	r1	250	3
13	r1	260	3
14	r1	300	2
15	r1	340	1
16	r1	200	3
17	r1	250	3
18	r1	401	3
19	r1	400	3
20	r1	500	1

Figure 2.4: Results found based on the selection strategies

traversed using depth-first-search (DFS) to extract all complex matches. SASE++ breaks the query evaluation into pattern matching and results construction phases and only stores so-called maximal runs from which other runs can be efficiently computed. The pattern matching process computes the main runs of an automaton with certain predicates postponed. Result construction derives all Kleene+ matches by applying the postponed predicates to remove non-viable runs. Although these compression techniques reduce the memory cost, the number of runs can still exceed memory and CPU resources for large windows and frequent prefixed events that would initiate a match. These techniques risk generating and updating many candidate runs that are afterwards discarded without generating output. This may be circumvented by delaying the evaluation of partial matches using so-called lazy automata [KSS15a]. However, this requires precomputed selectivity measures of the prefixed events. Furthermore, the cost of cloning runs on-the-fly for the partial matches without common prefixed events and repeated operations of computing events' predicates remain the same.

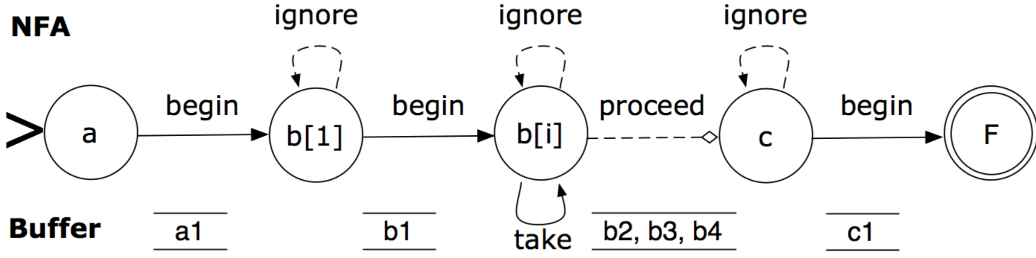


Figure 2.5: Non-deterministic Finite Automaton example from [ZDI14a]

[AÇT08] chose a simpler way to express the detection of complex events by using Finite State Machines (FSM), which have the same functionalities as NFA. However, it is a pull-based event acquisition and processing system. The authors propose that it is neither necessary nor efficient to process all generated events when only a small fraction of them make up a complex event. Event acquisition is based on a cost based plan that exploits temporal constraints among events and statistical event models, to reduce latency and communication costs. An example is illustrated in Figure 2.6 where an event detection plan is represented as Finite State Machine.

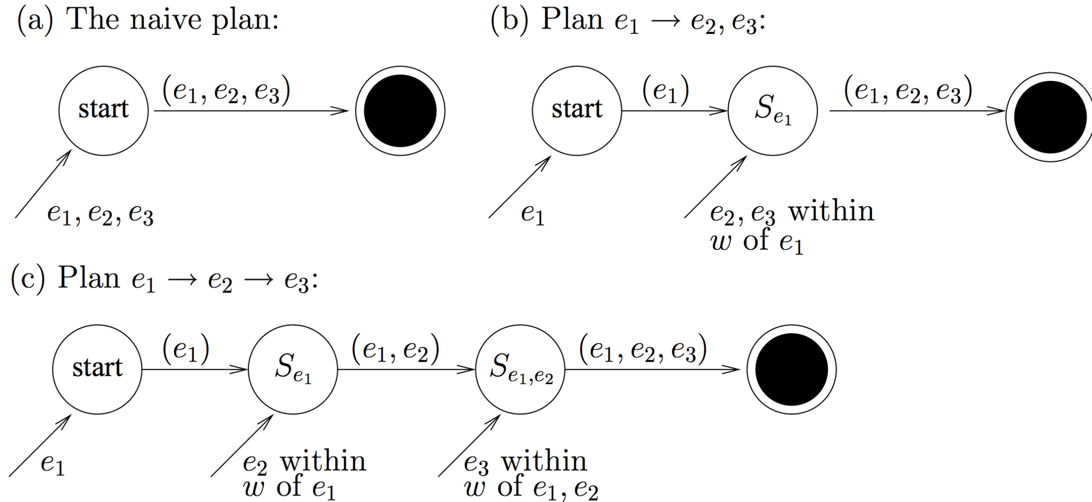


Figure 2.6: Finite State Machine example from [AÇT08]

Automata-based techniques limit the adaptation of optimization techniques provided by DSMSs, for which Tree-based models are usually used, but they also provide new approaches inspired by the field of the regular expression.

2.3.4.2 Tree-based Models

An alternative to NFA, some works have proposed a tree-based model for both query parsing and the evaluation of events. That is, the compiled tree parses the CEP query operators, and the events from the stream are added to the buffers of the operators. A realisation of such model has been presented in ZStream [MM09] and grounded in Esper [ESP], where a set of query operators forms the *left-deep* or *right-deep* tree and leaf nodes are assigned with buffers. Events from the stream are stored into the leaf nodes' buffers as they arrive, while intermediate nodes store partial matches that are assembled from the leaf nodes' or sub-tree buffers as shown in the following figure 2.7. Then, the nesting of query operators in the tree determines the executional order and matches are propagated while verifying all the query predicates at all buffers. ZStream [MM09] defines an event-based system, where events are processed as they arrive as well as batch-based systems, where the operators are executed over a set of primitive batched events. The optimisation techniques employed for this model are: cost-based joins and hash-based indexing for the equality predicates (joins) in the CEP query. Systems based on this model do not support Kleene closure computation under skip-till-any-match [MM09] and they also store partial matches in the sub-tree buffers.

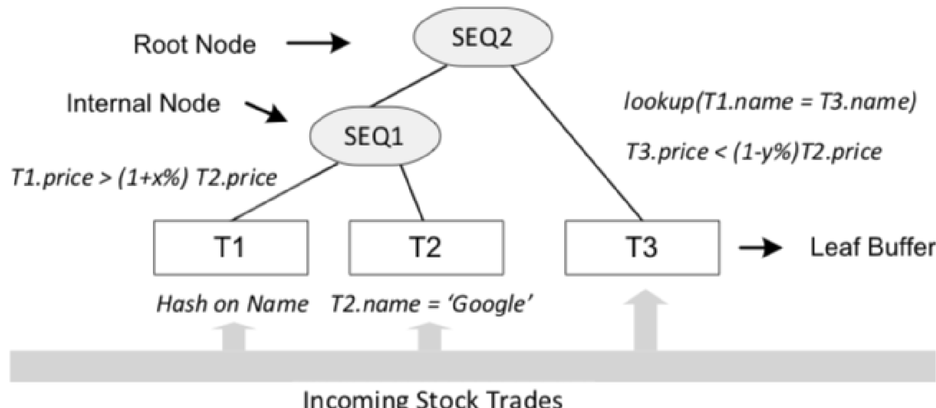


Figure 2.7: A left-deep tree plan for Stock Trades Query [MM09]

2.3.4.3 Logic-based models

Some works use logic-based models for CEP: the CEP query is translated into logic rules and is executed on a logic-based system. Notably, ETALIS [Ani+12] translates rules into Prolog rules and employs a Prolog engine for their evaluation. It does not explicitly provide any CPU or memory optimisation strategies. Instead, it shows how rules can be nested to produce complex ones. This type of system is suitable for the request-response computation, as they check if the defined pattern can be inferred or not depending on when the given request is posted. Thus, a complex event is detected only if the request is satisfied by the time it is processed. This is contrary the event processing systems, which evaluate updated knowledge for each newly arrived event. In addition, deductive systems choke under high stream rate which explains the lack of optimisation strategies.

2.3.4.4 Graph-based models

Since their modelization, graph based models have been deployed to merge the rules of logic-based models into a single graph [AÇT08; Was+12], with the aim of serving as a rule index data structure to reveal event dependencies. Hence, giving a general overview of the deployed complex patterns but not being used as a query plan for the detection process. Recently, a graph-based system was proposed to detect complex event trends (CET) [Pop+17]. It was designed to detect the longest possible patterns within a window. CET stores and reuses partial matches in a graph structure. On the expiration of a window, complete matches are constructed with a DFS. In contrast to SASE++, it stores the complete set of partial matches to avoid recomputation of common sub-patterns. Hence, it results in exponential space complexity. Furthermore, to avoid the cost of DFS for

every incoming event, the results are constructed at the end of each window. As a consequence, it is based on pull-based semantics, in contrast to the push-based semantics of general CEP systems.

2.4 Query Optimization in CEP Systems

As defined by [Hir+18], stream query optimization is the process of modifying a stream processing query, often by changing its topology and/or operators, with the aim of achieving better performance (such as higher throughput, lower latency, or reduced resource usage), while preserving the semantics of the original query. Performance is a fundamental concern for all users of computer systems. Achieving acceptable performance at a reasonable cost is an important requirement. For a computer, performance is measured by the amount of useful work accomplished in relation to the time and resources used. For the users of a computer system, performance is measured against expectations and negotiated levels of service, known as Quality-of-Service (QoS). For more information on QoS in CEP, we refer to the original paper [Pal+18]. In the context of Complex Event Processing, performance metrics measure the efficiency of a query plan execution. The efficiency of a query plan execution is reflected by how fast a CEP system processes incoming data. The principal metrics used to measure performance are *Memory* and *CPU* consumption, where the latter highly correlates with *Throughput*. Throughput is typically measured in units of events per time unit, i.e, how many events per second are processed. Memory in such systems is highly demanded in processing events and storing active partial matches for matching patterns. Regardless of the techniques used to achieve better performance, reducing active partial matches is considered as a highly relevant performance optimization goal[KS18b; ZDI14b]. Considering this, each newly arrived primitive event needs to be checked against all active partial matches, the number of which could be exponential to the number of processed events in the worst case scenario. Thus monitoring all the partial matches becomes a bottleneck in the case of all CEP-models.

Different approaches [WDR06; Agr+08; AÇT08; MM09; REG11; KSS15b; Ray+13] were proposed in the literature to cope with the volume and velocity aspects of the Big Data nature of event stream, with the aim of achieving high throughput or low CPU costs and memory consumption. Thus one can differentiate numerous aspects within the concept of optimization although they are also closely interwoven with one another.

2.4.1 Optimizations according to predicates

A simple approach to reduce the runtime cost of queries is predicate-related optimization, where three strategies stand out as explained in the following:

- Predicate pushdown is a common and important optimization technique for pushing selection as well as window predicates down a query plan, in order to apply those selections as early as possible during the evaluation process. Note that here we assume that the raw events are input at the bottom of the plan. SASE [WDR06] pushes predicate and window evaluation down to reduce the cost of event sequence scan and NFA construction. In this work, the authors define a smart data structure, called Partitioned Active instance Stack (PAIS), where sequence construction is only performed in stacks of the same partition. The crux of the decomposition into partitioned stacks, relevant to the currently examined primitive event, is to evaluate partition predicates early and thus limit the depth of sequence construction searches. While early evaluation of sliding window limits the number of potential candidate events for evaluation in the PAIS.

- With postponing using early filters, the pruning of inconsistent events is applied as a function of the predicate's type. It can be applied during edge evaluation for state transition in the NFA model or postponed to the results creation phase. In [ZDI14b] the authors focused on the Kleene+ operator as the most expressive and expensive operator. To ensure high performance on these expensive queries they use an optimization technique to prune inconsistent events based on value predicates on-the-fly or by using a postponing method. Accordingly, they break the query evaluation into two steps: pattern matching and result construction. In the former step, they evaluate incoming events without predicates for the Kleene+ operator while the evaluation of normal state transition is performed during edge evaluation, thus they minimize the number of active match instances for Kleene+ operator, as shown in figure 2.9. In the later step and based on the matches produced, they enumerate all possible combinations while applying the early postponed predicates. To this end, they define 4 categories of predicates and decide whether a predicate can be evaluated on-the-fly or postponed until the results creation phase. Despite these postponing techniques, the number of runs can still exceed memory and CPU resources for large windows and frequent prefixed events that would initiate a match: these techniques may generate and update many candidate runs that are later discarded without generating output.

- Hashing for equality predicates: A hash table is conceptually a contiguous section of memory with a number of addressable elements in which data can be quickly added, deleted and retrived. Hash tables increase memory consumption for the purpose of gaining speed. They are certainly not the most memory-efficient means of storing data, but they provide very fast look-up times. ZStream [MM09] replaces equality predicates with hash-based lookups wherever possible. Hashing can reduce search costs for equality predicates between different event classes; otherwise, equality multi-class predicates can be attached to the associated operators as other predicates. As shown in Figure 2.7, the first event T1 is hashed on *name* and when the equality predicates *T1.name=T3.name* is applied, a lookup in the hash table built on T1 can be performed directly.

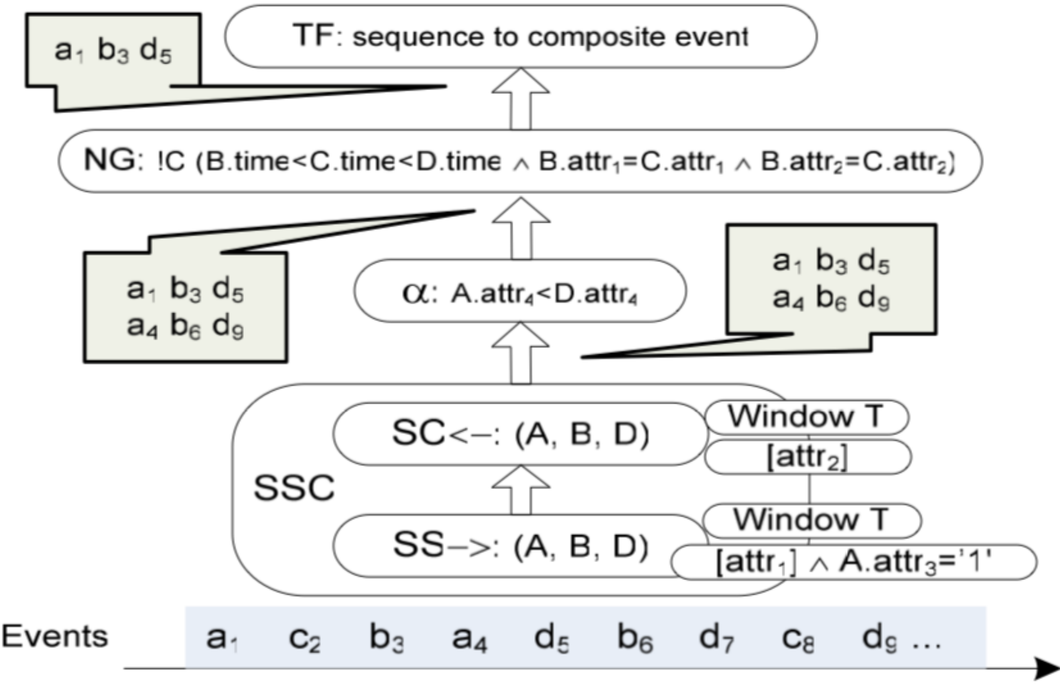


Figure 2.8: Sequence Scan and Construction Optimazation [WDR06]

2.4.2 Optimization of query plan generation

CEP systems take a query and should convert it to a query plan for execution. The typical steps are as follows. First, the query is rewritten in a more efficient way. Then, the cost models are used to be able to determine the optimal query plan. Cost models can be static or dynamic using the characteristics of events such as their arrival frequency or their predicate selectivity. The static cost model is based

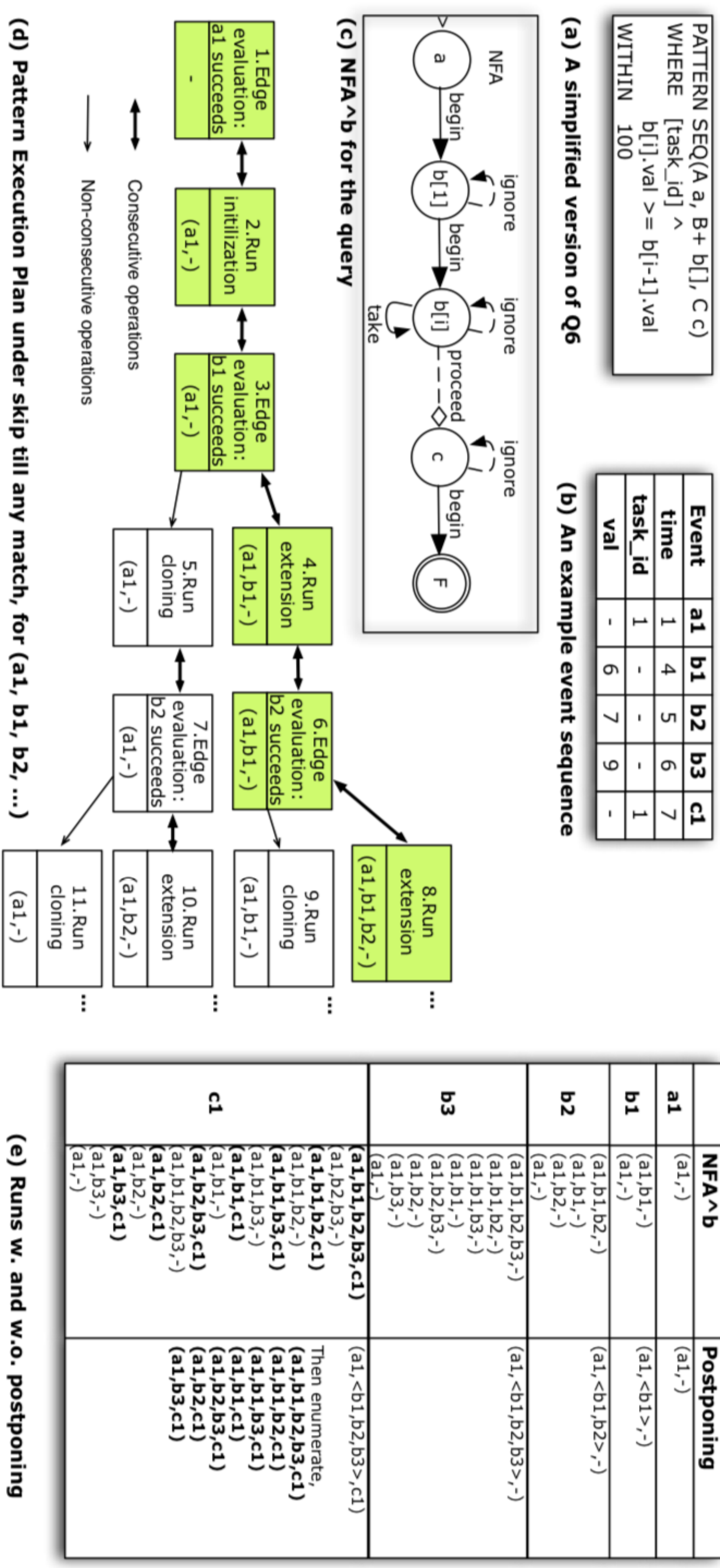


Figure 2.9: Postponing with early filters Optimazation [ZDI14b]

on fixed assumptions of events characteristics. Dynamic cost models are updated and used on-the-fly based on such characteristics to swap to a new optimal plan. This dynamic system must also take into account the cost of regenerating a new plan. Automata-based CEP [Bre+07; SMP09], described in section 2.3.4.1, use the order-based-plan to set the order in which arriving events will be processed. The Order-based-plan consists of a permutation of primitive event types defined by a pattern. While Tree-based CEP [MM09] use the tree-based plan to find the optimal physical trees (e.g., left-deep, right-deep or bushy), that can be used to evaluate the defined pattern.

[AÇT08] were the first to explore cost-based planning for complex event detection with the aim of minimizing network traffic across distributed event sources, while also helping to reduce the processing cost. The defined cost model was based on temporal constraints among constituent events and events frequency statistics. In NextCEP [SMP09], authors limit the query rewriting for three operators OR, SEQ and NOT. For the OR operator, they use a greedy Huffman algorithm [Huf52] to create a tree and the optimal order of the OR operator is generated with a depth first traversal. For the NEXT and NOT operator they use a dynamic programming solution where the lowest cost pattern can be found by enumerating all the equivalent patterns and computing each cost. The NextCEP cost model was based on the arrival frequency of the primitive events. In ZStream[MM09], authors define a series of rules that generate an exponential number of equivalent expressions, called algebraic rules, in order to avoid exhaustive searches of all these expressions as done by NextCEP[SMP09]. Once a query expression has been simplified using the algebraic rewrites, i.e. *(SEQ(A,AND(!B,!C),D) is translated simply to SEQ(A,!OR(B,C),D))*, and hashing has been applied on the equality attributes, the authors use a cost model for optimal tree-based plan generations, where they apply a dynamic programming solution to enumerate all possible tree topology costs. In contrast to NextCEP the cost model takes into account the arrival rates of primitive events and the selectivity of their predicates. In addition, they define a constant threshold t for the monitored statistics to trigger plan regeneration. In [KSS15a; KSK16], authors propose a lazy evaluation approach where events of high frequency are buffered until the rare event type instance is received. In case the selectivity of events is already known and remains invariant, they use a simple NFA structure to determine potential pattern matches. Otherwise, if the selectivity of events may change over time they use a Tree-NFA, an NFA backed by a tree structure, which reorganizes itself according to the present observed events rates, similarly to Eddies[AH00]. The most important strength of these approaches is that the optimal evaluation plan is guaranteed for any given set of events. However, it results in an overload of plan generation where the process of creating and deploying

a new evaluation plan is very expensive. As discussed before, this was proposed in ZStream[MM09] to avoid this overload, however, some re-optimization opportunities may have been missed. Recently, [KS18a] presented a novel method for detecting re-optimization opportunities by verifying a set of distances on the monitored data characteristics. They formally prove that they avoid launching re-optimization plan generation that does not improve the currently employed plan. To do this, they define a list of in-variant distances. The idea is to use margins instead of a simple thresholds to avoid the ping-pong effect i.e. keep changing query plan due to noise. Thus in-variant distances are defined between events having the closest arrival rates, in case even a slight oscillation in the events rates causes the event types to swap positions periodically when ordered according to this statistic. They use some data analysis methods to discover the minimal distance that will be used as a violation constraint. However, finding the right distance remains as always a difficult task and depends on the data and keeping in mind the overload of this method if it were applied on-the-fly. These contributions aim to reduce the number of active partial matches by rewriting/reordering the queries based on statistical characteristics of the primitive events. However, the number of partial matches remains one of its major problems, especially when dealing with expensive queries or large windows, even if some existing work focused on their smart indexing.

2.4.3 Optimization of memory

In the context of CEP, memory management is critical for performance measurement where workers aim to optimize throughput and not to shed the storage load directly. With the arrival of primitive events, the number of storage data structures grow proportionally, such as events' buffers and partial active matches' data structures. In [Agr+08], the evaluation plan is set to the initial order of the sequence pattern, but they design efficient data structures to enable smart runtime memory management. They define a versioned system of shared buffer to compactly encode all active partial and complete matches instead of having a single buffer for each one. To avoid erroneous results they use version numbers that share the same prefix with all the events of the same partial match, and a pointer to the previous event, as shown in the figure 2.10. In addition, they incrementally prune all the used data structures based the window size and reuse the expired instance whenever possible. This strategy can reduce memory requirements from exponential to polynomial, at the cost of the compression/decompression operations.

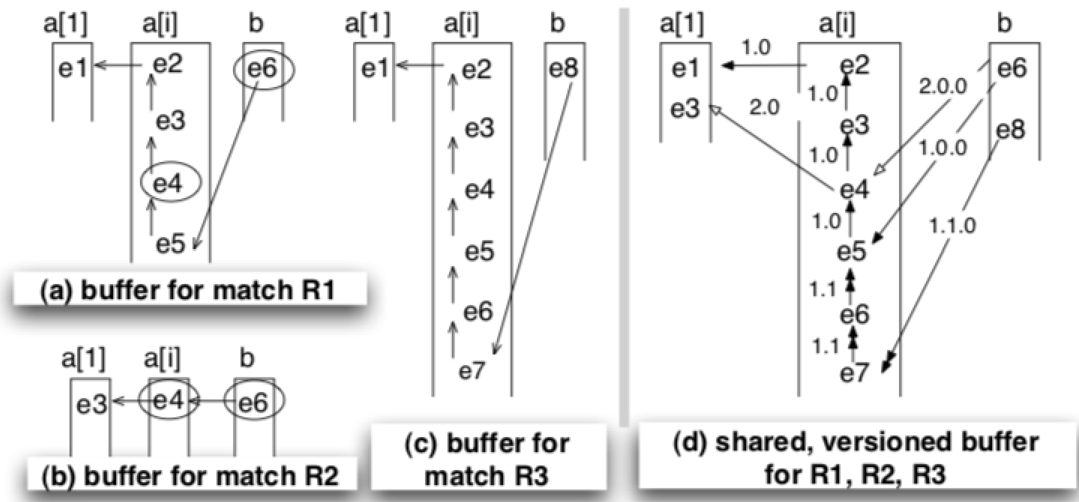


Figure 2.10: shared versioned buffer Optimization [Agr+08]

To summarise, the table 2.1 concisely presents the optimization techniques used, which show very promising results while still remain, largely, unsolved. Thus, optimization issues are still far from being fully resolved.

2.5 Predictive Analytics & Complex Event Processing

Predictive Analytics *encompasses a variety of techniques from data mining, predictive modelling and machine learning, which analyze current and historic facts to make predictions about future or otherwise unknown events*[1][NC07]. Over the past several years, this field has seen an explosion of interest from both academia and industry and has been implemented in different contexts. Complex event processing and Predictive Analytics both target detection of undesired behaviour and patterns of interest, where results of one field can be readily applied to the other making it more efficient and more powerful. Hence, with the help of PA, CEP could be made more automatic and intelligent leading to improved accuracy, speed, consistency, and easier maintenance. PA can be used to learn new rules from data, to make optimal decisions for query optimization and to predict complex event detection based on sequence prediction.

With regard to the focus on optimization trends in complex events processing in section 2.4, we will first discuss how PA may influence the query plan adjustment to avoid low throughput. Then, how automatic rules can be learned and created. Finally, we discuss existing techniques to enable the prediction of complex events.

[1]**https://www.predictiveanalyticsworld.com/predictive-analytics**.

Approach			Optimization Strategies				
			Predicate Related Optimization			Query rewriting	Memory Management
			Push Down	Postponing	Hashing		
Detection Models	Automata based	Akdere [AÇT08]	$\surd$	ϕ	ϕ	$\surd$	ϕ
		Cayaga [Bre+07]	ϕ	ϕ	ϕ	$\surd$	$\surd$
		SASE [WDR06]	$\surd$	ϕ	$\surd$	ϕ	ϕ
		SASE+ [Agr+08]	$\surd$	ϕ	$\surd$	ϕ	$\surd$
		SASE++ [ZDI14b]	$\surd$	$\surd$	$\surd$	ϕ	$\surd$
		FLINK [Fli]	$\surd$	ϕ	$\surd$	ϕ	$\surd$
		LazyNFA [KSS15a; KS18a]	$\surd$	ϕ	ϕ	$\surd$	ϕ
		NextCEP [SMP09]	ϕ	ϕ	ϕ	$\surd$	ϕ
	Tree Based	Zstream [MM09]	$\surd$	ϕ	$\surd$	$\surd$	$\surd$
	Graph Based	CET [Pop+17]	$\surd$	$\surd$	ϕ	ϕ	$\surd$
	logic based	Etalis [Ani+12]	ϕ	ϕ	ϕ	ϕ	ϕ

Table 2.1: CEP Optimisation Strategies & Detection Models

2.5.1 Predictive Analytics for optimal decision making

Predictive Analytics technologies are also implemented to improve performance of complex event processing systems. As discussed previously, in [MM09; KSK16; KS18a], statistics were used for query plan adaptation to avoid low throughput. A slightly different approach is followed in [LG16], where workers assume that the history could be a mirror to the future. They propose ACEP as an approximate complex event processing system with the aim of maximizing the number of full matches found. Their idea is to represent a history of past matches concisely into a data structure. Then at run time they discard some matches that are unlikely to result in a final full match. In the same context, other works deal with uncertainty in the occurrence of events and prune intermediate results (partial matches) during pattern matching process [Art+12]. Finally, PA is also used in a distributed CEP context to reduce communication costs incurred while exchanging event information between distributed systems [AÇT08].

2.5.2 Predictive Analytics for automatic rules generation

Faced with the complexity of defining complex queries and with the aim of reducing human involvement, different works [TGW09; SSS10; MP12; MCT14; MTZ17] have focused on the automatic generation of CEP rules. Thus they extract rules that are driven by data and learned from history. [MP12] introduces a domain expert dependent method based on hidden Markov models (HMM). The expert reports the occurrence of a relevant complex event at a specific point in time, the system then automatically deducts the rule allowing the recognition of such events in the future. Similarly, [MCT14] proposed a framework that analyzes historical traces to infer the hidden causality between received events and detected situations, and uses them to automatically generate CEP rules. On the other hand, they solve the learning problem by modelling rules and time series as a set of constraints and calculating their intersections. Recently, [MTZ17] developed a generic system that extracts predictive temporal patterns with time and sequence constraints and then transforms these patterns on-the-fly into predictive CEP rules. To achieve this, they use a Shapelet-based classification for rules discovery over multivariate (multi-dimensional) time series [Lii91]. Most of these contributions were in complete agreement and highlight the importance of handling multidimensional data mining algorithms.

2.5.3 Predictive Analytics for complex events prediction

Complex Event prediction aims to provide the possible future events of partially matched sequences which can be turned into a full match. This would enable users to mitigate or eliminate undesired future events or states and identify future opportunities. The problem of predictive CEP has remarkable similarities with the *sequence pattern mining and prediction*. In this context, a large body of sequence prediction models have been proposed including: Prediction By Partial Matching (PPM) [CW84], All-K-Order-Markov [PP99] and the Probabilistic Suffix Tree [BYY04]. These models are based on the Markov property and suffer from the catastrophic forgetting of older sequences [Kir+17]. Only k recent items of training sequences are used for performing prediction: increasing k often induces a very high state complexity and consequently such techniques become impractical for many real-life applications [Gue+15].

2.5.3.1 Sequence Pattern Mining and Prediction

A large number of sequence prediction methods [BYY04; CW84; Gue+15; PP99] come from the field of temporal/time-series pattern mining, where patterns are defined using association rules or as frequent episodes. These methods employ variants of decision trees and probabilistic data structures. Most of them are based on the Markov property and suffer from the issue of forgetting older models and have high computation costs. Recently, two methods CPT and CPT+ [Gue+15] have been proposed, which keep all historic data in a compressed format and offer increased accuracy. It is based on the prefix tree (aka trie) data structure and supports only one dimensional sequences. Furthermore, these models are not optimised for streaming applications, where the training dataset is unknown; a large number of events arrive at a high rate; events occur at random intervals rather than at the regular tick-tock intervals of traditional time series and, thus, a real-time response is required.

2.5.3.2 Machine Learning-based Predictions

Further directions for tackling this problem are incremental (online) machine learning algorithms which learn incrementally over event streams, such as Support Vector Machines, recurrent Neural Networks and Bayesian Networks [FB12; AÇU10; MRT12]. For these algorithms, classifiers are updated each time a new training instance is found to provide a predictive response. The main disadvantages of these algorithms in the context of CEP are as follows: (i) they are use-case specific and require considerable effort (and training datasets) to model each dataset, e.g. a model can either be based on *recent* history or is *updatable* based on

the older values; (ii) they do not provide any performance guarantees in terms of error bounds, which causes additional difficulty when runnig actions such as performance tracking and regular maintenance: since the learned parameters keep on changing dynamically [MRT12].

2.6 Processing Top-k queries

In this section, we take an interest in particular types of continuous queries which monitor top-k data objects over sliding windows. *Top-k queries* cover many real-time applications ranging from financial analysis to the system monitoring. We distinguish CEP queries from top-k queries by the fact that the former aims *to detect* data while the latter aims *to filter* data from an incoming stream. In the first case, the final user will be provided the data that is extracted; in the latter, the final user will be provided what is left after the data is filtered.

2.6.1 Definitions

The notion of continuous top-k queries was introduced by [MBP06] where a query is defined as a linear scoring function determining the relevance of an item and the result of such query is the K most highly ranked items. Top-k query replaces the first ranking model proposed by [Lin+05] which defines a query as a set of attributes to optimize (e.g minimize or maximize). They are known as *Skyline queries*, where the skyline ranking was replaced by a linear function. The result of such query is a set of tuples that are not *dominated* by any others, thus forming a skyline. Where an item i_1 dominates another item i_2, if and only if i_1 is preferable to i_2 in all the attributes of the query.

Definition 6: Top-k Query

Top-k Query provides only matches with the highest score, based on a user-specified *scoring function.*

Definition 7: Scoring Function

The Scoring Function $F(p_1, p_2, ..., p_n)$ generates a score for each match of the query by aggregating scoring predicates, where p_i is a scoring predicate.

2.6.2 Top-K Algorithms

The top-k query problem has been investigated in static databases and used in various applications where end-users are interested only in the most significant responses in the huge response space. The well-known top-k algorithms, ONION [Cha+00] and

PREFER[HKP01], cannot be directly used in static databases nor can they be easily adapted to fit streaming environments. Indeed, given the considerable volume of data, the core of the problem which they have solved is as follows. How can the data be pre-analyzed to prepare the appropriate meta-information, in order to effectively respond to the top-k queries? Clearly, in a streaming environment, computing complete changes for each update and re-initializing the process of identifying top-k elements for each newly observed change is too expensive. Therefore, the main challenge is to provide an incremental process that can effectively update top-k results, even at extremely high throughput and in very large query windows. The workers' advanced efforts in continuous top-k query on data streams fall into two groups, namely multi-pass approaches and single-pass approaches. The former is a re-computation approach while the latter is an incremental approach.

2.6.2.1 Multi-pass-based approaches:

In Multi-pass-based approaches [Yi+03], the idea is to maintain a larger number $k^{'}$ as a candidate set C instead of maintaining k objects in the current window, where $k \leq k^{'} \leq k_{max}$ and k_{max} is the maximum capacity of C. Whenever a query result expires from the window then recomputation of the top-k results is only performed when the top-$k^{'}$ window has less than k objects ($|C| < k$). This is because recomputing the top-k results is the most expensive operation for top-k maintenance. [MBP06] proposes the TMA algorithm (Top-k Monitoring Algorithm) to reduce the re-computation cost by maintaining a "grid structure" to access only a few cells by defining an influence region. The influence region of a query specifies the data space to which an object must belong in order to modify the results of the query. Therefore, any update in the influence region will modify the result of the query. The re-computation process is involved only when some of the existing top-k objects expire and new arrivals have a lower score than the expired records (so that the influence region expands). The main drawback of this algorithm is that it is data distribution dependent. At the same time, [MBP06] proposes a SMA (Skyband Monitoring Algorithm to address the problem of re-computation bottlenecks in the TMA algorithm. The authors use the K-Skyband as the minimal set of objects required to be maintained in order to answer top-k queries where K-Skyband consists of every object that is dominated by at most (k-1) other objects. SMA is required to be faster than TMA at the expense of memory consumption used to index K-Skyband objects, while both need to conduct expensive top-k re-computations from scratch in certain cases.

2.6.2.2 One-pass-based approaches:

In one-pass-based approaches, [Yan+11] proposed their algorithm named MinTopK. This algorithm eliminates the re-computation bottleneck and thus realizes completely incremental computations and minimal memory usage. Based on the assumption that the life span of an incoming object is known upon its arrival, it is possible to predict if its score allows it to be in the future top-k window sets. Thus it enables the maintenance of partitioned sub-window candidate sets and to immediately eliminate all objects that will never be in the top-k sets. MinTopK is highly sensitive to data distribution, especially when the sliding window is too small, thus the number of insertions/deletions required to maintain the candidates is dramatically increasing. SAP [Zhu17] enhanced the MinTopk algorithm with a dynamic self-adaptive partition of the window based on different query parameters and data distributions. To the best of our knowledge, MinTopK and SAP are the best solutions in the case of top-k algorithms that output exact results, with respect to approximate ones such as [Zhu+17]. [Zhu+17] proposes a (ϵ, δ)-approximate continuous top-k system that filters out arrived objects which have a probability less than 1 - δ of being a query result, and generates summary information on candidates with roughly the same scores.

All algorithms previously mentioned cover the needs for major top-k query processing systems, but one size does not fit all and more complex top-k systems have requirements that cannot be met by this paradigm. These conditions include, for example, if the ranking score is related to the arrival time of the object, second, if the arrival of a new object may change the scores of old ones in top-k or the candidate set, and third, if the data expiration is not linear with the time, systems based on sliding-windows are not adequate as they can only treat linear data expiration. Therefore these real-world data stream processing applications require ad hoc developments with standard programming languages.

2.7 Conclusion

In this chapter, we provided the concepts and the background to the topics addressed in this thesis. We also provided specific background to understand the challenges of processing expensive queries and designing predictive systems. We discussed the recent advances and related works ranging from query optimization to predictive analyses.

Obtaining high performance at reasonable cost is a key concern for stream processing systems. Thus, different works address the big data aspects such as volume and velocity of event streams. Some works focus on throughput optimization and some works optimise memory consumption. They aim to reduce the number

of active partial matches, which consume memory, by rewriting/reordering the queries or based on statistical characteristics of the primitive events. However, the memory consumption due to a large amount of partial matches remains one of the major problems. This is especially true when dealing with expensive queries or large windows. Even if some existing works use smart indexing techniques with promising results, still there are performance issues which are largely unsolved. Thus, optimization issues remain fairly unexplored. Thus, in this thesis one of the focus is on optimizing CEP performance by eliminating partial matches and using intelligent indexing technique.

We also saw that one of the trends is to combine Predictive Analysis (PA) with Complex Event Processing (CEP) systems. Hence, with the help of PA, CEP could be made more automatic and intelligent leading to improved accuracy, speed, consistency, and easier maintenance. PA can be used to make optimal decisions for query optimization, to learn new rules from data, and to predict complex event detection based on sequence prediction. In this thesis we focus on predicting future complex events. For this problem, some of the existing solutions are not suitable for streaming applications. Other works are use-case specific and require considerable effort to model each dataset.

Finally, we consider also a particular type of continuous queries which monitor top-k data objects over sliding windows to identify k highest-ranked data items. Top-k queries face several challenges because they have to maintaining a candidate list of data items that can be the part of top-k list in stream settings. Due to stream settings, the regular arrival of new data may change the state of existing top-k candidate list. Thus, it requires efficient incremental algorithms and data structures.

Above research gaps now set the context for the remaining chapters that will detail our contributions.

Part II

Enhancing Complex Event Processing

3

Enabling Efficient Recomputation Based Complex Event Processing for Expensive Queries

Contents

3.1 Introduction **46**
3.2 Motivating examples **46**
3.3 Related Works **48**
3.4 Preliminaries and definitions **51**
3.4.1 Definitions 51
3.4.2 Query Tree 53
3.5 Baseline Algorithm **55**
3.6 RCEP: Recomputational based CEP **57**
3.6.1 The Event Tree 57
3.6.2 Creating the Complex Matches 59
3.7 RCEP: General Recomputation Model **65**
3.7.1 Multidimensional Events 65
3.7.2 Multidimensional Event Tree 66
3.7.3 Joins between Z-addresses 68
3.7.4 Handling Sliding Windows 72
3.7.5 Optimising Z-address Comparison 74
3.8 Experimental Evaluation **74**
3.8.1 Setup and Methodology 74
3.8.2 Performance of Indexing and Join Algorithms 76
3.8.3 Performance of Sliding Windows 78
3.8.4 CEP Systems' Comparison 79
3.9 Conclusion **83**

3.1 Introduction

As discussed earlier, traditional strategies of Complex Event Processing (CEP) require extensive utilization of memory and CPU to extract complex temporal matches from events stream defined via a CEP query pattern. This is mainly due to the way in which complex temporal correspondences are extracted from an event flow. In fact, matches incrementally emerge over time while partial matches accumulate in the memory and then they are processed sequentially for incoming events. The number of partial matches for expressive CEP queries can be polynomial or exponential to the number of events within a window (proof: 3). Reducing the number of partial matches by compressing them is not a satisfactory solution because it is highly use case dependent and only aggravates the repeated calculation of matches when they are compressed.

In this chapter, we demonstrate how the recomputation of matches, without relying on the partial matches, can be made into a first-order CEP strategy for complex operators such as skip-till-any-match and Kleene+. In particular, we show how to efficiently store and query multi-attributed events within a window and how to join them in batches to efficiently recompute matches for an incoming event. Our hybrid-join algorithm reuses previously computed joins to further economise on recomputation of matches. The proposed techniques result in the reduction of memory and CPU requirements while increasing the throughput by many orders of magnitude compared to existing solutions. Our experimental observations support our arguments and the vast advantages of our techniques over previous methods have been confirmed through experimentation.

3.2 Motivating examples

We now provide three real-world examples that use complex operators for CEP.

- *(1) Stock Market Analytics:* Stock market analytics platforms evaluate expressive CEP queries against an events stream to identify and then exploit opportunities in real time. A common query in this context is the V-shaped or inverted V-shaped pattern [ZDI14a; Agr+08; Pop+17; MM09]. For such patterns, we look for stock events that consist of downticks (denoted as **a**) followed by one or more monotonically increasing upticks (denoted as **b**), which is then followed by a downtick (denoted as **c**), thus an inverted V-shaped pattern. Query 1 expresses such pattern, where a Kleene+ operator is used

to select one or more monotonically increasing events (Stock + b[]) for each distinct [comapnyID] within a window of 30 mins which slides every 2 minutes.

QUERY 1 . PATTERN SEQ (Stock a, Stock+ b[], Stock c)
WHERE [companyID] AND a.price < b.price AND
b.price < NEXT(b.price) AND c.price < FIRST(b.price)
WITHIN 30 mins SLIDE 2 mins

- *(2) Credit-Card Fraud Analytics.* The goal of credit card fraud management is to detect fraud within a very short time, in order to prevent financial loss [Art+17b]. A common query in this context is to detect the "Big after Small" fraudulent transaction [AAP17; Art+17b]. That is, the attacker first withdraws a large amount of money after one or a series of small amounts. Query 2 shows such pattern, where it is matched for each distinct [*cardID*] over the defined window and the granularity it slides with.

QUERY 2 . PATTERN SEQ (Card a, Card+ b[])
WHERE [cardID] AND a.amount ≥ b.amount * 5 AND
b.amount > NEXT(b.amount) WITHIN 30 mins SLIDE 5 mins

- *(3) Health-Care Analytics.* Cardiac arrhythmia is a group of serious heart diseases in which the heartbeat is irregular, too fast or too slow. It can lead to life-threatening complications such as stroke or heart failure.

QUERY 3 . PATTERN SEQ (Activity a, Activity+ b[])
WHERE [personID] AND a.rate*2 < b.rate AND b.rate < NEXT(b.rate) AND
b.type = 'passive' WITHIN 10 mins SLIDE 1 mins

Query 3 monitors physical activity per patient and detects life-threatening conditions when the heart rate gradually increases until it doubles in comparison to the first measurement, despite passive physical activity. The matched patterns provide lifesaving monitoring in real-time during the critical stages of cardiac arrhythmia [Pop+17].

Complex Operators From the CEP Queries

In contrast to general regular expression matching, CEP employs complex operators such as event selection strategies to select events within a matched pattern that are not contiguous to each other. These event selection strategies are classified as *partition contiguity* (PC), *skip-till-next match* (STN) and *skip-till-any match* (STA), as explained in chapter 2. PC partitions the matches according to certain defined attributes and requires an event to directly follow another given type of event in a match. Both STN match and STA strategies determine how events can be skipped between the produced matches. STN skips events until a relevant event arrives, while STA skips all the events to find all possible combinations, and is the most flexible of them all. Fig. 3.1 (a,b,c,d) shows the execution of Query 1 and Query 3 over events streams. From Fig. 3.1 (a,b), the execution of Query 1 with the STN strategy would only produce the first two matches, while STA strategy provides all combinations. Similarly from Fig. 3.1 (c,d), only the first match (in window W_1) is produced for the STN and remaining are produced for the STA strategy. The use of STA strategy is essential in many use cases since it provides the flexibility to skip local fluctuations. For instance, match number 9 (and its enumerations 3 - 8) in Fig. 3.1 (b) can only be produced with the STA strategy.

3.3 Related Works

Processing Model: Plenty of algorithms and CEP systems have been proposed as discussed in the 2nd chapter. A common strategy used by these systems is to incrementally produce query matches while first producing partial matches with the arrival of events. Hence, when a new event arrives, the system (i) creates a partial match and (ii) checks with all existing partial matches if a full match can be constructed. The strategy may seem viable at first glance; to incrementally process matches would seem to avoid the recomputation cost. However, the number of partial matches to be stored and processed can be exponential to the number of events within a window for the STA and Kleene+ operators. The system cannot discard any partial match unless the window expires since each partial match can lead to a possible match. For these reasons, existing CEP systems do not perform well when stressed over the real-world work loads and expressive queries as described above. One common remedy proposed to optimise the incremental approach is to factorise the commonalities between partial matches that originate from the same set of events [ZDI14a; KSS15a; Agr+08]. For this, the query evaluation is broken into two phases. The first phase tracks commonalities between partial matches and compresses them using an additional data structure. The second phase constructs complete matches while decompressing the set of common partial matches. This

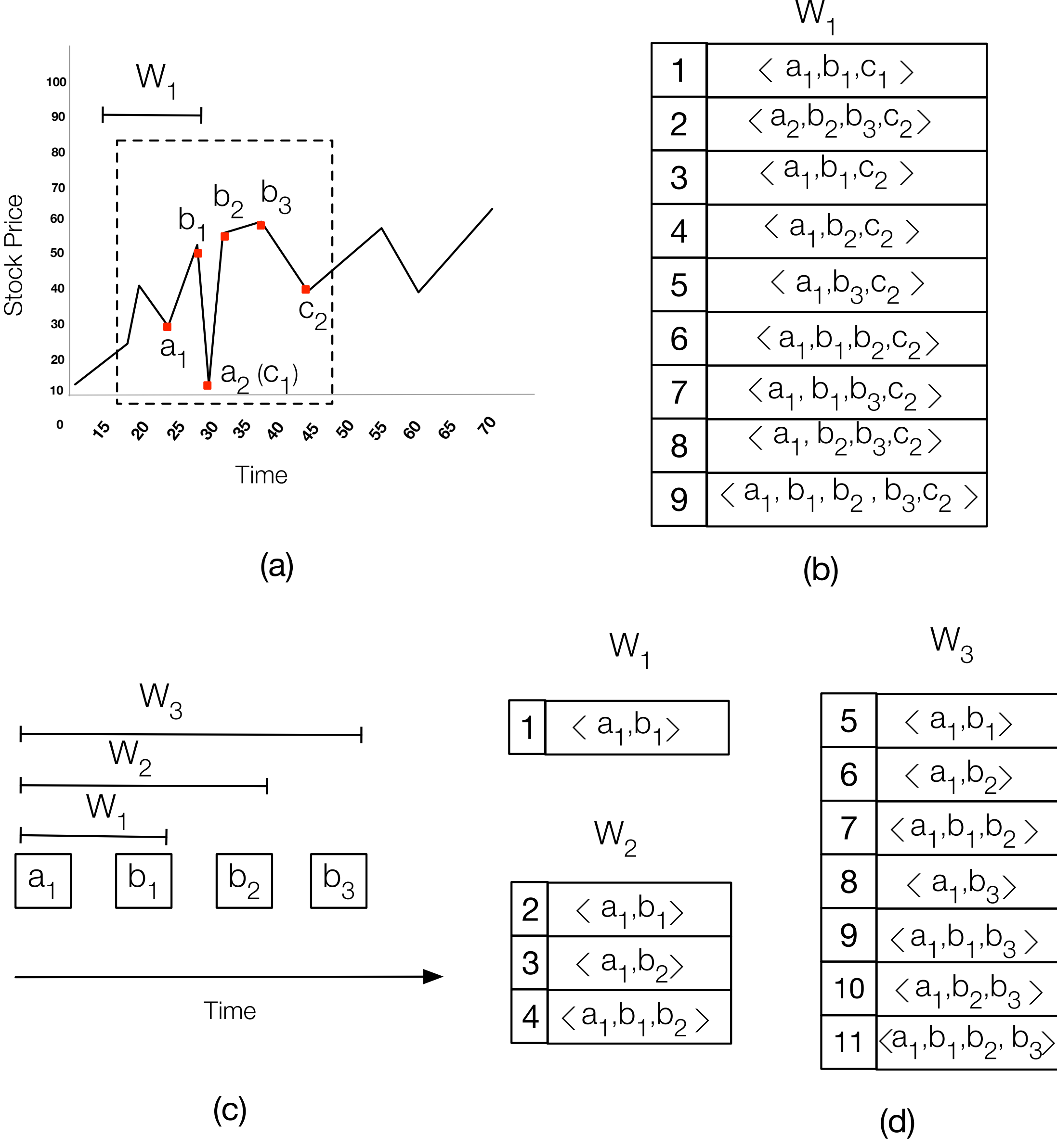


Figure 3.1: (a) Events stream S_1 for the Stock Market, (b) matches for Query 1 over S_2, (c) Events stream S_2 of a patient's heart beat, (d) matches for Query 3 over S_2 and three windows.

strategy can reduce the memory cost. Nonetheless, it results in the added cost of the compression/decompression operations and redundant computation of partial matches that are shared by multiple complete matches.

Scalability: Existing CEP solutions fail to scale over real-world datasets due to their space and time overheads. For example, SASE+ [ZDI14a] and Apache Flink [Fli] require an index space that is two to three orders of magnitude larger than the number of events in a window. This renders these solutions only applicable on small size datasets and simple CEP queries.

Proposed Approach

In this chapter, we advocate a new type of model for CEP wherein we identify two key points: (i) storing partial matches is expensive both in terms of CPU and memory cost, hence only events within a certain window should be stored; (ii) the matches should be recomputed from those set of events only when we are sure of their existence.To materialise these points, we require efficient indexing and querying techniques. That is, we need to avoid the recomputation process on all stored events, and the existence of new matches for an incoming event should be identified before starting the complete recomputation process. Hence, our journey to provide efficient techniques led us to explore various diverse fields such as theta-joins (which allows comparison relationship such as $\leq$ and $\geq$), multidiemsnional indexing and range query processing. We employ space filling curve [Mor66] and a B+tree index for events within a window, and provide multiple optimisation techniques so sush an index is a viable option for the streaming settings. We also provide a hybrid-join algorithm for theta-joins with inequality predicates. These optimisations enabled us to efficiently prune events before starting the recomputation process, but also to execute such process only when it is required. Our provided techniques are generic in nature and can be employed in general streaming-based systems.

Contributions

Our key contributions are as follows:

- (1) We provide a novel recomputation model for processing expressive CEP queries. Our algorithms reduce the time and space costs for complex queries with Kleene+ and STA operators.

- (2) To efficiently employ a recomputation model, we propose a hybrid-join algorithm to produce final matches from events in a batch mode. Our algorithm leverages the dominance property between set of events and joins the clustered events in batches to reduce the cost of pair-wise joins.

- (3) We provide multiple optimisation techniques for storing events using space-filling curves. These techniques enable us to use efficient multidimensional indexing in streaming environments.
- (4) We experimentally demonstrate the performance of our approach against state-of-the-art solutions in terms of memory and CPU costs under heavy workloads (Section 3.8).

3.4 Preliminaries and definitions

In this section, we present CEP query representation and query evaluation techniques and the notations are defined in table 3.1.

3.4.1 Definitions

CEP Query

Definition 8: CEP Query

A CEP Query Q is defined as a quadruple $Q = (P, \Theta, \omega, s)$, where $P = \langle p_1, p_2, \dots, p_k \rangle$ $(k \geq 1)$ is a sequence of pairwise disjoint variables of the form p and p^+, Θ=$\{\theta_1, \theta_2, \dots \theta_l\}$ is a set of predicates over the variables in P. ω is the time window and s specifies the slide parameter, which determines the granularity at which the window content changes.

The $p \in P$ variable binds the sequence of a single event $\langle e_i \rangle$, while the qualified variable $p^+ \in P$ binds a sequence of one or more events $\langle e_1, e_2, \dots, e_n \rangle$, with $n \geq 1$, for a query match. Θ is a set of predicates over variables which must be satisfied by matching events. We further distinguish between *constant* and *variable* predicates, i.e. $\Theta = \Theta^c \cup \Theta^v$. $\theta \in \Theta^c$ has the form $(p.A_x \ \phi \ c)$: $p.A_x$ refers to an attribute of matching events and $c \in \mathbb{N}$ is a constant, $\phi \in \{>, <, \leq, \geq, =\}$. The variable predicates $\theta \in \Theta^v$ have the following forms: (i) $(p_i.A_x \ \phi \ p_j.A_x)$, i.e. a relationship between the attributes of two different matching events; (ii) $(p_i.A_x \ \phi \ \mathsf{NEXT} \ (p_i.A_x))$, i.e. a relationship between the current and next event using the NEXT function; and (iii) $(p_i.A_x \ \phi \ \mathsf{FIRST} \ (p_i.A_x))$, i.e, a relationship between the current and first selected event in the specific $\vec{E}$ using the FIRST function.

Example 1: From Stock market query, we have three variables p_1= a, $p_2 = b^+$ and p_3 =c, where p_2 has the Kleene+ operator applied to it. The query contains the following variable predicates: $\Theta = \Theta^v_a \cup \Theta^v_b \cup \Theta^v_c$, such that $\Theta^v_a = \{(a.id = b.id), (a.price < b.price)\}$, $\Theta^v_b = \{(b.id = c.id), (b.price < \mathsf{NEXT}(b.price)), (a.price < b.price), (c.price < \mathsf{FIRST}(b.price))\}$ and Θ^v_c =$\{(b.id = c.id), (c.price < \mathsf{FIRST}(b.price))\}$. Temporal relations, which are the sequences between the variables $p \in P$ are

described through the `SEQ` clause. The size of the window is `30 minutes` and it slides every `2 minutes`.

CEP Query Matching

To define the matching of a CEP query Q, we use a substitution $\gamma = \{p_1/\vec{E}_1, \dots, p_k/\vec{E}_k\}$ to bind the events sequences ($\vec{E}$) with the variables. Each pair $p/\vec{E}$ represents a *binding* of p variable to a sequence $\vec{E} = \langle e_1, e_2, \dots, e_n \rangle$ of events in $\mathcal{S}$. These bindings are then evaluated over the set of predicates Θ in Q.

Definition 9: CEP Query Match

Given a CEP query $Q = (P, \Theta, \omega, s)$ and an events stream $\mathcal{S}$. A substitution $\gamma = \{p_1/\vec{E}_1, \dots, p_k/\vec{E}_k\}$ is a match of Q in $\mathcal{S}$ iff the following conditions hold:

1. $\forall \theta \in \Theta$, $\mathsf{Eval}(\theta\gamma) = \mathit{true}$,
2. $\forall p_i/\vec{E}_i, p_j/\vec{E}_j (j > i) \in \gamma$, $\forall e \in \vec{E}_i\ \exists e' \in \vec{E}_j$ such that $e.t < e'.t$.
3. $\forall p_i/\vec{E}_i, p_j/\vec{E}_j (j > i) \in \gamma, \wedge\, e_n \in \vec{E}_i,\ e'_1 \in \vec{E}_j,\ |e_n.t - e'_1.t| \leq \omega$.

The Semantics of the Eval Function

Herein, we provide details of the Eval function used in the CEP Query Match definition 9. That is, how Next, First and Θ are evaluated on selected events.

The substitution $\gamma = \{p_1/\vec{E}_1, \dots, p_k/\vec{E}_k\}$ binds the events sequences ($\vec{E}$) with the variables. Each pair $p/\vec{E}$ represents a *binding* of a variable p to a sequence $\vec{E}_i$ of events in $\mathcal{S}$. These bindings are then evaluated over the set of predicates Θ in Q. For a predicate $\theta \in \Theta$, $\theta\gamma$ denotes the *instantiation* of θ by γ and is obtained from θ by simultaneously replacing all variables $p \in P$ with the corresponding events sequences $\vec{E}$. The instantiation of a set of predicates Θ is $\Theta\gamma = \{\theta_1\gamma, \dots \theta_l\gamma\}$. The evaluation of the instantiation $\mathsf{Eval}(\Theta\gamma)$ is defined as follows:

- $\mathsf{Eval}(\Theta\gamma) \equiv \mathsf{Eval}(\{\theta_1\gamma, \dots \theta_l\gamma\}) \equiv \mathsf{Eval}(\theta_1\gamma) \wedge \dots \wedge \mathsf{Eval}(\theta_l\gamma)$
- $\mathsf{Eval}(\vec{E}.A_x\ \phi\ c) \equiv \forall e \in \vec{E}\ (e.A_x\ \phi\ c)$
- $\mathsf{Eval}(\vec{E}_i.A_x\ \phi\ \vec{E}_j.A_x) \equiv\ \forall e \in \vec{E}_i, e' \in \vec{E}_j\ (e.A_x\ \phi\ e'.A_x)$
- $\mathsf{Eval}(\vec{E}.A_x\ \phi\ \mathsf{Next}(\vec{E}.A_x)) \equiv\ \forall e_i, e_{i+1} \in \vec{E}\ (e_i.A_x\ \phi\ e_{i+1}.A_x)$
- $\mathsf{Eval}(\vec{E}.A_x\ \phi\ \mathsf{First}(\vec{E}.A_x)) \equiv\ \forall e_i (i > 1), e_1 \in \vec{E}\ (e_i.A_x\ \phi\ e_1.A_x)$

3.4.2 Query Tree

Given a CEP query, we need to compile it from a high-level language into some form of automaton [Bre+07; WDR06; Agr+08; Fli] or a tree-like [MM09; ESP] structure to package the semantics and executional framework. Since we are working with the recomputation-based model, a traditional tree structure customised for the streaming and recomputation settings would suit our needs. Given Q, we construct a tree where leaf nodes are the substitution pairs, i.e. $(p_i/\vec{E}_i)$ to store the primitive events and the internal nodes represent the joins on the defined predicates Θ and temporal ordering. We call it a query tree $\mathcal{T}_q$. Our model differs from other tree-structures [MM09; ESP], since we do not store any partial matches. An example of such a tree for Query 1 is shown in Fig 3.2, where we have three leaf nodes for the variable bindings $a/\vec{E}_1$, $b/\vec{E}_2$ and $c/\vec{E}_3$. The internal nodes in Fig. 3.2 evaluate the defined Θ in terms of joins (denoted as $\bowtie_\Theta$) for all the variables $p \in P$. Furthermore, given that for CEP queries the matched events are required to follow the sequential order, the joins on the timestamps (denoted as $\bowtie^t$) are also provided in the $\mathcal{T}_q$.

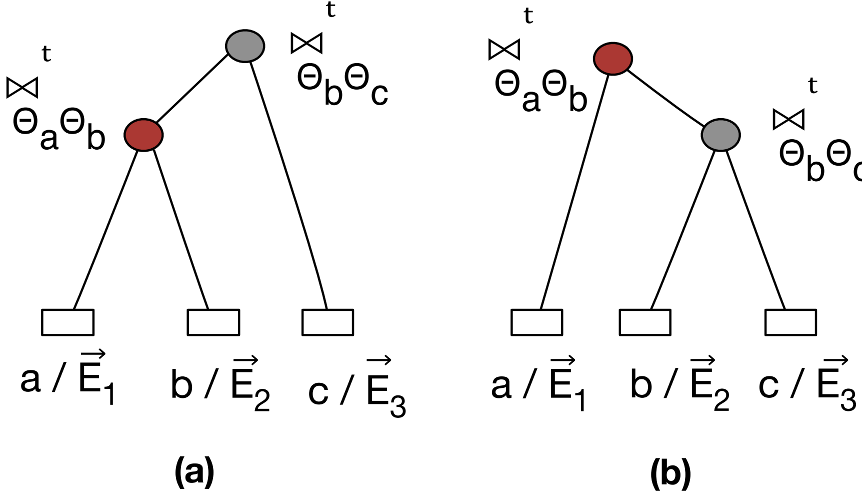


Figure 3.2: (a) Left-deep and (b) Right-deep Query tree for Query 1 in Example 1

Notation	Definition
$\mathcal{S}$	An event stream
e_i	An event
Q	Query
A	A set of attributes
P	A sequence of pairwise disjoint variables: $\langle p_1, p_2, \ldots, p_k \rangle$ of the form p and p^+
p_i	A variable used in the query such as a, b+ and c in the examples. See the definition of P above
p_i^+	A variable with Kleene+
Θ	A set of predicates: $\{\theta_1, \theta_2, \ldots \theta_l\}$ over the variables in P
Θ^c	Set of constant predicates
Θ^v	Set of variable predicates
θ	A predicate. See the definition of Θ above
ϕ	An operator $\in \{>, <, \leq, \geq, =\}$
$\vec{E}$	An event sequence
$p_i/\vec{E}_i$	A substitution pair representing a leaf node to store the primitive events
γ	A set of substitution pairs $= \{p_1/\vec{E}_1, \ldots, p_k/\vec{E}_k\}$ to bind the events sequences ($\vec{E}$) with the variables
$\mathcal{T}_q$	Query Tree
$\mathcal{T}_e$	Event Tree
$\bowtie_\Theta$	The leaf nodes evaluate the defined Θ in terms of joins
$\bowtie^t$	Join on the timestamp
B	Bitvector
$B_{u,v}$	Indicates whether a u element in $\vec{E}_i$ is joined with the vth of $\vec{E}_j$ using the predicates
$\vdash$	Operator denoting dominance (Dominance is defined later in the chapter)
$\nvdash$	Operator denoting not dominated
z	Z-address
$g_j^t(z)$	A function which takes a Z-address and returns a bitstring by extracting t MSB bits for every j bits skipped
$\veebar$	XOR operator
ω	The time window
w	Size of the window
s	The slide parameter of the time window

Table 3.1: Definition of notations

3.5 Baseline Algorithm

The aim of baseline algorithm is to evaluate the query tree over events within a defined window without storing partial matches and recomputing the matches for each incoming event. Hence, it first stores events in the relevant leaf nodes (events sequences) of the query tree. Second, it triggers the query evaluation to produce complex matches. Given a query tree $\mathcal{T}_q$, each incoming event $e \in \mathcal{S}$ can result in the following main steps.

Algorithm 1: Baseline Algorithm

```
   Input: Query Tree T_q and an events stream S
   Output: Set of Matched Sequences
1  Q ← (P, Θ, ω, s) ;                                        // CEP Query
2  E⃗ ← {E⃗_1, E⃗_2, ..., E⃗_k}, k = |P| ;                    // Events sequences from T_q
3  for each e ∈ S do
4      for each E⃗_i ∈ E⃗ do
5          if isCompatible(E⃗_i, e) then
6              E⃗_i = E⃗_i ∪ e ;                               // Step 1
7      if isCompatible(E⃗_k, e) then
8          ExecuteJoins(E⃗, Θ);                               // Step 2
9          ExecuteKleenePlus(E⃗, Θ);                          // Step 3
```

Step 1. Add e to the compatible events sequence $\vec{E_i}$, so that constant predicates Θ^c filter unwanted events for each $\vec{E_i}$ in $\mathcal{T}_q$. This step corresponds to the accumulation of events within a defined window. See Algorithm 1 from *lines 4-6*.

Step 2. For each incoming event e, check if such an event can trigger the query evaluation to produce matches. That is, if e can be part of $\vec{E_k}$ (i.e. last event sequence), it can complete a set of matches; since it contains the highest timestamp within the window. Hence, execute the query tree by joining the events within each $\vec{E_i}$ using the predicates Θ and timestamps. This step assembles all the events for each $\vec{E_i}$ that can produce the set of matches. See Algorithm 1 from *lines 7-8*.

Step 3. For a $p_i^+ \in P$, compute all the combinations for events in $p_i^+/\vec{E_i}$, i.e. a power set of events in $\vec{E_i}$. This step groups all the combinations by following the one or more semantics of the Kleene+ operator. See Figure 3.5 and Algorithm 1 *line 9*.

We now present details of the two main processes of Algorithm 1, i.e. joining the set of events and computing the power set of events for the Kleene+ operator.

Execution of Joins

Let $\vec{E}_i$ and $\vec{E}_j$ be two event sequences with theta-join $\vec{E}_i \bowtie^t_\Theta \vec{E}_j$ over the timestamp t and predicates Θ. Hence, we have joins on multiple relations in **Step 2**. The generic cost of such joins, i.e. pairwise joins, is $O(|\vec{E}_i||\vec{E}_j|)$ and the problem of its efficient evaluation resembles that of traditional theta-joins with inequality predicates [Kha+17]. The wide range of methods for this problem include: the textbook merge-sort, hash-based, band-join and various indices such as Bitmap [GUW02]. These techniques are mostly focused on equality joins using a single join relation. Inequality joins on multiple join relations are notoriously slow and multi-pass *projection-based* strategies [BF79; Kha+17] are usually employed. These strategies, however, require multiple sorting operations, each for a distinct relation, and are only optimised for static datasets, where indexing time is not of much importance. Even with these strategies, the worst-case complexity of these algorithms remains $O(|\vec{E}_i||\vec{E}_j|)$ as the output size can be as large as $|\vec{E}_i||\vec{E}_j|$, where all events share the same join keys and the join degenerates into a cartesian product. Considering this, we employ the general nested-loop join for our preliminary algorithm. Our experimental analysis showcases that even such algorithms provide competitive performance.

Execution of Kleene+ Operator

For **Step 3**, we need to create all the possible combinations of matches over the joined events. That is, enumerating the powerset of event sequences with p^+ bindings. In this context, we used Banker's sequence [LvHS00] to generate sequences of events with p^+ bindings, where we need only to count from 1 to $2^m - 1$. That is, we check the number of events in event sequences with p^+ bindings after the join process. For $|m|$ number of such events, we create 1 to $2^m - 1$ matches.

Complexity Analysis

Herein, we briefly present the complexity analysis for the three steps described in Algorithm 1. **Step 1** results in a constant time operation, since an incoming event can be directly added to an event sequence. **Step 2** has a polynomial time cost (pair-wise joins) and depends on the number of patterns P defined in a CEP query. For n events in a window and $k = |P|$, we have $O(n^k)$. **Step 3** requires the creation of an exponential number of matches for Kleene+ operators. For n events in a window, we have $O(2^n)$. However, this would require storing predecessor matches to produce the next one and would result in an extra load on memory resources.

Drawbacks

The two main drawbacks of the baseline algorithm are as follows. (1) In the absence of constant predicates, we cannot partition the events for specific events sequences $\vec{E}_i$. Therefore, each event is added to all the events sequences. (2) Even if we could have partitioned the events in their respective $\vec{E}_i$, we require a costly pair-wise join algorithm due to the multi-relational and inequality nature of the joins. This results in a major performance bottleneck in **Step 2**, we cannot discard an event within an $\vec{E}_i$ unless it is accessed and compared with all the other events. In the following, we see how we can optimise these performance bottlenecks.

3.6 RCEP: Recomputational based CEP

In this section, we discuss optimisation techniques to cater for the bottlenecks of the baseline algorithm. For the sake of clarity, we restrict ourselves to the one dimensional case in this section. This allows us to present the generic optimisations techniques clearly, without loss of generality. Section 3.7 will present specific data structures for multidimensional events' storage and retrieval.

- For each event $e = (A, t) \in \mathcal{S}$, the size of the attribute set is $|A| = 1$. That is, there is just one attribute to match with the defined query predicates.
- The size of the window is equal to its slide: ω. This results in a *tumbling window*, where all the elements in the window expire at the same time.

The aforementioned restrictions mean we are able to concentrate on explaining the most crucial ideas. Both restrictions will eventually be removed in later sections; and our final algorithm still achieves the same upper bounds for both memory and runtime complexities. It is worth mentioning that the presence of only one attribute ($|A| = 1$) enables us to use a linear data structure for storing and retrieving events.

3.6.1 The Event Tree

To efficiently partition the events, we need an indexing structure to store events within a defined window. Given $\mathcal{S}$, we build a B+tree such that the keys of the tree are the events' attribute $A_x \in A$ and value contains the timestamp τ (**Step 1** in Algorithm 1). For brevity, we use this topology of keys and values. However, the reverse topology is also possible. The tree is created incrementally with the arrival of new events and the height of the tree is $\leq log\ n$, where n is the total number of events within a window ω. Fig. 3.3 shows an event tree of stock market events with price as an attribute. It is rudimentary to construct such an event

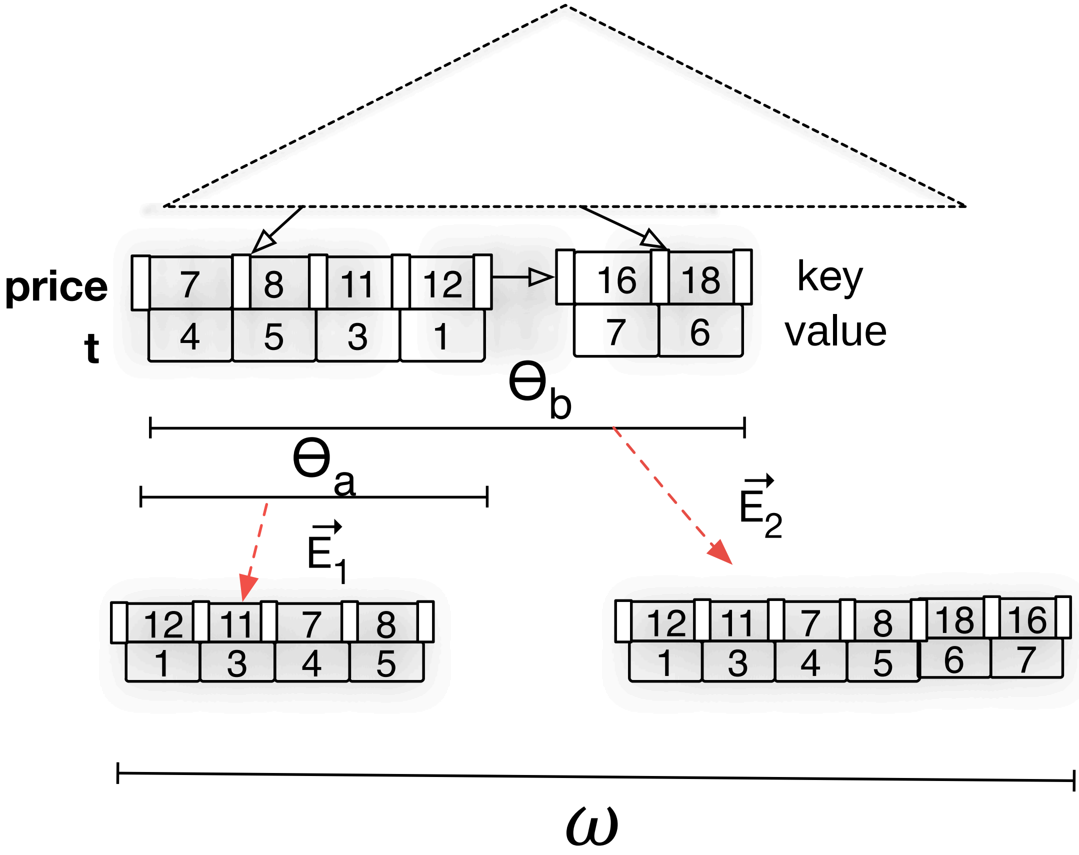


Figure 3.3: An Event Tree for the stock market events (price and timestamps) within a window ω

tree in $O(n \cdot log_{br} n)$ for n events with br as a branching factor (number of children at each node). We call such tree an event tree $\mathcal{T}_e$.

Given $\mathcal{T}_e$, our next task is to extract the partitioned set of events for each events sequence $\vec{E}$. This process is initiated when a trigger event arrives (**Step 2** in Algorithm 1), i.e. if the incoming event is compatible with $\vec{E_k}$. Using such a trigger event and the predicates in the CEP query, we need to create a set of range queries and extract the events from $\mathcal{T}_e$ that fall under them.

Let $\mathsf{Range}_{\Theta_i}(\vec{E_i})$ be a function which produces a range query $q = [min(\vec{E_i}), max(\vec{E_i})]$, using the minimum and maximum attribute values of events $e.A_x \in \vec{E_i}$ and binary relations from Θ_i. This range query can be executed over $\mathcal{T}_e$ in a traditional manner. That is, given an interval $[min(\vec{E_i}), max(\vec{E_i})]$ search down $\mathcal{T}_e$ for $min(\vec{E_i})$ and $max(\vec{E_i})$ values. All the leaf nodes between the ranges form the answer set and then sorting it over timestamps yields $\vec{E_j}$. For each range query, the tree search takes $O(2 \cdot log_{br} n)$ and reporting c covered points by the ranges takes $O(c)$, assuming that the leaf nodes are linked together. Sorting operation over timestamps, using the

traditional algorithms, can be performed in $O(c \cdot logc)$ to produce $\vec{E}_j$. The following example explains the construction and execution of such range queries.

Example 2:

Consider $\mathcal{T}_e$ in Fig.3.3 and $\mathcal{T}_q$ in Fig.3.2 (b). Now consider that a trigger event $e_i = (6, 9)$ ($price = 6$ and $t = 9$) arrives (note that we are detecting an inverted V-shaped pattern). We use e_i for the right-most leaf node $c/\vec{E}_3$ in $\mathcal{T}_q$; since it can complete a set of matches. Next, we need to extract the events for $b/\vec{E}_2$ that can be joined with events in $c/\vec{E}_3$. Hence, we create a range query $q_1 = [7, 18]$ (18 is the max price) while considering the binary relation for predicate $\theta_b = c.price < b.price$ and $e_i = (6, 9)$ in $c/\vec{E}_3$. The events covered by q_1 are shown in Fig. 3.3. We sort them by timestamp to produce $b/\vec{E}_2$. Finally we are left with $a/\vec{E}_1$. Hence, we create another range query using the predicate $\theta_a = a.price < b.price$. Since the maximum value of the price in $b/\vec{E}_2$ is $price = 18$, the boundaries of this range queries are as follows: $q_2 = [0, 18 - 1]$ (considering t an integer in this example). The events covered by q_2 are shown in Fig. 3.3 and sorting them by timestamp yields the final contents of $a/\vec{E}_1$.

The event tree solves our problem of filtering unnecessary events before the join process. Next, we see how to optimise the execution of joins between filtered events sequences.

3.6.2 Creating the Complex Matches

The set of range queries produces a set of events sequences for a query tree. To produce a set of matches, we need to employ *theta-joins* between them since we are dealing with both constant and variable predicates.

Problem Description

Considering the discussion in section 3.5 on theta-join execution problem, we propose a light-weight hybrid-join algorithm that solves this problem heuristically, providing significant performance gains over its counterparts. We first define the *dominance property* between events, which is the backbone of our algorithm.

Definition 10: Event's Dominance

Given two events $e, e' \in \mathcal{S}$ and a binary condition ϕ, e dominates e' iff $e.A_x \; \phi \; e'.A_x = true$.

The dominance property can easily be extended for the set of attributes A within an event and the set of binary conditions in Θ as described in Section 3.7.2. We denote that e dominates e' by $e \vdash e'$ and if not by $e \nvdash e'$. The dominance property also enables the *transitivity property*: given e, e', e'', if $e \vdash e'$ and $e' \vdash e''$ then $e \vdash e''$.

Our hybrid-join algorithm uses a sort-merge approach and utilises the dominance property to reuse the results of the previous join operations. Furthermore, we use memory compressed bitvectors to store the intermediate join results, thus reducing the memory footprint.

Hybrid-join Algorithm

Algorithm 2 shows our hybrid-join algorithm between two events sequences $\vec{E}_i$ and $\vec{E}_j$, where the events in $\vec{E}_i$ should have lower timestamps than in $\vec{E}_j$. For clarity, we consider $\vec{E}_i$ as an outer relation. Note that in order to join events in $b/\vec{E}_2$ with it self (as an inner relation) we use the same algorithm. The algorithm first sets up a list of bitvectors to indicate the position of joined events in $\vec{E}_j$ and $\vec{E}_i$: let $\{B_1, B_2, \ldots, B_u, \ldots\}$ be a collection of v-dimensional bitvectors. It is a map that will determine which element is joined with which other element. Consider that u corresponds to u^{th} element of $\vec{E}_i$ and v corresponds to v^{th} element of $\vec{E}_j$. This means that the vth bit of B_u, denoted by $B_{u,v}$, indicates whether a u element in $\vec{E}_i$ is joined with the vth of $\vec{E}_j$ using the predicates Θ. Furthermore, a variable *start* is also used to indicate the position to start the join operation on $\vec{E}_j$. For clarity, we first describe the algorithm from *lines 15 - 23* and then come back to *lines 5 - 12*. From *line 15*, the algorithm visits the events in $e' \in \vec{E}_j$ for an event $e \in \vec{E}_i$. It first checks the temporal ordering of events, i.e, timestamps $e.t$, and then the joined attribute $e.A_x$ (*line 17*). Note that the attributes of events are not sorted. If the temporal ordering and the attributes are matched between events, it sets the bit in B_u at that position (*line 18*). Otherwise if (i) the temporal order does not match, it stops visiting events in $\vec{E}_j$ for an event e – following the merge-sort algorithm; (*line 20*) (ii) else it unsets the bit in B_u at that position (*line 22*). Now we explain the operation of *lines 5 - 12*, where the dominance property is utilised to reuse the previously computed join operations.

For this task, the algorithm first checks the dominance property between e and e'' in $\vec{E}_i$ at positions u and $u+1$ (*line 7*). If a previously matched event e'' is dominated by e, then for e we do not have to visit all the elements in $\vec{E}_i$ just the ones that do not match with e''. Hence, using $B_{u,v}$ we only visit unset bits and compare only the attribute values of the events $e \in \vec{E}_i$ and $e' \in \vec{E}_j$ that do not match with e'' (*line 8 - 12*). The set bits are copied/reused for e and the algorithm avoids the recomputation of joins.

Example 3: Fig. 3.4 shows how Algorithm 2 works for two events sequences $\vec{E}_i$ (outer relation) and $\vec{E}_j$ with $\phi = \{>\}$ for the attribute A_x. The numbered operations in Fig. 3.4 are explained as follows. **(1)** Starting from the end of both sequences, event $(4, 40)$ with $e.t = 4$ in $\vec{E}_i$ is joined with three events in $\vec{E}_j$, e.g. $\{(5, 22), (6, 15), (7, 21)\}$. The bitvector B_u records the location of joined events in

Algorithm 2: Hybrid-Join Algorithm between $\vec{E}_i$ and $\vec{E}_j$

Input: Events sequences $\vec{E}_i$, $\vec{E}_j$, events in $e \in \vec{E}_i$ should follow events in $e' \in \vec{E}_j$, join predicate $e.A_x \ \phi \ e'.A_x$

```
1  Initialise a Bitvector B_u, where u, v represents the joined events in E_i and E_j respectively,
   and set all the bits to 0 ;
2  start ← |E_j|
3  for u ← |E_i| to 1 do
4      e ← E_i[u];
5      if u ≠ |E_i| then
6          e'' ← E_i[u + 1]
7          if e.a ⊢ e''.a then
8              B_u ← B_{u+1};
9              foreach unset bit at index v ∈ B_u and v ≤ start do
10                 e' ← E_j[v]
11                 if e.A_x φ e'.A_x then
12                     B_{u,v} ← 1;
13         else
14             start ← |E_j|;
15     while start > 0 do
16         e' ← E_j[start];
17         if e'.t > e.t AND e.A_x φ e'.A_x then
18             B_{u,start} ← 1;
19         else if e'.t < e.t then
20             break the while loop;
21         else
22             B_{u,start} ← 0;
23         start ← start - 1;
```

$\vec{E}_j$. **(2)** Since the event $(3, 50)$ at $e.t = 3$ in $\vec{E}_i$ dominates the event at $e.t = 4$ ($e.A_x > e''.A_x$) in the same events sequences, i.e. $\vec{E}_i$, we reuse the results of previously computed joins (in step (1)) using B_u. **(3)** Since the event at $e.t = 2$ does not dominate the event at $e.t = 3$ in $\vec{E}_i$, we restart the comparison process from the end of $\vec{E}_j$. **(4)** The event at $e.t = 1$ dominates the event at $e.t = 2$ in $\vec{E}_i$. Therefore, we reuse the joins evaluated in step 3 and continue the join procedure until the timestamps ordering fails.

The performance of the hybrid-join algorithm is highly dependent on the distribution of events' within the events sequences. We will see in Section 3.7.3 how to arrange such distribution to get the benefits of a hybrid-join algorithm. The complexity analysis of the algorithm is as follows: the outer loop in Algorithm 2 (*line 3*) in the worst case takes $O(|\vec{E}_i| \cdot |\vec{E}_j|)$, while it is clear to see that the extra space complexity is only $O(|\vec{E}_i| \cdot |\vec{E}_j|)$ **bits** of space, which is one of the main advantages of our algorithm.

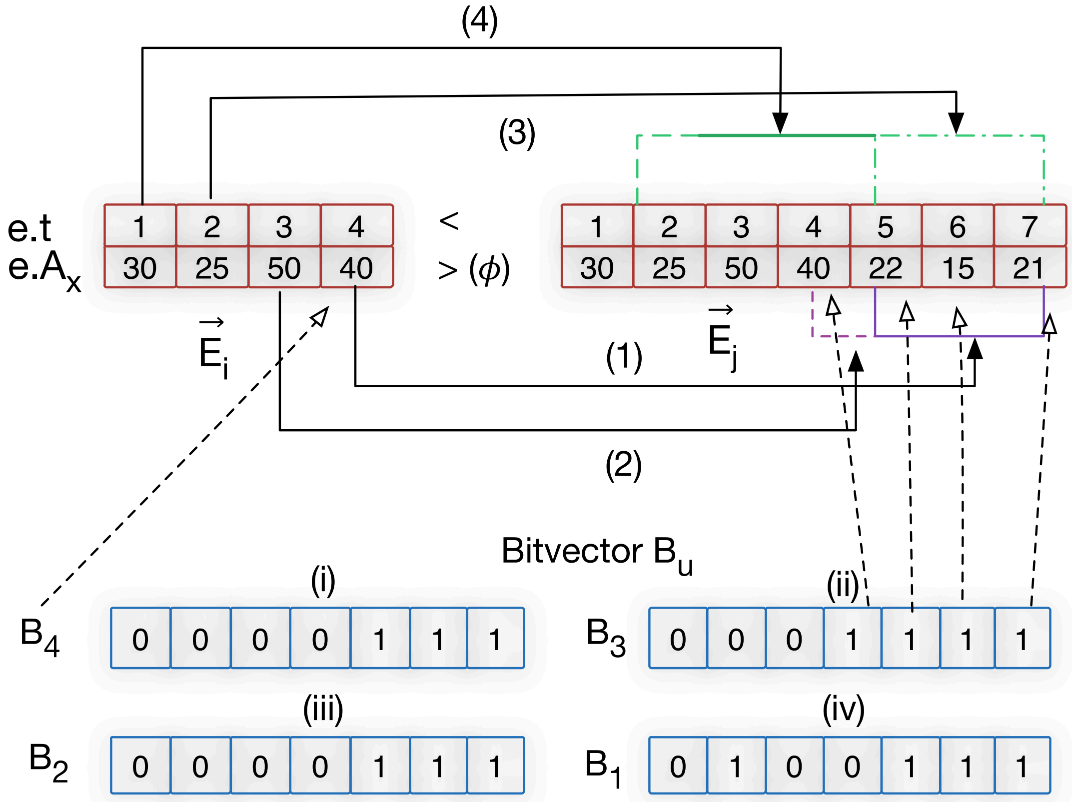


Figure 3.4: Hybrid-Join execution between two events sequences $\vec{E}_i$ and $\vec{E}_j$

Matches for the Kleene+ Operator

Algorithm 2 only shows the joins between two events sequences. However, for the Kleene+ operator ($p^+ \in P$) we need to create all the possible combinations of matches while following the temporal ordering (**Step 3** in Algorithm 1). That is, enumerating the powerset of events with p^+ bindings. To implement Kleene+ operator efficiently, we reuse the bitvector B_u while generating the binary representation of the possible matches using Banker's sequence [LvHS00]. That is, we check the number of set bits in each B_u. For b number of set bits in B_u, we need to create 1 to $2^b - 1$ matches. This means if we generate all binary numbers from 1 to $2^{|b|} - 1$, and translate the binary representation of numbers according to the location of bits in the bitvector B_u, we can produce all the matches for the Kleene+ operator.

Example 4: Now let's see an example of an inner join corresponding to a Kleene+ operator. This would mean joining elements of $\vec{E}_j$ with itself. For joining, we already start with those elements of $\vec{E}_j$ joined with $\vec{E}_i$ in the previous step. For example $(1, 30)$ belonging to $\vec{E}_i$ was joined with $\vec{E}_j = \langle(2, 25), (5, 22), (6, 15), (7, 21)\rangle$. Starting from that, we iteratively apply the same join algorithm till we exhaust the elements that need to be joined. For instance, we start with $(2, 25)$ and join

Algorithm 3: Matches for the Kleene+ Operator

Input: An event $e \in \vec{E_i}$ and its bitvector B_u with b number of set bits, sequence $\vec{E_j}$ with Kleene+ operator to be executed

```
1  max ← 2^b − 1;
2  k ← 0
3  for i ← 1 to max do
4  |  OutputMatchElement(e);
5  |  k ← 0;
6  |  for j ← i to 0 do
7  |  |  if j & 1= 1 then
8  |  |  |  k ← k + 1;
9  |  |  |  index ← NextSetBit(B_u, k);
10 |  |  |  OutputMatchElement(E_j[index]);
11 |  |  |  j >>← 1 ;                    // bitwise right-shift operation
```

with the rest $\vec{E_j} = \langle(5, 22), (6, 15), (7, 21)\rangle$. The algorithm will find that $(2, 25)$ can be joined with all of them in this example. In the next iteration, we start with elements that were joined with $(2, 25)$ i.e. now we try to find the joins with $(5, 22)$. In this case, all elements $\vec{E_j} = (6, 15), (7, 21)\rangle$ can be joined with. Note that just like dynamic programming, helped with dominance property, the results of all joins are stored and reused.

Remember that the objective is to output all the possible combinations for Kleene+ operator. For that, we proceed as follows: first, imagine join results are stored using a tree data structure: for example $(1, 30)$ as root, $(2, 25)$ as its child which in turn has $(5, 22)$ $(6, 15)$ and $(7, 21)$ as children, and so on. To output all the sequences our problem now resembles the classic problem of printing all paths from root to leaf. Second, once these paths are generated, we generate yet more combinations for each path using banker's sequence.

Banker's sequence will be applied on all the paths found during inner and outer joins. Let's assume for $\vec{E_i}$ $(1, 30)$ the path is represented in the form of bitvector $B = 0100110_2$ that represents joins location for $\vec{E_j} = \langle(2, 25), (5, 22), (6, 15)\rangle$ with set bits $b = 3$, the generated binary numbers from 1 to $2^{|b|} - 1$ are as follows: 001_2, 010_2 and 100_2, etc (Figure 3.5). Now equate 1 as take element at the specified bit location and 0 as do not take the element. Then using generated binary numbers, B and $\vec{E}$, we generate all the combinations of matched events. Interested readers can refer to Appendix B.1 for further details regarding the Banker's sequence and its mapping for a bitvector.

Figure 3.5 shows the enumeration of all the combinations for a $p^+ \in P$. For each event $e \in \vec{E_2}$, we need to output all the combinations of events $e' \in \vec{E_2}$ matched with e. Algorithm 3 first determines the maximum number of matches that can be generated from the number of set bits b in the bitvector B_u (*line 1*). That is, all

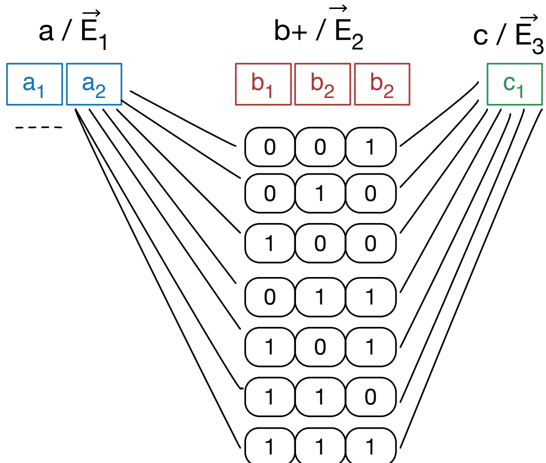


Figure 3.5: Execution of the Kleene+ operator using the Banker's sequence and generated binary numbers

events $e' \in \vec{E}_j$ that are matched with an event $e \in \vec{E}_i$. It then iterates over these numbers and uses their binary representation to determine the position of set bits (*lines 6,7*). For instance, if $i = 3$, then its binary representation 011 shows the last two bits are set (employing bitwise AND (&) operation). Using this information, it extracts the index of events in $\vec{E}_j$ using B_u (*line 9*). This index is used to output the events within a complex match without storing them in the memory.

Multiway Joins

We have so far restricted our discussion of joins over two events sequences but our join technique extends to multiple events sequences. For multi-way joins, not only do we need to decide the order of the joins but also how to employ the intermediate joined results. Building a cost-based query optimizer for the first task is a non-trivial task and several works, in particular, ZStream [MM09], have addressed this problem. These techniques can be applied to our approach. In fact, the pruned events from the set of range queries provide strict cardinality measures for join ordering in an adaptive manner. In this contirubution, we focus on the other side of the problem, i.e. optimising the two-way theta-joins and how to index and query relevant events from the event tree. The ordering of the joins is currently done by a static query planner, where joined relations for the right-most leaf node in the query tree result in the exact number of events to be joined. Furthermore, the intermediate results are shared using the bitvector B_u from Algorithm 2.

3.7 RCEP: General Recomputation Model

Here we provide the final RCEP algorithms by removing previous restrictions. Thus, we are now ready to lift the first restriction (Section 3.6) and events can have multiple attributes associated with a timestamp. Following this, we extend our aforementioned algorithms.

3.7.1 Multidimensional Events

Problem Description

An events stream $\mathcal{S}$ arrives in the system, where each event $e = (A, t)$ contains d dimensions ($d = |A| + 1$) and each dimension is of f bits. We can represent each event as a point in a *d-dimensional* space. To extract the required events for events sequence $\vec{E}$, we need to employ d-dimensional range queries. One common approach to this problem is to define a space hierarchy, where nearby multidimensional points fall into same partition and points that are far away are stored in different partitions. The resulting hierarchy can be stored in the form of multidimensional trees, such as KD-tree, R-tress and their many variants [Sam05]. Since CEP is update-intensive, such trees would not produce the required performance measures [QGT16] (as confirmed in Section 3.8). Furthermore, the performance of these structures deteriorates when data dimensionality increases [Bey+99]. An alternative approach is to embed the multidimensional events into 1D space and perform indexing and querying over the 1D space instead. This embedding is often achieved through fractal-based *space-filling curves* [Hil91; Mor66], in particular Peano-Hilbert [Hil91] and Z-order [Mor66] curves. These space-filling curves can ensure that events that are near in the data space are clustered together.

Basic Design of Indexing

We employ a Z-order curve for indexing due to its superiority in the context of its generation and monotonic ordering [NCB15]. Each point in a Z-order curve is represented by a unique number called a *Z-address*. A Z-address is calculated by *interleaving* the bits of all dimensions' values of a data point (or event). The Z-address of a d-dimensional event, with each dimension of f bits, contains df bits, which can be considered f *d-bit groups*. Given a Z-address with f d-bit groups, the first-bit group partitions the space into 2^d equal sized subspaces, the second-bit group partitions each subspace into 2^d equal sized smaller subspaces, etc. Fig. 3.6 (a) shows a two dimensional Z-order curve, where the data points are divided into four main quadrants. Given an event $e = (A, t)$, we create a Z-address z by bit interleaving in the following order: $\mathsf{Interleave}(t, A_1, A_2, \ldots, A_{d-1})$. The resulting

z is then stored in a compressed bitvector. Note that we use timestamp as the first and most important dimension for Z-addresses. This ensures the sorting of Z-addresses in a tree structure with timestamps as the most important dimension. In the rest, we interchangeably used the term event and Z-address.

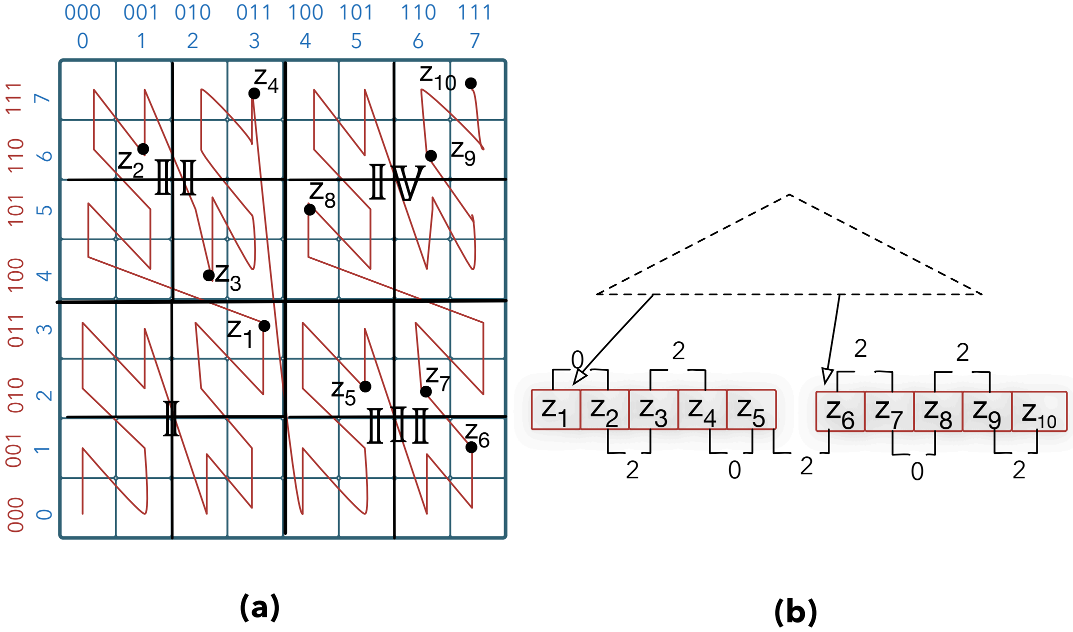


Figure 3.6: (a) Two-dimensional Z-order curve, (b) Event Tree indexing of Z-order Curve

3.7.2 Multidimensional Event Tree

Z-addresses are hierarchical by nature and the Z-order curve provides the two main properties: monotonic ordering and clustering of events [Lee+07]. The first property ensures that dominated events are placed before non-dominated ones, while the second property ensures the natural clustering of events with identical d-bit groups into regions called Z-regions. These describe the space covered by the Z-order curve's segment (see Fig. 3.6 with four main Z-regions). Let $min(r)$ and $max(r)$ are the two functions useed to produce the smallest and largest Z-addresses in a Z-region r, i.e. $min(r)$ produces a Z-address which contains the smallest values in all the dimensions in r, while the Z-address produced by $max(r)$ contains the largest values in all the dimensions in r given. Our first goal is to maintain similar Z-addresses within a Z-region. This goal stems from the fact that we need block-based dominance tests over Z-regions such that we can efficiently assert if a block of events is dominated by others (or not) and whether it can be joined to produce matches.

The seminal work of Orenstein and Merett [OM84] or the more recent UB-tree [Ram+00a] stores linearly ordered Z-addresses (keys) in a B+tree: keys are addresses of Z-regions enclosed in leaf nodes with $[min(r), max(r)]$ as boundaries. Since Z-addresses are monotonically ordered, their insertion in the UB-tree follows the traditional B+tree algorithm. This means Z-regions can be of any size and shape after the insertion of new Z-address. For example, in Fig. 3.6 (b), a leaf node of size 5 encapsulates Z-addresses for different Z-regions. One solution in this context would be to manually maintain the $[min(r), max(r)]$ interval for each leaf-node [Lee+07]. However, this requires continuous maintenance of $min(r)$ and $max(r)$ with new updates and their propagation to the parents/ascendant nodes. Such maintenance can be expensive for an update intensive application. Considering this, we employ an incremental approach, where the multiple Z-regions within a single leaf node are incrementally maintained during insertion and are retrieved as clusters during the range query evaluation.

Insertion in Event Tree

The insertion process tracks the distance between neighbouring Z-addresses in a leaf node. The distance is defined through a function $\mathsf{Dis}(z_i, z_j)$, which takes two Z-addresses, z_i and z_j, and determines the common most significant bits (MSB) between them. For example, $\mathsf{Dis}(z_i, z_j) = 4$ for $z_i = \mathbf{1011}0001$ and $z_j = \mathbf{1011}1010$. The complete insertion process is as follows. Each incoming event is first mapped to a Z-address and a DFS (Deep First Search) is executed to locate the appropriate leaf node. This leaf node (i) might have free space or (ii) may result in an overflow. For the first case, we insert the Z-address and calculate (subsequently store) the distance from its neighbours in the same leaf node. For the second case, we find the split point based on the maximum distance between Z-addresses. That is, we employ a binary search over the Z-addresses in a leaf node to determine the location of the Z-address which has the greatest distance from its neighbour in the same leaf node. This leads to two disjointed Z-regions (leaf nodes).

A leaf node can still contain Z-addresses of disjointed Z-regions after the insertion process. These Z-regions will be identified during range query processing. Although a leaf node can be divided into multiple disjoint Z-regions during the insertion phase, we pack these points together to minimise the memory and CPU overheads for higher throughput. Fig. 3.6 (b) shows that the distance between z_1 and z_2 is 0 since they belong to two different Z-regions and have no common MSB. However, for z_2 and z_3 the distance is 2 for the common 2-bits.

Range-Query Processing

The creation of a set of range queries over the event tree follows a similar procedure to that described earlier. Given a set of predicates Θ and the events sequences, we create a set of range queries (bounding boxes), where each is of the form $q = [q^l, q^h]$; $q^l = (t_1, A_1, \cdots, A_{d-1})$, $q^h = (t', A'_1, \cdots A'_{d-1})$ and $\forall_i A_i \leq A'_i \wedge t \leq t'$ (l means lowest and h means highest in the query range). To aid the range query construction, we also maintain an inverted index to keep track of the lowest and highest values of each dimension in the event tree. The bounding box boundaries q^l and q^h are mapped to the Z-addresses z^l and z^h using bit interleaving. All the Z-addresses in the event tree $\mathcal{T}_e$ enclosed by z^l and z^h form the answer set. The search procedure in $\mathcal{T}_e$ for z^l and z^h is accomplished by fast bitwise operations. Note that due to the nature of Z-order curves that sometimes goes outside the bounding box, there can be some *false positive* points within the defined ranges. While constructing the Z-regions, we will also remove those false-positive points. After the execution of a range query, we get a list $\{(z_1, dis), (z_2, dis), \ldots, (z_c, dis)\}$ of Z-addresses, each accompanied by the distance measured from its neighbours. Our task is to cluster the Z-addresses according to their Z-regions or common d-bit group so that the Z-regions are smallest among the possible splits. To accomplish this, we employ a sequential scan over the list of Z-addresses and split them into Z-regions where there is a difference of common d-bit group between them. Since Z-addresses are monotonically ordered, the Z-addresses that belong to the same Z-regions are located next to each other in the list.

Example 5: Consider Fig. 3.6 (b) that both leaf nodes fall under a range query. We sequentially scan each leaf node: starting from the first node, the distance between z_1 and z_2 is 0, while z_2 and z_3 is 2. Hence, this results in two disjoint Z-regions, where the first contains $\{z_1\}$ and the second contains $\{z_2, z_3, z_4\}$. Moving forward, the distance between z_4 and z_5 is 0, hence a new Z-region $\{z_5\}$ is created. While moving to the next node, we check the distance between z_5 and z_6, and z_6 and z_7. If the distance is the same, i.g. 2, we insert z_6 and z_7 with z_5. Finally, after going through all the elements in the leaf nodes, we will have the following Z-regions: $\{z_1\}$, $\{z_2, z_3, z_4\}$, $\{z_5, z_6, z_7\}$ and $\{z_8, z_9, z_1\}$.

3.7.3 Joins between Z-addresses

We first describe the dominance property between two Z-addresses and then form the notion of Z-region dominance. Let $g^t_j(z)$ is a function which takes a Z-address and returns a bitstring by extracting t MSB bits for every j bits skipped. For example, for $z = \mathbf{1}10\mathbf{0}11$, $g^1_2(z)$ takes all the first MSB jumped by 2 bits and $g^1_2(z) = \mathbf{10}$.

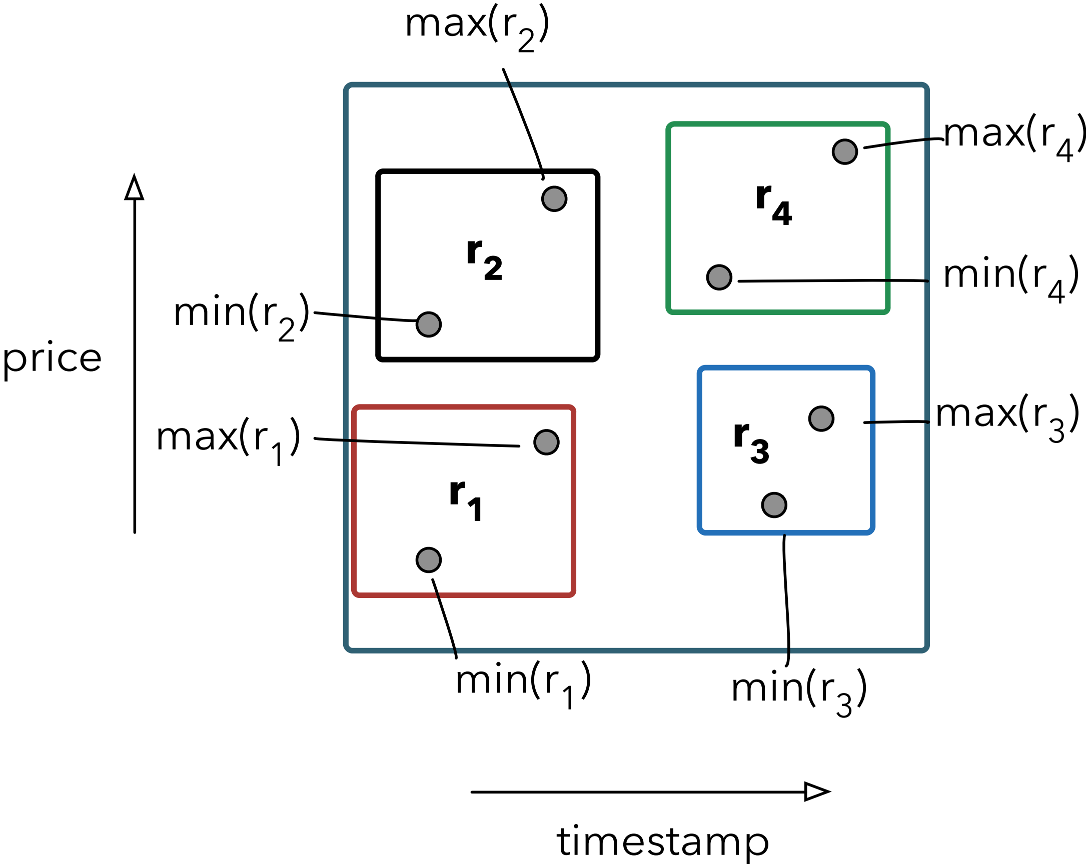


Figure 3.7: The dominance test between different Z-regions

Definition 11: Z-address Dominance

Given two Z-addresses z and z' and a set of binary relations $\{\phi_1, \dots \phi_d\}$ from the set of variable predicates Θ^v in Q over d-dimensional events (Z-addresses), z dominates z' ($z \vdash z'$) iff $g^1_{d-1}(z)\ \phi_1\ g^1_{d-1}(z') \wedge g^2_{d-1}(z)\ \phi_2\ g^2_{d-1}(z') \cdots \wedge g^d_{d-1}(z)\ \phi_d\ g^d_{d-1}(z')$. Note that if there is domination in all dimensions then it means that the z-value also dominates.

The Z-address dominance also follows the transitivity property, i.e. if $z \vdash z'$ and $z' \vdash z''$ then $z \vdash z''$. Based on this, we introduce a block-based dominance test over Z-regions. Since Z-regions cluster similar points together, the dominance test at the boundaries of the Z-regions i.e. $min(r)$ and $max(r)$, can provide proof of the joins. Note that $min(r)$ and $max(r)$ provide Z-addresses containing the maximum and minimum values of all the dimensions respectively in a region r.

lemma 1

Given two Z-regions (r and r') and a set of binary relations $\{\phi_1, \ldots \phi_d\}$ from the variable predicates Θ^v in Q over d-dimensional events (Z-addresses), r dominates r' ($r \vdash r'$) iff

1. $\forall_i g^i_{d-1}(max(r))\ \phi_i\ g^i_{d-1}(min(r'))$ if $\phi_i = \{<, \leq\}$
2. $\forall_i g^i_{d-1}(min(r))\ \phi_i\ g^i_{d-1}(max(r'))$ if $\phi_i = \{>, \geq\}$

lemma 2

Given two Z-regions (r and r') and a set of binary relations from Lemma 1, r partially dominates r' ($r \sim r'$) iff Lemma 1 does not hold and only if the following conditions hold:

1. $\forall_i g^i_{d-1}(min(r))\ \phi_i\ g^i_{d-1}(max(r'))$ if $\phi_i = \{<, \leq\}$
2. $\forall_i g^i_{d-1}(max(r))\ \phi_i\ g^i_{d-1}(min(r'))$ if $\phi_i = \{>, \geq\}$

To keep the discussion brief, we prove Lemmas 1 and 2 in appendix B. Herein, we outline their intuition. The transitivity property and monotonic ordering of Z-addresses in Z-regions dictate that if conditions 1 and 2 are satisfied regarding $max(r)$ and $min(r)$ points, then $\forall z_i \in r$ also follows the same properties. However, in this case, we have to check the binary conditions ϕ for each dimension to select the appropriate boundaries. If Lemma 1 does not hold then it means that two Z-regions either overlap, or that they do not. Lemma 2 checks if two Z-regions overlap and hence some events that can be joined. If Z-regions do not overlap, we are sure that no events can be joined between them. From Lemma 1 and Lemma 2 we have the following theorem.

theorem 1

Given two Z-regions r and r' and a CEP query Q, if Lemma 1 holds then $r \vdash r'$ and all the events in r can be joined with all the events in r'. Otherwise, if only Lemma 2 holds then $r \sim r'$ and some events in r can be joined with some other events in r'. If both Lemmas 1 and 2 do not hold then $r \nvdash r'$ and no events in r can be joined with any event in r'.

Example 6: Consider two predicates $\theta_1 = a.price < b.price$ with $\phi = \{<\}$ and $\theta_2 = a.time > b.time$ with $\phi = \{>\}$. Using these predicates and binary conditions, we determine the dominance property between Z-regions in Fig. 3.7. (1) Z-regions $r_3 \vdash r_2$ (subsequently $r_2 \nvdash r_3$) since all the events in r_3 have prices less and timestamps greater than all the events in r_2. This compiles to Lemma 1 (2) r_3 partially dominates r_4, since for r_3 all the events have lower prices but some

have equal or greater timestamps. Similarly r_1 partially dominates r_4. This compiles to Lemma 2.

Using Theorem 1, we present the join algorithm for the sets of Z-regions. Let R and R' be two sets of Z-regions, generated from the lists Z and Z' respectively using the range queries as described in Section 3.7.2.

Algorithm for Joining Z-regions

Based on Theorem 1, Algorithm 4 presents how the events within the set of Z-regions R and R' are joined in a batch manner. It takes bitvector B_u as in Algorithm 2, to store the intermediate joined results. Furthermore, it uses the lists of Z-addresses Z and Z' (for result construction), that is covered by the range queries and are used to produce the set of Z-regions R and R'. Finally, it initialises pointers $start_r$ and end_r to keep track of the bits to toggle in B_u . The algorithm iterates over Z-regions $r \in R$, and for each region it evaluates the three cases in Theorem 1 over Z-regions $r' \in R$ (*lines 5-13*). For Lemma 1, it sets all the bits using the pointers of each region and using bit-level parallelism in the function setBits (a function that set the bits between two given positions in a bitvector)(*lines 7,8*). For Lemma 2, it simply uses Algorithm 2 since there can be some events in each Z-region that can be joined (*lines 9,10*). Finally if Lemmas 1 and 2 do not hold, since no events in the Z-regions can be joined, it unsets the bits in B_u using the aforementioned pointers (*lines 11, 12*). Note that the unsetBits function is employed for brevity only, In general, we do not have to unset the bits after the initialisation of a bitvector.

Algorithm 4: Join between the set of Z-regions R and R'

```
Input: Sets of Z-regions R, R', Bitvector B_u, List Z and Z' of all the points in R and R'
       respectively
1  Initialise a Bitvector B_u, where |B_u| = |Z| and each B_{u,v} = |Z'|;
2  Initialise List pointers start_r, end_r, start_r', end_r'
3  for each r ∈ R do
4      end_r ← |r|
5      for each r' ∈ R' do
6          end_r' ← |r'|
7          if Lemma 1 holds then
8              setBits(B_u, start_r, end_r, start_r', end_r')
9          else if Lemma 2 holds then
10             Use Algorithm 2 to determine the joins between Z-addresses in r and r';
11         else if Lemmas 1 and 2 do not hold then
12             unsetBits(B_u, start_r, end_r, start_r', end_r')
13         start_r' ← end_r' + 1
14     start_r ← end_r + 1
```

Complexity Analysis of Join Algorithm The performance of joins between the set of Z-regions R and R' depends on the predicates and the data distribution. For all types of predicates and data distribution combinations, we can always identify 3 cases, which are named and explained as follows:

- Dominated regions: A type of data distribution that will have many regions dominating other regions.
- Skipped regions: Another type of data distribution that will have many large regions not satisfying the predicates and thus joining them can be skipped.
- Independent: Finally, there will be data distributions with mostly overlapping regions. There will still be some dominating regions, which can be batch joined. Other points will still need to be pairwise joined.

1- Dominated Regions data: In this case, only a few Z-regions within R and R' have to be pair-wise joined (using Algorithm 2). The majority of Z-regions in R and R' would follow the dominance property and will be joined in a batch manner, i.e. *lines 7-8* in Algorithm 4. Consider if only $r_1 \in R$ and $r_2 \in R'$ have to be pair-wise joined, then the runtime complexity is $O(|r_1||r_2| + |\bigcup R \setminus \{r_1\}|)$, where $|\bigcup R \setminus \{r_1\}|$ is the cost of setting bits in the remaining Z-regions of R that are joined in a batch manner with Z-regions in R'.
2- Skipped Regions data. In this case, most of the Z-regions in R' are not dominated by R, nevertheless, they will be skipped in a batch manner, i.e. *lines 11-12* in Algorithm 4. Using the same reasoning of the Dominated regions data points, this case follows the same run-time complexity measures.
3- Independent data. For this case, m out of n Z-regions in R have to be pairwise joined with the Z-regions in R'. The remaining can either pass or fail the dominance test, i.e. *lines 7-8* or *lines 11-12* in Algorithm 4. Hence, the run-time complexity is as follows: $O(|\bigcup_m R||\bigcup_m R'| + |\bigcup_m R \setminus \bigcup_{n-m} R|)$. In the worst case, it follows the run-time complexity measures of the hybrid-join algorithm, i.e. $O(|\bigcup R||\bigcup R'|)$ runtime measures and $O(|\bigcup R||\bigcup R'|)$ bits of space.

3.7.4 Handling Sliding Windows

We are now ready to lift the final constraint of the sliding windows. That is, the size of the window ω and slide s can be of different sizes. This means with the arrival of an event, we need to re-calculate the boundaries of the new window after applying the sliding factor. This can result in some events in the event tree $\mathcal{T}_e$ to be outside the boundary of the window and hence to be deleted. We propose CPU and memory friendly methods for this task. Let $\mathsf{ValidTime}(\mathcal{T}_e)$ be a function which provides the validity time of $\mathcal{T}_e$, i.e. the insertion time of the newly arrived event plus the window size.

Memory Friendly Method

The first method deletes the expired events as soon as possible. It consists of the following two steps: (i) if $\mathsf{ValidTime}(\mathcal{T}_e) > s$, create a range query over $\mathcal{T}_e$ to determine the expired events; (ii) delete events covered by the range query. This operation may trigger *re-distribution* and merge operations over $\mathcal{T}_e$. However, older events are deleted as soon as possible.

CPU Friendly Method

For our second technique, we circumvent the cost of re-distribution and merge operations by resorting to a technique based on the logarithmic method [BS80]. This technique is based on two event trees $\mathcal{T}_e^1$ and $\mathcal{T}_e^2$ and is described as follows.

1. At the beginning, we start with an empty $\mathcal{T}_e^1$. For now, the insertion and range query processing is only applied over $\mathcal{T}_e^1$.

2. With the arrival of new events, we check if $\mathsf{ValidTime}(\mathcal{T}_e^1) > s$. If so, it means some expired events are present in $\mathcal{T}_e^1$. We keep these events and initiate $\mathcal{T}_e^2$. For now, the insertion will take place in $\mathcal{T}_e^2$ and range query processing in both $\mathcal{T}_e^1$ and $\mathcal{T}_e^2$. The temporal predicates in the range query make sure that expired events in $\mathcal{T}_e^1$ are not selected.

3. With the arrival of new events, if $\mathsf{ValidTime}(\mathcal{T}_e^1) > \omega$ it means that all events in $\mathcal{T}_e^1$ are outside of the window. Hence, we discard the older event tree and all operations are executed over $\mathcal{T}_e^2$.

4. With the arrival of further new events, if $\mathsf{ValidTime}(\mathcal{T}_e^2) > s$, we recycle $\mathcal{T}_e^1$ and use it as secondary storage. Hence, we keep rotating between these two event trees.

This second technique avoids the rebuilding cost of an event tree in the streaming settings and results in a slight increase in memory by keeping the older events, which still costs far less than maintaining the partial matches.

3.7.5 Optimising Z-address Comparison

We also used some optimisation techniques to save the cost of the bit interleaving and comparison of Z-addresses during insertion, as well as range query processing over an event tree. We do not provide details here, but full details are in the appendix B.4. The idea is to avoid the cost of decoding the Z-values for comparison and thus we used algorithms that can directly compare Z-addresses.

A second optimisation was used to address the nature of the Z-order curve: it contains jumps and can go out of the bounding box of a range query. Hence when calculating the range queries, we can have some false positive points, i.e. Z-addresses that are not part of the result, within the answer set. Please refer to [OM84; Ram+00a] for the detailed discussion on this problem. We used algorithms to calculate such boundaries without going through the process of bit interleaving and de-leaving called them NextJumpIn (NJI) and NextJumpOut (NJO).

3.8 Experimental Evaluation

3.8.1 Setup and Methodology

Experimental Infrastructure: Our proposed techniques have been implemented in Java and our system is called RCEP. All the experiments were performed on a machine equipped with an Intel Xeon E3 1246v3 processor, 32 GB of memory and 250 GB PCI express SDD. It runs a 64-bit Linux 3.13.0 kernel with Oracle's JDK 8u05. For robustness, each experiment was performed several times and we report median values.

Datasets: We employ both real and synthetic datasets to compare the performance of our proposed techniques.

Synthetic Stock Dataset (S-SD): We use the SASE+ generator, as used in [Agr+08], to produce the synthetic dataset. Each event carries a timestamp, company-id, volume and the price of stock. This dataset enables us to tweak the selectivity measures $\frac{\#ofMatches}{\#ofevents}$ to evaluate the performance of the systems at different workloads. In total, the generated dataset contains 1 million events.

Real Credit Card Dataset (R-CCD): We use a real dataset of credit card transactions [AAP17; Art+17b]. Each event is a transaction accompanied by several arguments, such as the time of the transaction, the card ID, the amount of money spent, etc. The total number of transactions in this summary dataset was around 1.5 million.

Synthetic Correlation Dataset (S-CD): To compare our join techniques, as described in Section 3.7.3, we generated a synthetic dataset [GB01] based on

dominated and independent distributions. The data dimensionality (or joined relations) varied from 2 - 6 and data cardinality from 10K to 100K.

Physical Activity Monitoring Dataset(PAM): We use a real dataset of physical activity monitoring [Pop+17]. It contains data of 18 different physical activities (such as walking, cycling, playing soccer, etc.) with inertial measurement sensors and a heart rate monitor.

Queries: We consider the 3 different Queries as defined at the beginning of this chapter (Section 3.2). Each of them is executed on the different datasets to compare the performance of our system.

For our evaluations, we specifically chose an additional variation of query 2 (credit card) which has a constant predicate on $a/\vec{E_1}$ to control the selectivity of starting match a, which in turn also controls the produced matches. For example, we used $a.amount\%17 = 0$ which controls the starting match and in turn determines the number of final matches. We used this to provide a worst case scenario for our batch processing based algorithm. Our hypothesis is that a low number of starting events $(a/\vec{E_1})$ penalise our lazy approach. Thus, in the results, we call the original query 2 as the case with *high selectivity* which equates to a high number of matches in the data. We call the above variation of query 2 as the case with *low selectivity*, which results in low number of matches.

Methodology: We compare our CEP system with ZStream [MM09], SASE+ [Agr+08] and the open source industrial system Apache Flink [Fli]. Note that CEP module of Flink is based on the optimizations proposed by SASE++ [ZDI14a]. All of these systems support skip-till-any-match and Kleene+ operator. Note that, we do not compare CET [Pop+17] due to the dissimilarity in semantics and its exponential memory complexity. The executional model of Apache Flink is based on the NFA design of SASE++ [ZDI14a] and CEP queries in Flink were expressed as Flink operators. These operators are processed in parallel by Flink, however, the events' order is maintained. Unless otherwise specified, all experiments use a slide granularity $s = 1$ and we employ the CPU friendly method of sliding windows for our system. To demonstrate the effectiveness of our join evaluation techniques, denoted as HJoin in figures, we compare it with the projection-based IEJoin [Kha+17] and nested-loop join (NLJoin). To achieve a fair comparison, we have implemented these join techniques on the top of our system. In a nutshell, IEJoin sorts events in a set of lists according to all the join relations and then determines the intersection of such lists; and the NLJoin is a standard join with nested loops. Note that under certain parameters, SASE+, Apache Flink and ZStream do not produce results for several hours. Thus, we discontinued the results in the charts for these systems, for such cases.

Metrics: We measure three standard metrics common for CEP systems: throughput, memory requirements and detection latency [Agr+08; ZDI14a; Pop+17; RLR16].

Throughput is defined as the number of events processed per second by a CEP system. The memory requirements were measured by considering the resident set size (RSS) in MBs. RSS was measured using a separate process that polls the `/proc` Linux file system once a second. Note that this method did not interfere with the overall timing results, from which we concluded that it did not perturb the experiments. Detection latency is the time between the occurrence of a complex event (marked by the arrival of the last event completing the match) and its detection and reporting by the CEP system. As described earlier, we use the selectivity measures $\frac{\#ofMatches}{\#ofEvents}$ to test different workloads. It is controlled by changing the predicates and window sizes in the CEP queries. We varied the selectivity, window size and the number of matches for our experiments to test the CEP systems' under a heavy workload.

3.8.2 Performance of Indexing and Join Algorithms

Comparing MultiDimensional Indexing

Herein, we investigate how an event tree compares with the traditional multidimensional indexing techniques. As the R-tree is commonly believed to offer competitive efficiency on multidimensional data and queries, we compare it with our event tree. We use stock data streams with an event of four dimensions from our use case; we change the number of events within a window; and we keep track of the total time spent in the insertion of events and processing range queries for each incoming event. Fig 3.8 presents the results. Since the proposed event tree is based on the linear indexing data structure (single dimensional search), it outperforms the heuristics structure R-tree (multi-dimensional indexing and search) considerably: by a factor of 2 over small window sizes and one order of magnitude for large window sizes.

Effect of Data Dimensionality

Herein, we investigate different join execution techniques, starting with the question of how data dimensionality (joined relations) effects different join techniques, i.e., HJoin, IEJoin and NLJoin. The answer to this question will showcase two of our optimisation techniques: (i) how virtual Z-addresses reduce and maintain the CPU cost in HJoin for the different number of dimensions; (ii) how the IEJoin algorithm has a strong dependence on the number of used dimensions or joined relations. Figs. 3.9 (a) and (b) show the total execution time of the two-way inequality joins for dominated regions (as explained in Section 3.7.3) and independent distribution

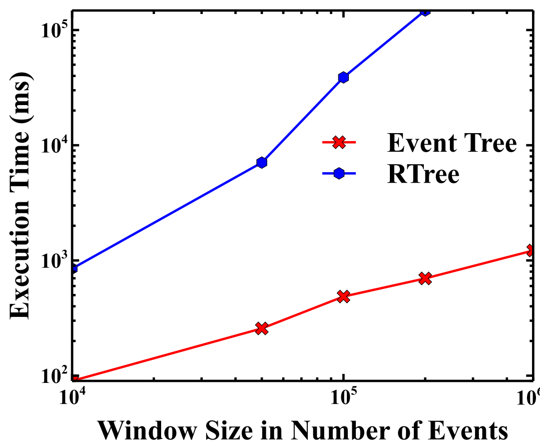


Figure 3.8: Insertion and Query Time: Comparative Analysis of Event Tree and RTree

datasets, while data cardinality is fixed at 50K in each relation. Note that we tested our algorithms on dominated regions and independent datasets because such distributions can impact the performance of any algorithm based on the dominance property. The results from skipped regions data distribution is similar to the one of dominated regions, since both have a similar complexity as discussed before. HJoin, owing to the virtual Z-addresses, effective space pruning capability and block-based dominance tests, performs better than IEJoin when the data dimensionality is increased. HJoin performs better on the dominated regions dataset since a lot of Z-regions pass the dominance test and hence events are joined in batches. Furthermore, since virtual Z-addresses employ fast CPU friendly bitwise operations to compare two events, its performance does not deteriorate with the increase in the number of dimensions. In contrast, IEJoin requires indexing of each dimension and its cost increases with the dimensions, thus becoming too expensive for a larger number of dimensions and is even slower than NLJoin. NLJoin has no effect on changing the dimensions since it is a pair-wise join technique.

Effect of Data Cardinality

How do different join techniques scale by varying the data cardinality for different types of data distributions? This further measures how well HJoin works with the overheads of recomputation by evaluating joins over batches of events. Figs. 3.9 (c) and (d) show the executional time of two-way inequality joins against the data cardinality (10K to 100K) while data dimensions (joined-relations) are fixed at 3 -a value taken from our use case. The execution times of all the algorithms rise with the increase in data cardinality. However, HJoin is less costly than its competitors for

both data distributions. The IEJoin algorithm is even slower than NLJoin because it uses a lot of time to create the indexes between the joined relations. That is the cost of sorting both relations according to each dimension, producing a permutation of indexes between each sorted dimension. Such a high index creation time outclasses its benefits for the streaming settings. Furthermore, both IEJoin and NLJoin show similar performance measures for both types of data distribution, since they do not employ any data-related heuristics. In contrast, the HJoin algorithm is less costly for the dominated regions dataset. This is because a large number of Z-regions pass the dominance and hence are joined in batches using bitwise operations. HJoin is slightly more costly for the independent dataset, relative to dominated regions dataset, for large data cardinalities. This is due to the increase in the number of pair-wise joins since a fewer Z-regions either pass or fail the dominance test. In any case HJoin performs better than NLJoin and IEJoin over different distributions.

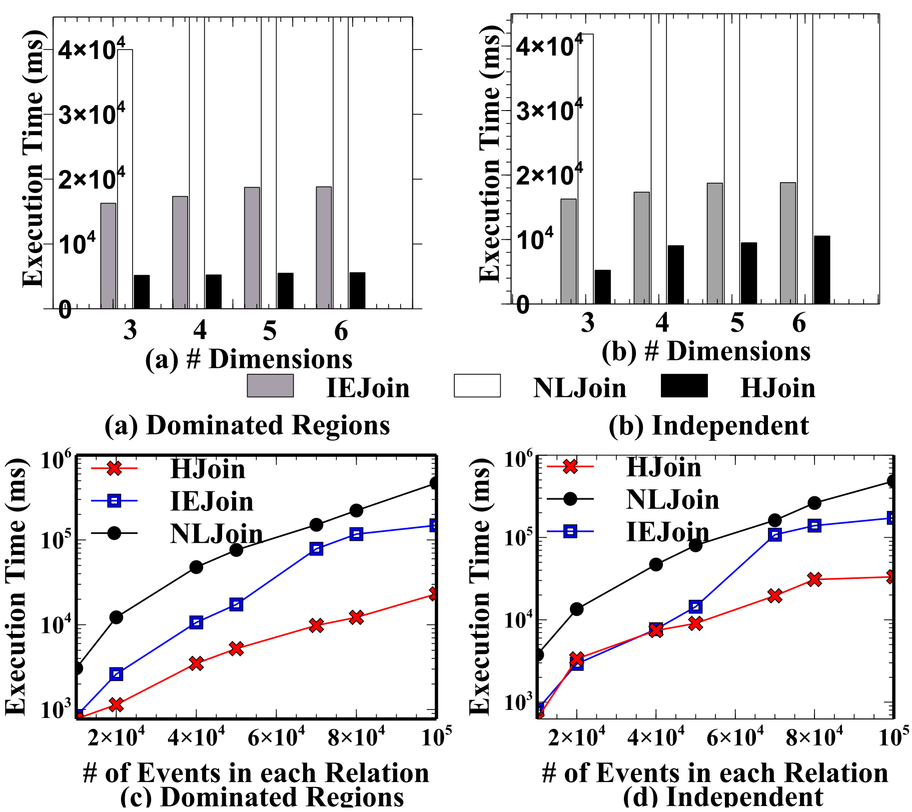


Figure 3.9: Analysis of Join Algorithms over different Data Distributions

3.8.3 Performance of Sliding Windows

In Section 3.7.4, we presented memory and CPU efficient methods for evaluating sliding windows. Herein, we showcase actually how much better the CPU friendly method is in terms of CPU cost. Fig 3.10 shows the performance of these two

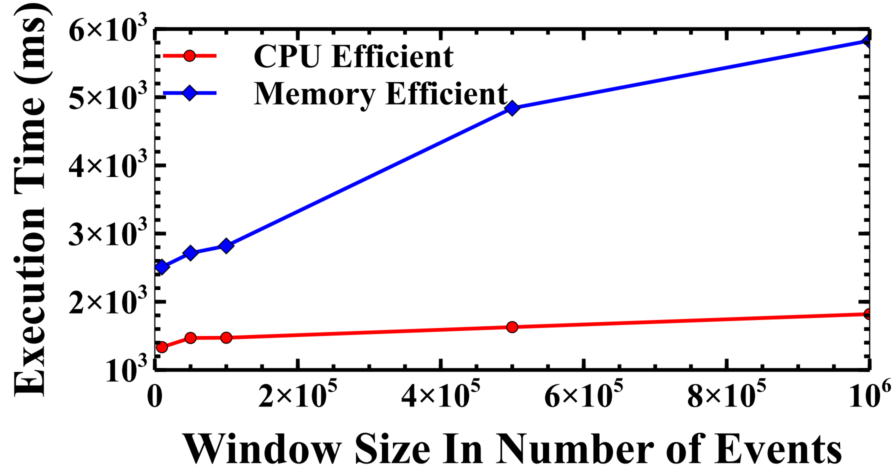


Figure 3.10: Sliding Window Comparison

methods over different windows size while considering the slide granularity of $s = 1$: since it is the worst case behaviour in terms of per event cost due to the single insertion and single eviction for each incoming event.

With small windows, the size of the event tree is small. Hence, the redistribution cost of leaf nodes after insertion and deletion is small for both memory and CPU efficient techniques. However, there is an extra cost of range query processing for the memory efficient technique to determine which events are outside the window and to be deleted. This cost increases with the increase in the window size. The redistribution of the event tree with frequent deletions becomes so expensive that it is outperformed by CPU efficient techniques by a large margin. Hence, the overheads of memory efficient techniques, as to be expected, are considerably higher than CPU efficient technique.

3.8.4 CEP Systems' Comparison

Performance in terms of Memory

The first question we investigate is, what are the benefits, in terms of memory cost, for only storing the events within a window. Fig. 3.11 answers such question. As expected, the increase in the selectivity measures (or window size) results in a large number of partial matches for traditional systems, and as a consequence, a larger utilisation of memory. In all cases RCEP performs best in terms of memory consumption. SASE+ and ZStream consume high memory due to exponential partial matches generated. Surprisingly Apache Flink performs worst in terms of memory. We found that it reuses instantiated runs instead of deleting older runs

for optimization purpose to reduce pressure on garbage collection. This enhances the CPU performance at the cost of higher memory consumption. In fact this reminds us about our CPU friendly method. In contrast, our recomputation-based system, namely RCEP, scales almost linearly to the number of events within a window. That is, for high selectivity, more events are selected to produce matches, hence the memory requirement increases almost linearly to the number of events and not to the number of partial matches. Additionally recall that, for memory efficiency, we used memory efficient bitvectors to store intermediate join results. The results confirm the aforementioned issues of storing partial matches and are aligned with our design principles.

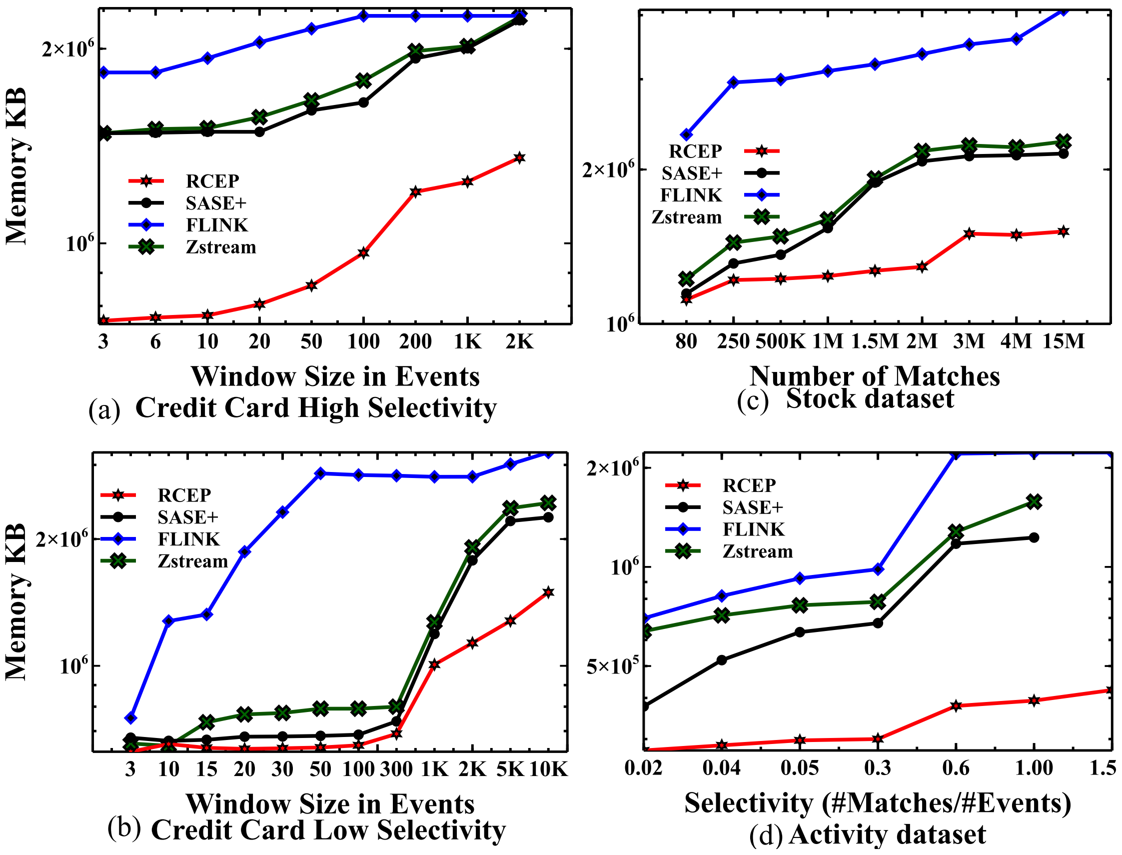


Figure 3.11: Memory Performance.

Throughput Performance

Next, we investigate from the point of view of the throughput: How do the recomputation based systems perform compared to systems that incrementally process partial matches. Figs. 3.12 shows the relative performance of the CEP systems over all the datasets and with different selectivities. We can see that, in general, RCEP has much higher throughput than Flink, ZStream and SASE+ As a

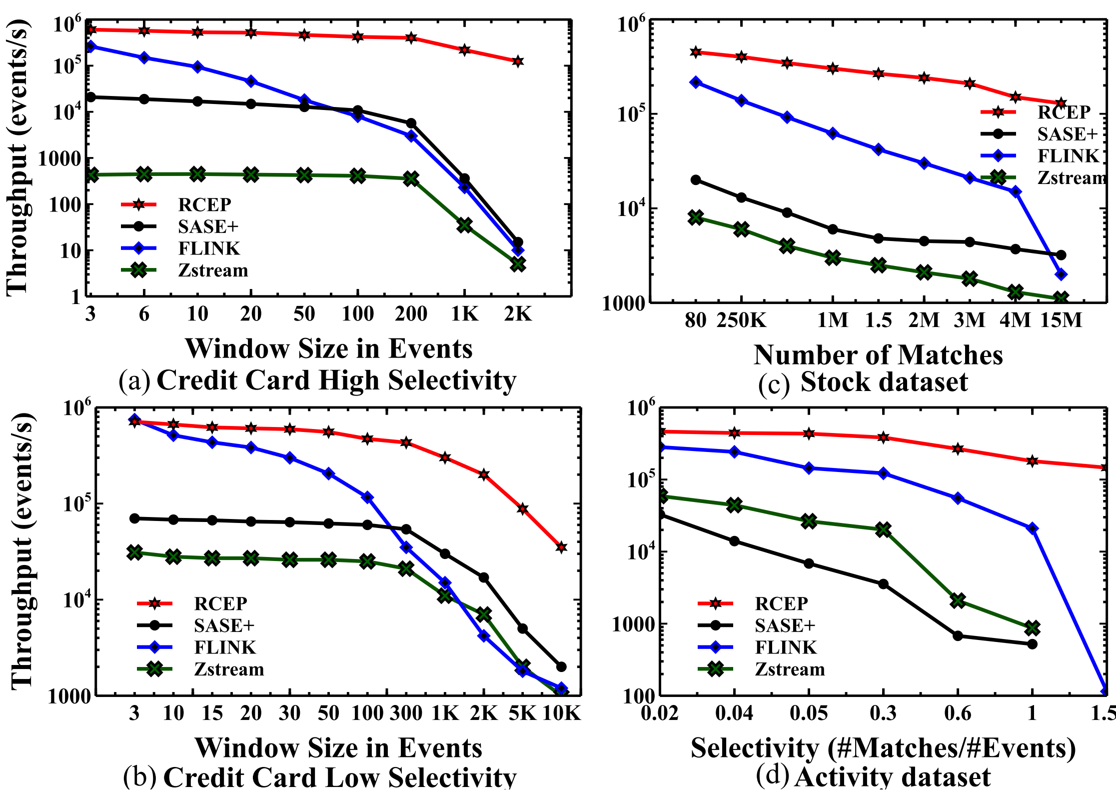


Figure 3.12: Throughput Performance.

matter of fact, SASE+, ZStream and Flink do not produce results for several hours for some values of selectivity and window sizes. This is because the throughput of SASE+, ZStream and Flink are highly dependent on the number of partial matches within a window. As the window size (or selectivity or number of matches) increases, these systems produce and process a large number of partial matches. Flink spends most of its time compressing and decompressing common events within the partial matches. That is, it travers through the stack of pointers using DFS (Deep First Search) to extract all matches. SASE+ utilises list structure to store all partial matches and need to check them one by one. ZStream performs worst in general. This is because ZStream uses a cost model which is highly dependent on data statistics and would need to be optimised every time there is new data.

In contrast, RCEP first employs an event tree and the range queries to extract a small set of events that can produce matches. Second, it employs hybrid-join algorithms and partition events based on the Z-regions and creates matches in batches. This results in a low cost per event within a window hence high throughput and scalability over larger window sizes.

In figure 3.12(b) we consider the worst case scenario for our batch join based algorithm, where the number of starting events $(a/\vec{E}_1)$ is low. In this case, RCEP's throughput performance is worse than that of Flink only for 1 point when the

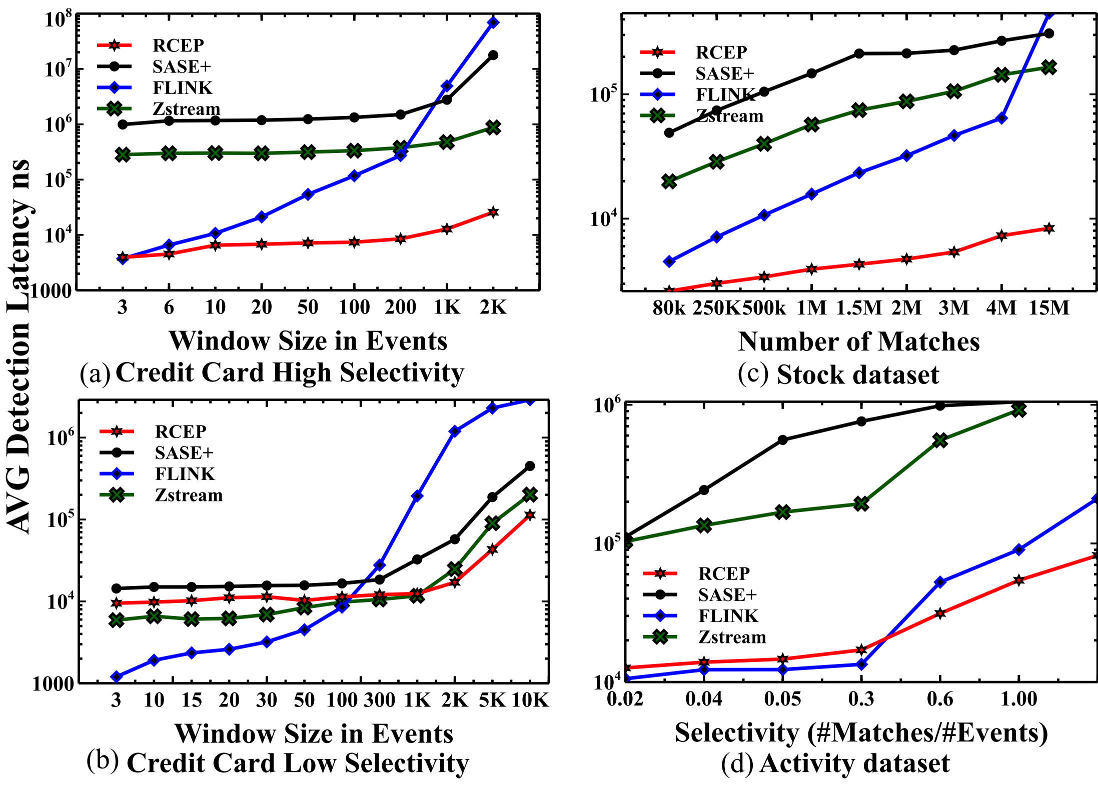


Figure 3.13: Average Detetion Latency

window size is too small. After that, Flink, ZStream and SASE+ get relatively worse when compared to RCEP as the window size increases. This can be explained as follows: increasing window size automatically increases the number of partial matches. All in all, these throughput experiments demonstrate that RCEP achieves significant performance improvements over existing CEP solutions.

Average Detection Latency

To further reinforce the aforementioned results, in Fig. 3.13, we show the performance in terms of average detection latency (this metric is explained in Section 3.8.1). Note that for Figs. 3.13 (b) we use a query with low selectivity (which is a variant of query 2 as explained before) and another with high selectivity (original query). We compare the detection latency between RCEP, Flink, ZStream and SASE+. From Figs. 3.13 (b), we can see that Flink and ZStream outperform, just initially, our system and SASE+ in terms of average detection latency. This is because Flink constructs matches as soon as events arrive and uses an optimised data structure to index partial matches. RCEP suffers due to low selectivity combined with low window sizes. This happens in the above particular case because of the following. Here, RCEP extracts and collects all the events for batch processing, but in the end there are only a few complete matches. Being a batch processing algorithm,

our fixed cost is slightly higher than SASE+, ZStream and Flink as we have to collect all the events for batch processing and getting only a few complete matches in the end does not compensate for the high fixed cost. In any case this corresponds to very special case of very low window sizes and low selectivity. However, after the window size is more than 100, then RCEP performs better for this special case of low selectivity query. Also note that the performance of Flink degrades very rapidly with the increasing number of partial matches. This is due to the fact that Flink has to compress and decompress more and more partial matches. For same reasons we can see that Flink initially performs slightly better than RCEP on activity dataset as shown in Figs. 3.13 (d). However, the performance of Flink decreases with the increasing number of partial matches. In other cases RCEP always performs better that other systems.

3.9 Conclusion

In this chapter, we proposed an efficient re-computation based, complex event processing system for expensive queries. We observed that traditional systems suffer from performance degradation as they store all partial matches in their memory. Thus, they waste memory and additionally computation resources to perform searches on these partial matches.

To solve this problem, we introduced an efficient multidimensional B-tree based indexing of events. Instead of storing all partial matches, which are the different combinations of matching events, we just store the events. When an event arrives that can complete the match, we collect and process the events in batches to find the complete matches. For events with multiple attributes, we employ a Z-address based indexing scheme that converts multi-dimensional attributes into a single value. This allows us to adapt the B-tree based indexing scheme for multi-dimensional events and use range queries to extract a small set of events that can produce matches. We employ a heuristic-based algorithm for theta-joins (HJoin) and partitioned events based on Z-regions to join events and create matches in batches. Hence, RCEP does not waste computational resources on sequential scans of large number of partial matches and decompression operations of matches.

Results show that RCEP performs much better and consumes less memory than existing systems such as SASE+, ZStream and Flink. This is consistent with our analysis, and partial matching techniques result in a much higher cost per event compared with recomputation-based techniques.

4
A Generic Framework for Predictive Complex Event Processing using Historical Sequence Space

Contents

4.1 Introduction **85**
4.2 Contribution **87**
4.3 Our Approach **88**
4.3.1 Querying Historical Space for Prediction 89
4.3.2 Summarisation of Historical Space Points 91
4.4 Implementation **93**
4.4.1 System Architecture 93
4.4.2 User Interface 93
4.5 Experimental Evaluation **94**
4.5.1 Experiment Setup 94
4.5.2 Datasets and CEP Queries 96
4.5.3 Accuracy Metrics 96
4.5.4 Precision of Prediction with Summarisation 96
4.5.5 Comparison with other Techniques: 96
4.6 Conclusion **97**

4.1 Introduction

Today, analytics are moving towards a model of proactive computing [EE11], and CEP is also experiencing a paradigm shift towards proactive and predictive

computations. That is, given a partial match, *predictive CEP* systems provide the possible future events of partially matched sequences which can then be converted into full matches. This can enable users to mitigate or eliminate undesired future events or states and identify future opportunities. Associating prediction and automated decision making enables progressing towards proactive event-driven computing. Proactive applications have attracted the attention of researchers from different fields. The problems of predictive CEP bear remarkable similarities to those of *sequence pattern mining and prediction*. Sequence prediction consists of predicting the next element for a given input sequence by only observing its previous items. The number of applications related to this problem is quite large. It covers applications such as consumer product recommendations, i.e. given a sequence of a consumer past purchases, it can predict its next purchases, weather forecasting, i.e. given a sequence of observations about the weather over time, it can predict the expected weather tomorrow; web page prefetching, i.e. given the history of access data, it can predict the data that will be requested next time, and stock market prediction, i.e. given a sequence of movements of a stock over time, it can predict the next movement of said stock.

In this context, a large body of sequence prediction models have been proposed, such as: Prediction By Partial Matching (PPM) [CW84], All-K-Order-Markov [PP99], Dependancy Graph (DG) and Probabilistic Suffix Tree [BYY04]. Some important limitations are present in each of these models. First and foremost, these models are based on the Markovion hypotheses that an event is only dependent on its predecessor, however this is not the case in our application and many others. Second, these models suffer from the catastrophic forgetting of older sequences, where only the k recent items of training sequences are used to perform predictions: increasing k often induces a very high state complexity and consequently such techniques become impractical for many real-life applications [Gue+15]. These limitations can have a major impact on the outcome of the prediction, causing poor accuracy.

There are several challenges to build a swift and efficient approach. First, it is necessary to have an efficient data structure in terms of memory to index all sequences. The second challenge is in designing algorithms to handle complex data, i.e. multidimensional sequences. Third, the data structure should be updated efficiently when adding a new sequence. Finally, an algorithm must be precisely defined to perform fast and more accurate sequence prediction.

4.2 Contribution

Our approach leverages two key observations: (i) historical matches can give an "expert" view on future matches; (ii) in case that memory optimisation is needed, summarising older matches according to their observed importance is a way to avoid catastrophic forgetting, while operating in the main memory for a real-time response.

To administer the first observation, we propose a novel N dimensional (N-D) historically matched sequence space $\mathcal{H}_{space}$. Based on this, we design an index structure that leverages the embedding of a fractal-based space-filling curve, the Z-order curve [Mor66], to map the coordinates of the N-D space into a 1-D space, while preserving the locality of points. In order to query 1-D points, we propose a novel range query algorithm that caters for the unbalanced nature of the partial matches, and locates nearby points for predictive analysis.

To administer the second observation, since the index size expands proportionally to the number of matched sequences, we need to compress the older matches while at the same time preserving some of their information. By doing so, the result will be a minor loss of knowledge. To summarise older points in $\mathcal{H}_{space}$, we first gather the points covered by the *top-k* most infrequent range queries in a streaming manner. Second, we use the *weighted average mean* to summarise the points that are closer to each other. The weights of the points are determined by the frequency by which they are queried, in $\mathcal{H}_{space}$. This not only provides an efficient way of summarising older data but also offers an efficient way of maintaining some of the information of the older matched sequences according to their importance.

Our experimental evaluation over two real-world datasets shows the significance of our indexing, querying and summarising techniques. Our system outperforms the competitor by a considerable margin in terms of accuracy and performance.

Integrating the contributions specified above, we demonstrate a system for predictive CEP, called Pi-CEP (Predictive Complex Event Processing). It provides the aforementioned complementary functionalities and can be integrated with existing general purpose CEP systems. Additionally, we provide a user-friendly interface for users to interact with and visualize results. Moreover, we provide customized search components to meet the demands of advanced users.

This chapter studies the problem of sequence prediction in the context of CEP systems. However, our techniques are also relevant in the context of general multidimensional sequence prediction. We tackle this problem from a new point of view, with the main aim of pushing the real-time predictive CEP capabilities to the database layer so as to take advantage of and extend existing data structures, query execution and optimisation techniques.

This chapter is organised as follows. Section 4.3 presents our approach, followed by section 4.4 which describes our implementation. Finally, Section 4.5 shows some results. Note that the related work is discussed in Chapter 2.

4.3 Our Approach

Given a pattern query Q, a stream $S = \{e_1, e_2, \dots\}$ – where each event forms a tuple $e = (A, \tau)$ of attribute values $A = \{a_1, a_2, \dots, a_l\}$ associated with a timestamp τ – the objective of the CEP system is to detect chronologically ordered sequences of events, each of the form $\overrightarrow{e}_f = \langle e_1, \dots, e_m \rangle \; m > 0$, that occur in the event stream and are correlated based on the values of attributes and defined temporal operators in the pattern query Q. The pattern query Q over S is evaluated in a progressive way. That is, partial matches $\overrightarrow{e}_p = \langle e_1, \dots, e_i \rangle$, where $i < m$, are formed before a full match is detected. Then for a partially matched sequence $\overrightarrow{e}_p$, our task is to predict future events that can turn $\overrightarrow{e}_p$ into $\overrightarrow{e}_f$ using the universe of fully matched sequences. That is, if m is the length of the fully matched sequence, for a partially matched sequence $\overrightarrow{e}_p = \langle e_1, \dots, e_i \rangle$, we would like to predict the future events from e_{i+1} to e_m. To attain this, we model our solution based on an N-D historical matched event database called *historical space* $\mathcal{H}_{space}$. Using $\mathcal{H}_{space}$, we can employ range queries for the partially matched sequences to determine the predicted events.

Let $\mathcal{H}_s = \{A_1 \times A_2 \times, \dots, A_n\}$ be an N-D *lattice* for the universe of fully matched sequences, where an n-tuple $X = (x_1, x_2, \dots, x_n)$ defines a point in $\mathcal{H}_{space}$ and $x_i \in A_i \; \forall i \in n$. Then with the arrival of a partially matched sequence $\overrightarrow{e}_p$, we answer *N-D range queries* on $\mathcal{H}_{space}$, which in turn is detailed later in this section. The idea behind this approach is that the points lying within the range queries or its neighbours are the predicted events for the partially matched sequence $\overrightarrow{e}_p$.

A large number of works (such as Quad Trees, KD-Trees, etc.) [Hil91; Mor66; Sam05] have been proposed for encoding N-D space. Considering the large number of dimensions in our context, and the effectiveness of space-filling curves in spatial domains [Sam05], we use the Z-order curve [Mor66] for N-D space encoding.

The indexing approach based on the Z-order curve is detailed in Section 3.7.2 of Chapter 3. Let's recall some of the properties of the Z-order curve: It preserves the proximity properties among the points while leveraging the effectiveness of linear data structures (such as B+tree) for range queries. The construction of the Z-order curve is accomplished by the simple process of *bit-shuffling*. The Z-order curve is used with B+tree for our indexing scheme [Ram+00b]. The nodes of the B+tree are sequentialised using a Z-order curve to retrieve a given region efficiently In order to provide the compact representation of encoded dimensions, we employ compressed bitmaps [Cha+14] to store large Z-values and to compress the sequence of homogeneous 1's and 0's within Z-values. This not only provides a good compression ratio but also simplifies the comparison of Z-values by using fast bitwise operations over the bitmaps which are supported by hardware. The Z-value, the timeline of the Z-value and the number of time it appears in the history form the fondation of our indexing scheme. We call it the *temporal Z-value*.

Definition 12: Temporal Z-value

$\mathcal{Z}_\tau = (\mathbf{Z}, T, f)$ contains a Z-value $\mathbf{Z}$, a set of timestamps T to denote its timeline, and the frequency f of times $\mathbf{Z}$ appears in the historical space $\mathcal{H}_{space}$.

The essence of the temporal Z-value is that, due to the iterative nature of the underlying fractal, it imposes the required sequential order of matched sequences on $\mathcal{H}_{space}$. One other attractive feature of our indexing technique is that $\mathcal{H}_{space}$ can be efficiently stored and retrieved from the disk due to the linear properties of B+trees.

4.3.1 Querying Historical Space for Prediction

4.3.1.1 Pre-processing Stage

We first consider the case in which all the matched sequences in the $\mathcal{H}_{space}$ are of equivalent length (i.e. of same dimensions). In the pre-processing stage, we first construct the range queries and their corresponding Z-values. For this task, we maintain an inverted index to record the *minimum* and *maximum* values of all dimensions in the $\mathcal{H}_{space}$. That is, if there are d_p known dimensions in $\overrightarrow{e}_p$ and d_m is the expected total dimensions of a $\overrightarrow{e}_f$, the minimum and maximum values of the range query are defined as follows:

$$\begin{aligned} X_{min} &= \Big(x_1, \ldots, x_{d_p}, min(x_{d_{p+1}}), \ldots, min(x_{d_m})\Big) \\ X_{max} &= \Big(x_1, \ldots, x_{d_p}, max(x_{d_{p+1}}), \ldots, max(x_{d_m})\Big), \end{aligned}$$

where $\forall x_i \in A_i, min(x_i) \leq x_i \wedge max(x_i) \geq x_i$ and note that $x_1, \ldots, x_{d_p}$ are already known as they correspond to the already known points in the partially matched query. Also note that in case of summarisation, the procedure of tracking minimum and maximum values is not affected because the values are not discarded, but rather summarised. The range query points are then mapped onto the Z-values, i.e. $\mathbf{Z}_{min} = Zval(X_{min})$ and $\mathbf{Z}_{max} = Zval(X_{max})$. These range query points are used to traverse the B+tree. Furthermore, since both have the same values for known dimensions, any of them can be used to determine if a point in the $\mathcal{H}_{space}$ can be enclosed by the range query points.

In the case of determining range queries when the $\mathcal{H}_{space}$ contains variable lengths (dimensions) matched sequences may be addressed in a similar way. For example, if $\overrightarrow{e}_{f_1} = \langle e_1, e_2, e_3 \rangle$ and $\overrightarrow{e}_{f_2} = \langle e_1, e_2, e_3, e_4 \rangle$ are two matched sequences in the $\mathcal{H}_{space}$, a single range query will not cover both of them. One solution used for this problem is to create different historical spaces and a set of range queries covering different matched sequences of specific dimensions.

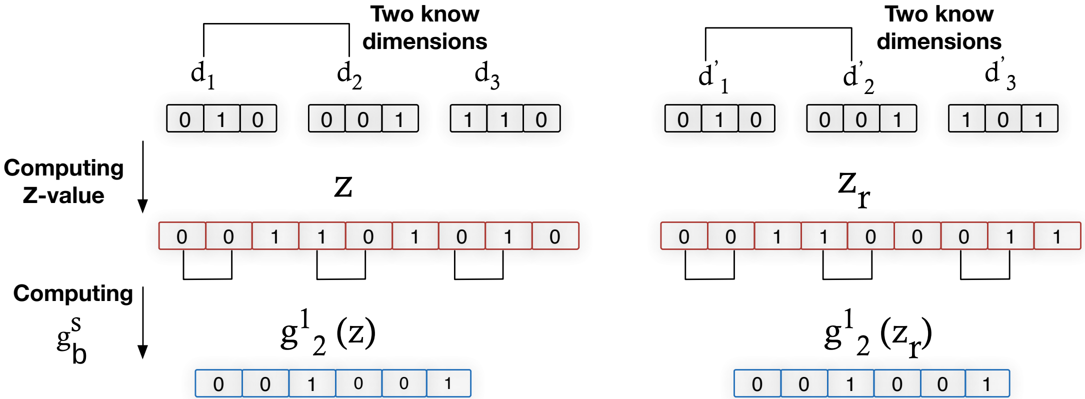


Figure 4.1: Z-value encoding and calculation of g_b^s over a point in $\mathcal{H}_{space}$ (left) and the range query point (right) with two known dimensions.

4.3.1.2 Post-processing Stage

In this stage, we search over the nodes of the B+tree that are intersected by a range query constructed in the pre-processing stage. The idea is to extract elements that lie in the range. This is done by checking the bits of known dimensions (i.e, where values are known), of node's Z-value $\mathbf{Z}$ and either the $\mathbf{Z}_{min}$ or $\mathbf{Z}_{max}$ value of the range query that we call $\mathbf{Z}_r$. Let g be a function mapping the Z-value to a bitstring (or binary number), we use $g_b^s(z)$ (small caps for Z-value's bitstring for brevity) to denote all the b most significant bits of $g(z)$, each skipped by s bits; e.g. $g_2^1(z) = 1010$ for $z = 101101$. We say that for two bitstrings z_1 and z_2 , z_1 is a b_1-prefix-s_1 of z_2 , iff z_1 is identical to all the highest (most significant) b_1 bits in z_2 each skipped by s_1 bits ; e.g. **1010** is a 2-prefix-1 of **101101**. Hence for a node's Z-value bitstring $g(z)$ and given query range bitstring $g(z_r)$, we compute $g_b^s(z)$, $g_b^s(z_r)$, and if z is a t-prefix-s of z_r, it shows that z is enclosed by the given range query. The values of b and s are the number of known and unknown dimensions in the given $\overrightarrow{e}_p$ respectively. Thus, using fast bitwise operations, we can easily find the contender set of points in the $\mathcal{H}_{space}$ for prediction. Figure 4.1 shows the computation of $g_b^s(z)$ and the range query point $g_b^s(z_r)$.

The same g_b^s function is also used to match variable length sequence points in the $\mathcal{H}_{space}$ and range query points. However, in this case the values of t and s are adjusted both for point z and range query point z_r. For instance, if there are two unknown dimensions and one known dimension in z, yet there are three unknown dimensions and one known dimension in z_r, the functions are as follows: $g_1^2(z)$ and $g_1^3(z_r)$. The skipping factor caters for the variability of length between points in $\mathcal{H}_{space}$ as well as the range query point.

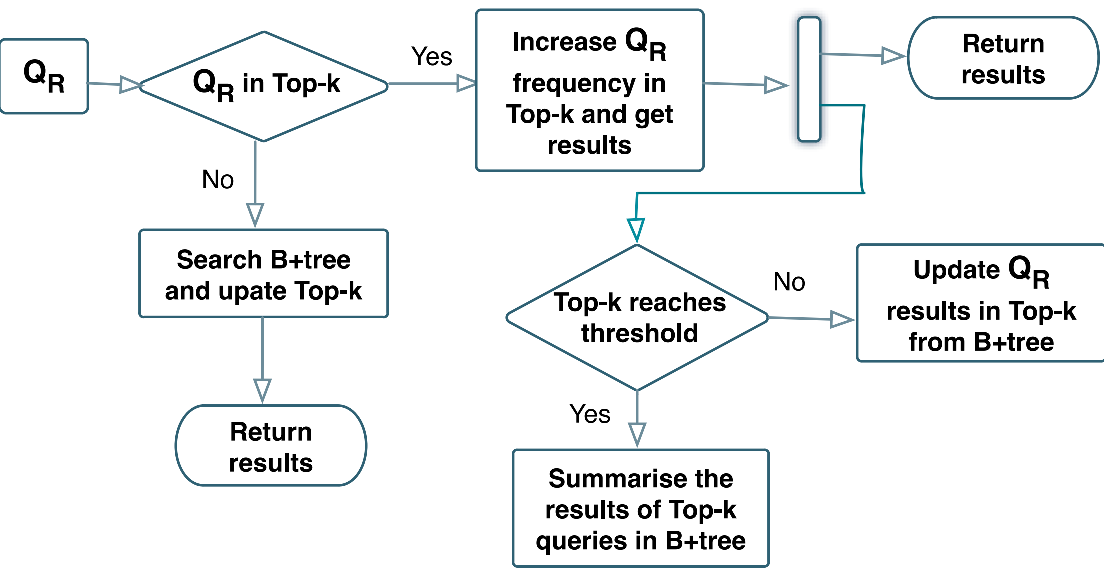


Figure 4.2: Range Query Processing with Summarisation

4.3.1.3 Reaching out to the Neighbours

The range query comparison and points in the $\mathcal{H}_{space}$ are initially strict for each prediction task. For a given partial match sequence $\overrightarrow{e}_p$, if the post-processing stage cannot find similar sequences to generate a prediction, we assume that the partial match contains some noise and the post-processing stage is dynamically relaxed. This is done as follows: first we get the neighbouring points of the range query Z_r that lies under the same B+tree node; second we check the similarity of the most significant bits (MSB) of the neighbouring point's bitstrings $g_b^s(z)$ and $g_b^s(z_r)$– called MSB similarity. Let $h(g_b^s(z), g_b^s(z_r))$ be a function that determines the degree of similarity between the MSB of two points; e.g. $h(g_b^s(z_1), g_b^s(z_2)) = 4$ for $g_b^s(z_1)$ =**0110**00 and $g_b^s(z_2)$ = **0110**11. Given two points z_i and z_j, if $h(g_b^s(z_i), g_b^s(z_r)) < h(g_b^s(z_j), g_b^s(z_r))$, it shows that z_j is closer to z_r than it is to z_i. Note that the above similarity measures are designed by considering the nature of Z-order curves, and techniques such as Hamming distance would not be appropriate in our context. Furthermore, similarity measures are used to add ± error margins. The set of neighbours with the highest MSB similarity according to the defined error margins are then chosen for the predictive response.

4.3.2 Summarisation of Historical Space Points

Till now, we kept all the history. However, in the case if the memory optimisation is desired we propose a summarisation technique. With summarisation, we aim to maintain the advantages of both worlds: keeping all the history vs. forgetting

older points. Our idea is to compress the most infrequent queried matches while at the same time preserving some of their information. The process of summarisation is divided into two tasks: (1) continuously evaluating and updating top-k most infrequent range queries generated by the system; (2) approximating summarising of older points within the top-k range queries that are closer to each other in the $\mathcal{H}_{space}$, according to their weights (i.e. f from Definition 12). This procedure is incremental and applied continuously over the system's life-time. The first task is well-studied and we employ a sketch and heap structure to provide approximate frequencies and the ordering of range queries. Our sketch stems from a count-min sketch [Cor11], but we modify it to incorporate range query information. Once a defined window expires for a pattern query, we extract the top-k most infrequent range queries in the sketch.

For the second task, as mentioned previously, the Z-order curve preserves the proximity of points in $\mathcal{H}_{space}$. Hence, points under the same parent node in the B+tree are inherently closer to each other. We use an approximate temporal Z-value $\hat{\mathcal{Z}}_\tau$ to summarise points under the same B+tree node which fall in the top-k most infrequent range queries. Our approximation technique is based on the *weighted linear combinations* (WLC) of Z-values. While other options exist, WLC has a strong history of successfully modelling the irregular and continuous characteristics of data according to observed weights. We compute $\hat{\mathcal{Z}}^n_\tau$ as a WLC of points closer to each other, i.e. under the same B+tree node:

$$\hat{\mathcal{Z}}^n_\tau = \sum_{i\,=\,0}^{n} w_i \cdot \mathcal{Z}^i_\tau, \;\; and \;\; w_i = \frac{f(\mathcal{Z}^i_\tau)}{\sum_{i=0}^{n} f(\mathcal{Z}^i_\tau)}$$

where $0 < w_i \leq 1$ denotes the weight given to a $\mathcal{Z}^i_\tau$. It is calculated from the set of points to be summarised together. We assign higher weights to the points having high frequency f in the $\mathcal{H}_{space}$, and $f(\mathcal{Z}^i_\tau)$ is the frequency of the ith point to be summarised.

The complete evaluation of range queries with summarisation is described in Figure 4.2. Given an incoming range query Q_R, we first check whether Q_R exists in the top-k or not. If not, we employ the usual range query search in the B+tree and update the top-k for Q_R. Otherwise, we update the frequency of Q_R in the top-k and return the result of Q_R. Furthermore, we check if the frequency of the top-k range queries has reached the defined threshold. If so, we generate the summary of range queries points and update the B+tree index. Otherwise, we update the results of Q_R, if any, from the B+tree. Finally, the result generator (in Figure 4.3) processes the results from the range query processor and sends it to the user interface in an appropriate format.

4.4 Implementation

In this section, we would like to explain how Pi-CEP handles prediction using pattern queries and customised parameters.

4.4.1 System Architecture

As mentioned already, the core functionality of Pi-CEP is implemented in Java, while the user interface is implemented in HTML and JavaScript: the back-end interacts with the user interface though Web Sockets. The high-level system architecture of the Pi-CEP is shown in Figure 4.3. The event streams and pattern queries are fed to the CEP engine, while customised parameters for the range queries, i.e. value of k are fed to the range query processor. For our current implementation, we are using the SASE CEP engine [WDR06]. This CEP engine produces full and partial matches that are delegated to the Z-value and range query generator respectively. The Z-values of the full matches are stored in B+tree index, while generated range queries are delegated to the range query processor.

4.4.2 User Interface

The user interface is shown in Figure 4.4. It is used to provide the interactive results of fully matched patterns and predictive events for partially matched patterns. It consists of three main parts: the control panel (left); the graph display panel (bottom right); and a fully/predictive matched pattern panel (top right). The control panel is used to let users specify the input pattern queries (ready-made or custom), and to select dataset and customised parameters for the range queries, such as the k value etc. The graph display panel shows two real-time graphs: the number of matched and predictive patterns; and the change in the size of the underlying B+tree index with the arrival of events following the summarisation process. The matched pattern panel shows the real-time matched set of events for a defined pattern and the predictive events provided by our system. We use different colour schemes for both of them to visualise it in an aesthetic fashion. Moreover, the system can be paused and resumed to compare each produced result.

We designed three main features of Pi-CEP. (1) Our system can provide predictive events. (2) Pi-CEP can efficiently evaluate and update top-k most infrequent range queries generated by the system and then summarise the older points within top-k range queries. The summarisation measures can be customised using different values of k. (3) Pi-CEP equips a user-friendly interface to fulfil user-computer interaction requirements in real-time.

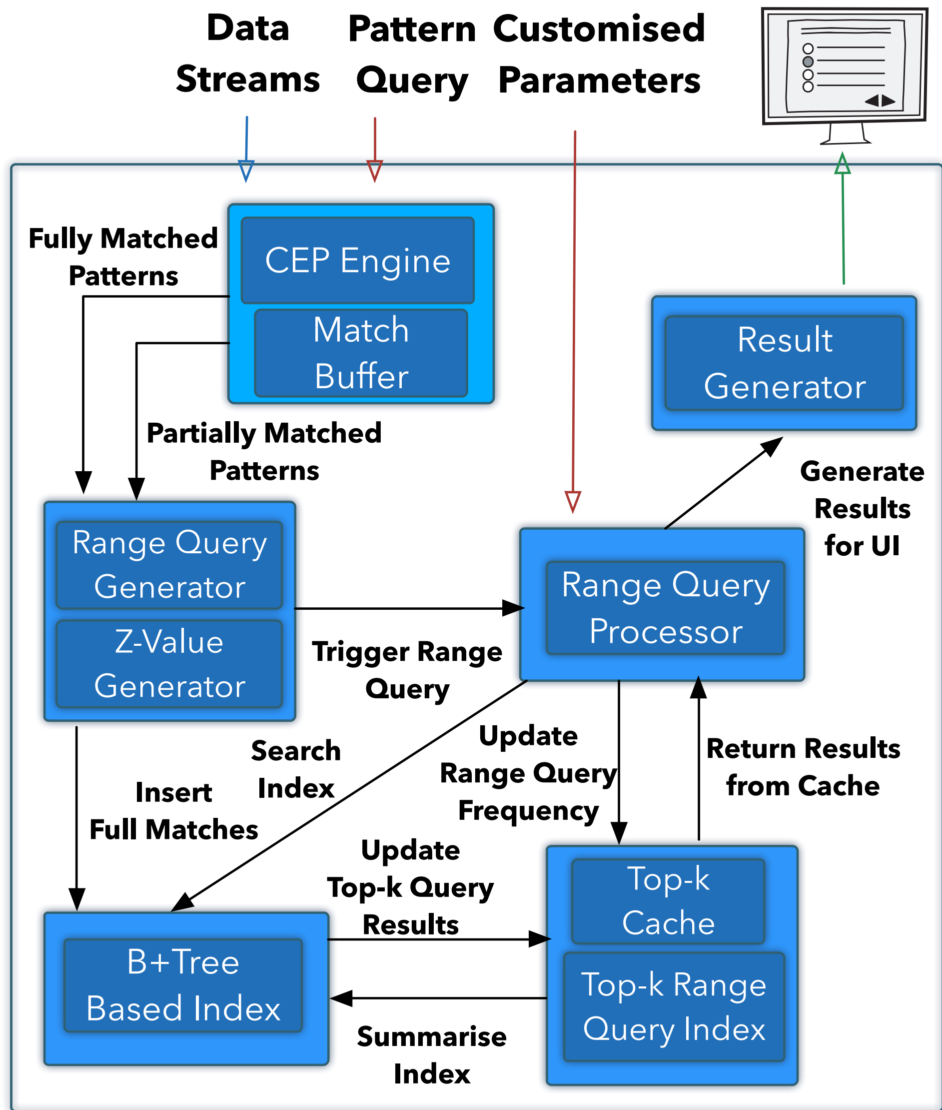


Figure 4.3: System Architecture of Pi-CEP

4.5 Experimental Evaluation

In this section, we evaluate the performance and precision of our proposed techniques.

4.5.1 Experiment Setup

Our algorithms are implemented using Java and evaluated on an Intel Xeon E3 1246v3 processor with 32GB of main memory and a 256Gb PCI Express SSD. It runs a 64-bit Linux 3.13.0 kernel with Oracle's JDK 8u112. For robustness, we performed 10 independent runs and report the median values.

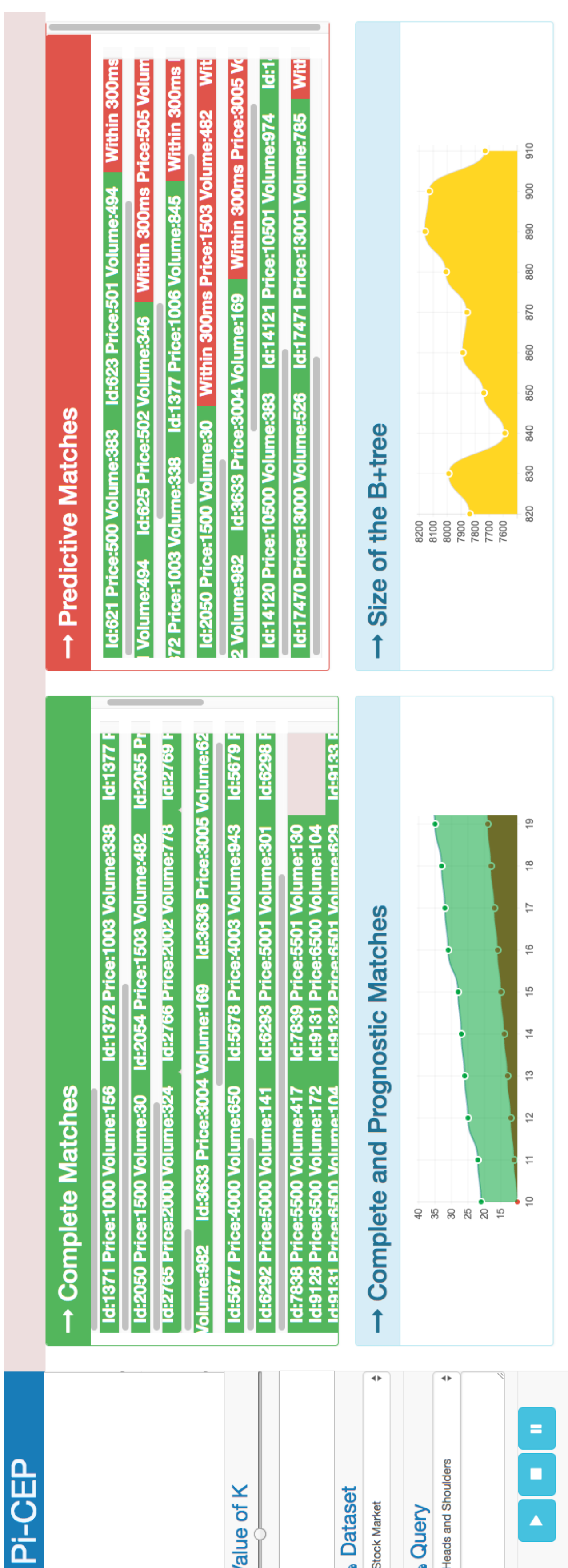


Figure 4.4: Interface of Pi-CEP

4.5.2 Datasets and CEP Queries

We use two real-world datasets for evaluation. All the datasets are first processed using the SASE [WDR06] CEP system to generate partially matched sequences (PM) and full matched sequences (FM) streams. These streams are then chronologically fed to our system. The two real-world datasets include: the *Activity Dataset* and the *Credit Card Transactions Dataset* (same datasets as chapter 3).

4.5.3 Accuracy Metrics

We provide the accuracy metrics as follows:

$$\frac{\text{\# of correctly predicted PM}}{\text{\# of PM}}$$

We keep the predicted PM until an FM arrives. Once an FM is detected, we compare if the predicted PM satisfies the FM. Furthermore, $k = 10$ is set for all the experiments and an error of $\pm 5\%$ is considered for correctly predicted PMs.

4.5.4 Precision of Prediction with Summarisation

The first question we investigate is "*How useful is the summarisation process w.r.t deleting older full matches?*" This measures the effect of forgetting older matched sequences on prediction. Figure 4.5(a) and (b) shows the prediction accuracy of both datasets by varying the matched sequences. To showcase the effectiveness of summarisation, we forget older matches after a window expires. From Figure 4.5, our summarisation technique results in better accuracy as time passes, since older values aid in predicting future matches. However, forgetting the history on the expiration of a window results in reduced precision and only recently matched sequences are used for prediction. Furthermore, since similar matched sequences repeat for the activity dataset (Figure 4.5(b)), its evaluation provides better accuracy measures.

4.5.5 Comparison with other Techniques:

Next, we investigate "*How our techniques compare to existing sequence prediction techniques (i.e. CPT+ [Gue+15]) both in terms of performance and accuracy?*" For this task, we extend CPT+ to work on streams, i.e. the fully matched sequences are inserted in a streaming fashion and partially matched sequences are used to predict the sequences in a streaming manner. Note that CPT+ only supports sequences with one-dimension, therefore, we use the Credit Card Transactions Dataset with 1-dimensional sequences. Figure 4.6(a) shows the accuracy comparison of both systems. While Figure 4.6(b) shows the performance of inserting matched sequences and querying predictive sequences while varying the number of fully matched

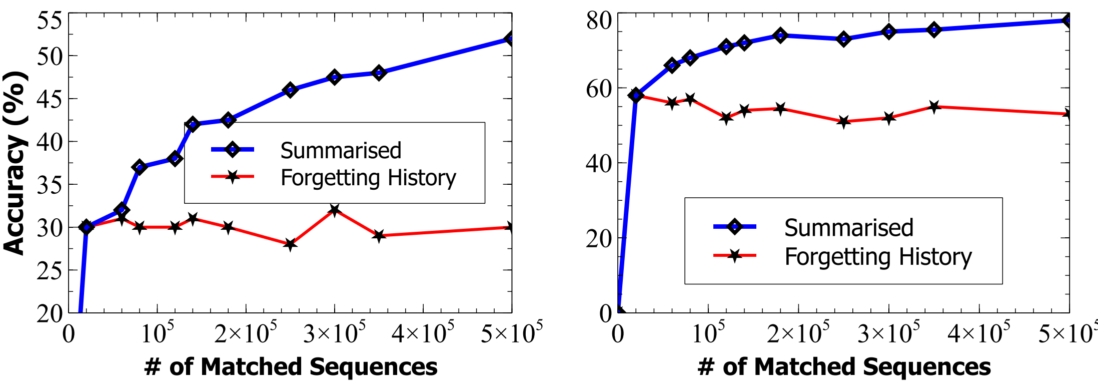


Figure 4.5: (a) Credit Card Dataset (b) Activity Dataset: Accuracy comparison of prediction for the number of matched sequence;

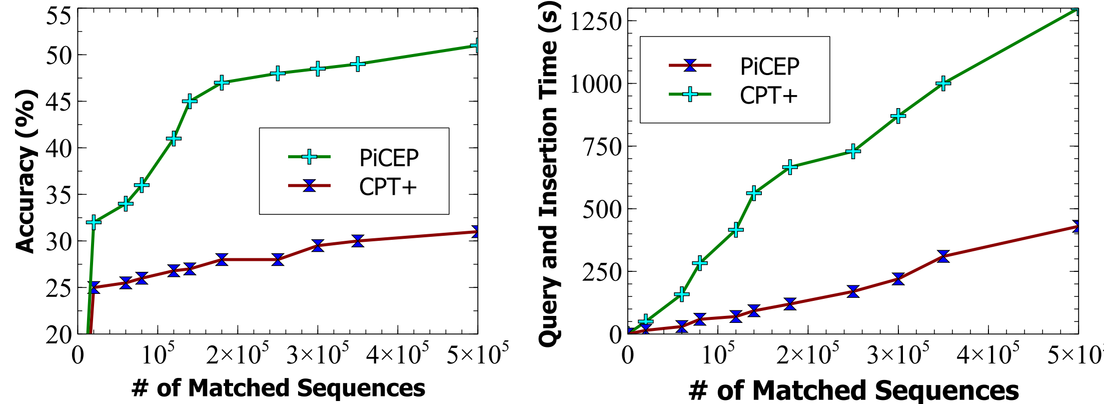


Figure 4.6: (a) Accuracy comparison of prediction for the number of matched 1-dimensional sequences (Credit Card Dataset); (b) Execution time in seconds for the insertion and prediction

sequences. From Figure 4.6, our system outperforms CPT+ both in terms of accuracy and performance. The reasons are summarised as follows: (i) CPT+ identifies frequent sequences in the training phase for efficient compression and prediction. This however is not efficient in streaming settings (ii) Due to the non-linear nature of CPT+ trie structure, it requires m comparisons for a sequence of length m, and for n distinct sequences, the cost is $O(m \times n)$: the B+tree requires $O(log\ n)$ for the lookup and insertion with fast bitwise operations.

4.6 Conclusion

We proposed a solution for a predictive CEP system. Our aim was to predict future matches from partially matched sequences using a historical sequence space. We used a memory efficient and real time updatable data structure to index all historic sequences. Then, we designed fast and accurate prediction algorithms. Additionally, a summarisation approach was proposed to compress non-important sequences.

Our experimental analysis demonstrated the potential of our approach and suggested that predictive CEP may well be best viewed as a distinct predictive task. Our preliminary research indicates that the study of sequence mining and prediction in CEP systems is promising and represents a new application of sequence prediction in the field of the internet of things and sensor networks. Moreover, it paves the way for further research possibilities on both predictive CEP systems and sequential pattern mining algorithms. In future, we would like to improve the prediction accuracy even more. We would like to also study if integrating machine learning approaches with our system can improve the performance.

Part III

Real World Event Processing Challenges

Introduction to DEBS Grand Challenges

In order to demonstrate the ability of our approaches, we participated in different challenges. The overall goal of these Challenges is to provide a common ground for researchers to evaluate and compare event-driven systems.

The ACM Conference on Distributed Event-Based Systems[1] is a conference in the area of event processing and offers a forum for researchers to exchange recent developments around event-based systems.

The 2016 DEBS Grand Challenge [Gul+16a] focused on reasoning over social network data to drive meaningful insights in real-time by defining two challenging queries. These queries, each for a different reason, cannot be handled by traditional techniques and therefore call for the development of specific architecture and data structures. The main novel features of this challenge were defined as follows: In the first query, the novelty was the non-linearity of the expiration of the elements. Since a traditional sliding window is not suitable, we investigated data structures offering the best tradeoffs for all the required operations. In the second query, unlike traditional approaches where no persistent data is stored over the stream, we had to manage a friendship graph which was persistent throughout the system execution. Due to the centrality of this structure, a careful design was therefore required.

The 2018 DEBS Grand Challenge [Gul+18b] focused on spatio-temporal streaming data. The novelty of this challenge was the combination of event processing and predictive analytics. The goal of the challenge was to make the naval transportation industry more reliable by providing predictions on vessel destinations and arrival times.

Following this, in this part of this thesis, we present the questions asked in the two Debs Grand challenges and the proposed techniques, which enabled as to contribute to overcoming these challenges and to develop `Upsortable` – a portmanteau of update and sort –. `Upsortable` is an annotation-based approach that allows the use of existing sorted collections from the standard Java API for dynamic data management.

The remainder of this part is structured as follows. Chapter 5 presents our solution for the 2016 Debs Grand challenge. Chapter 6 presents our solution for the 2018 Debs Grand challenge which was ranked second in this challenge. Appendix A provides guidance on the annotation-based approach `Upsortable`.

[1] **http://debs.org**.

5

High Performance Top-K Processing of Non-Linear Windows over Data Streams

Contents

5.1 Introduction 103
5.2 Input Data Streams and Query definitions 105
5.2.1 Input Data Streams 105
5.2.2 Query definitions 105
5.3 Architecture 106
5.4 Query 1 Solution 107
5.4.1 Data structures 108
5.4.2 Algorithms 110
5.5 Query 2 Solution 112
5.5.1 Data Structures 112
5.5.2 Algorithms 115
5.6 Evaluation 117
5.6.1 Experimental Settings 118
5.6.2 Queues implementation 119
5.6.3 Analysis of Query 1 Performance 119
5.6.4 Analysis of Query 2 Performance 121
5.7 Conclusion 122

5.1 Introduction

Stream processing has matured into an influential technology over the past decades, having a wide range of application domains including sensor networks and social networks. As a result, we have seen a flurry of attention in the area of *data stream*

processing. However, issues such as the identification of the most frequent elements (*top-k*) over graph structured networks, while considering the non-linear dynamic window model has not gained much attention in previous studies.

The 2016 DEBS Grand Challenge [Gul+16a] surfaced such issues, where a social network use case is considered. We herein summarize the two main queries/requirements for this challenge.

1. The identification of the top-3 active posts according to their scores which are computed in streaming settings: Scores of posts are increased with the arrival of new post-related comments and are decreased on expiration of related-comments and own score. This conforms to a non-linear window model.

2. The identification of social contagion in dynamic streaming settings. That is, given a window and the value of k, determining the top-k comments shared/liked between friends in the neighbourhood. The size of the largest clique determines the influence of a certain comment posted by a person.

Both the aforementioned queries present two orthogonal domains of interest and require different techniques to satisfy their requirements; however the basic constraints of stream processing are shared by both.

One naïve way of addressing the requirements of both queries is to compute the entirety of changes on each update and to reinitialize the process of identifying the top-k element for each newly observed change. Clearly, this naive method is too expensive in a streaming environment. In addition, the discussed state of the art Top-K algorithms in section 2.6.2 are adequate only in the case of linear data expiration and append-only data streams. However, to efficiently find the top-k posts according to their scores and the top-k comments with the highest influence in the connected network, the system should utilize certain guarantees to prune the search space based on some upper bound scores, and postpone the evaluation of query operators.

In this chapter, we present the solutions to the questions asked in the DEBS GC 2016. We propose several efficient and effective methods of enabling a lazy evaluation of non-linear window elements in Query 1. For this, we devise an easily computed upper-bound based on the temporal evolution of the posts. For Query 2, we rely on Turan's theorem [Tur41] to obtain guarantees on the absence of k-cliques in a graph. This lower-bound saves on numerous useless computations, especially in streaming settings. To store the global friendship graph, we propose two alternatives based on *roaring bitmaps* and *Bloom-filter* to handle the frequent updates and access in the community graph (i.e., friendship graph). The community graph is implemented by

an array-based data structure to efficiently determine cliques (i.e. identify social contagion or community detection) in the streaming settings.

The remainder of this chapter is structured as follows. Section 5.2 provides information about the data stream and query definition as defined by the DEBS organizers. Section 5.3 outlines the architecture of our proposed approach. Section 5.4 and 5.5 discuss customised solutions for both Query 1 and 2. Section 5.6 presents our experimental studies and Section 5.7 offers concluding remarks.

5.2 Input Data Streams and Query definitions

5.2.1 Input Data Streams

The input data is organized in four separate streams, each provided as a text file. Namely, the following input data files were provided:

- friendships.stream: $< ts, user_{id1}, user_{id2} >$, where ts is the friendship establishment timestamp, $user_{id1}$ is the id of one user and $user_{id2}$ is the id of another.

- posts.stream: $< ts, post_{id}, user_{id}, post, user >$, where ts is the post timestamp, $post_{id}$ is the unique id of the post, $user_{id}$ is the unique id of the user, $post$ is a string containing the actual post content and $user$ is a string containing the user's actual name.

- comments.stream: $< ts, comment_{id}, user_{id}, comment, user, comment_{replied}, post_{commented2} >$, where ts is the comment timestamp, $comment_{id}$ is the unique id of the comment, $user_{id}$ is the unique id of the user, $comment$ is a string containing the actual comment, $user$ is a string containing the actual user name, $comment_{replied}$ is the id of the comment being replied to (-1 if the tuple is a reply to a post) and $post_{commented2}$ is the id of the post being commented on(-1 if the tuple is a reply to a comment).

- likes.stream: $< ts, user_{id}, comment_{id} >$, where ts is the like's timestamp, $user_{id}$ is the id of the user liking the comment, $comment_{id}$ is the id of the comment.

5.2.2 Query definitions

The following are the criteria for the two main queries as defined by the challenge organizers:

5.2.2.1 Goal of Query 1

The goal of query 1 is to compute the top-3 scoring active posts, producing an updated result every time they change. The total score of an active post (P) is computed as the sum of its own score plus the score of all its related comments. Active posts with the same total score should be ranked based on their timestamps (in descending order), and if their timestamps are also equal, they should be ranked based on the timestamps of their last received related comments (in descending order). A comment (C) is related to a post (P) if it is a direct reply to P or if the chain of C's preceding messages links back to P. Each new post has its own initial score of 10 which decreases by 1 each time a span of 24 hours elapses since the post's creation. Each new comment's score is also initially set to 10 and decreases by 1 in the same way (every 24 hours since the comment's creation). Both post and comment scores are non-negative numbers. That is, they cannot drop below zero. A post is considered no longer active (that is, no longer part of present and future analysis) as soon as its total score reaches zero, even if it receives additional comments in the future.

5.2.2.2 Goal of Query 2

Given an integer k and a duration d (in seconds), find the k comments with the largest range, where the range of a comment is defined as the size of the largest connected component in the graph defined by persons who (i) have liked that comment (see likes, comments), (ii) know each other (friendships), and (iii) where the comment was created no more than d seconds ago.

5.3 Architecture

Figure 5.1 illustrates the overall architecture of our system. In order to parallelize query processing, the first step is to divide the system into a set of threads (i.e., Query 1 and 2 processors, input handler and output handlers for both queries), and replace the method calls between task boundaries with queues accordingly. Thus, given a set of files/streams containing events from `Comment`, `Like`, `Friendship` and `Post` streams, the data parser efficiently parses the events for these streams. The events are reordered according to their timestamps and queued into the respective queues: Query 1 only takes events from the `Post` and `Comment` streams, while Query 2 takes events from `Likes`, `Friendship` and `Comment` streams. We employ a set of different types of queues (blocking, non-blocking) to determine bottlenecks and the efficiency of our systems. Such discussion and explanations of queues is provided in the later sections.

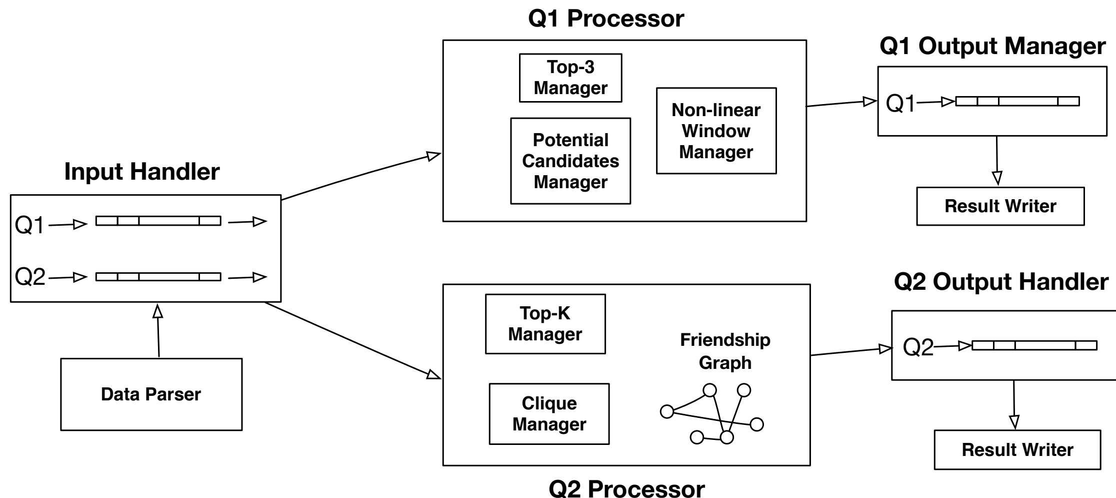


Figure 5.1: Abstract Architecture of the System

The specificity of Query 1 is the non-linearity of the expiration of its elements. To tackle this issue, the non-linear window manager is a trade-off between fast iteration – for removal – and fast update operations. The Query 2 processor employs a persistent friendship graph which is updated continuously for each newly arrived friendship event. This friendship graph is then utilised by the clique manager with the arrival of a `Comment` or `Like` event to determine the top-k most influential comments in the neighbourhood.

The query processors compute a list of top-k elements. When a change is detected it is sent to an output handler in order to avoid I/O pressure on the computing threads.

5.4 Query 1 Solution

In standard stream processing settings, elements expire in a linear fashion. Here in Query 1, the expiration of an element (i.e., a post) is subjective to its score reaching the value zero. Since the arrival of a new comment for a post increases its value, the main interest and complication of Query 1 reside in the fact that elements do not expire in a classical sliding window manner. Another interesting point of Query 1 is the cost of the posts' scores computation: it is an expensive process and should therefore be avoided through the use of less expensive bounds on scores. We first discuss the data structures used for managing expiration, upper-bound and top-3 in Section 5.4.1. Algorithms that employ these data structures are discussed in Section 5.4.2

	Ordered by expiration date DESC	Ordered by potential Score DESC	TOP 3
Remove expired comments	O(1)	O(log(n))	
New post	O(1)	O(log(n))	Modification occurs in update Top 3
New comment	O(log(n)) Remove Update Insert	O(log(n)) Remove Update Insert	Modification occurs in update Top 3
Update TOP 3	No changes	No changes	O(n)

Figure 5.2: Computational complexity for the different operations on the various data structures of Query 1. (DESC: Descendent)

5.4.1 Data structures

5.4.1.1 Posts eviction

To ensure that a post will expire, the simplest condition is that the last updates it receives is older than 10 days. By an update, we understand here the fact that a new comment has been linked to the post; therefore adding a value of 10 to the current score. Hence, we do not need to compute the exact score which is a computational expensive process, thus saving precious CPU resources.

To speed-up the process of posts eviction, we maintain the posts ordered by their ascending expiration date in a data structure called *expdate*. The process of updating the expiration date of a comment consists of first removing the post from the structure; and updating its value and adding back the value into the sorted structure: this process does not depends on the underlying sorted structure.

A simple sorted list is not a satisfying solution for sorting posts by their ascending expiration date, since adding back the value has a worst case computational complexity of $\mathcal{O}(n)$. The number of posts in this structure is large and such complexity is not efficient. We therefore rely on a `TreeSet` structure backed by a red-black tree, hence guaranteeing $\mathcal{O}(log(n))$ for insertion and removal. To evict comments at each tick, we poll the head of the tree – a constant time operation – until the current head's removal is valid.

5.4.1.2 Score management

The `potentialScore` data structure holds the posts that are the candidates to enter the top-3 structure. Since the computation of the exact score is an expensive process, we aim to use a less expensive upper-bound to sort the candidates to avoid the exact score computation as much as possible.

Following the procedure of the posts' eviction, posts are sorted by their upper bound albeit in descending order. When a post has a sufficient upper-bound to enter the top-3 list, its exact score is computed to affirm such hypothesis. We rely on a `TreeSet` structure to guarantee worst case $\mathcal{O}(log(n))$ for all required operations:

Post expiration The post may be at any place in the tree, therefore complexity is $\mathcal{O}(log(n))$.

New post The post has a score of 10, which is neither guaranteed to be an upper nor lower value in the tree, hence a $\mathcal{O}(log(n))$ complexity.

New comment If the upper-bound of the related post has increased, we first remove the post, update its upper-bound, and add it back to the structure. Insertion and removal are both in $\mathcal{O}(log(n))$.

Therefore the posts belong to two data structures: one for the management of eviction process and the second for score management. The algorithm of Query 1 ensures that both structures remains consistent.

5.4.1.3 Comments related to posts

To store comments related to a post, we decided to use a compact representation: we only store the ids of the comments into a list. Comments are added in chronological order, and their expiration is also chronological. It provides the guarantee that an earlier comment will expire before a later one, which is not the case for the posts. We could only retain the valid comments by checking and removing comments when required. This operation is costly and unnecessary, although keeping expired comments in memory does not require much space, and the space-time trade-off is definitely worth it. Comments related to a post are thus only removed when the post expires.

Comments are stored in an array-list of `long` next to their time of arrival. The time of arrival is used to determine whether a comment is still considered valid. Figure 5.3 depicts this structure. At time 11, the first comment has expired, the offset is set to 3. At time 21, the second comment also expired, thus the offset points to the first valid comment at index 5.

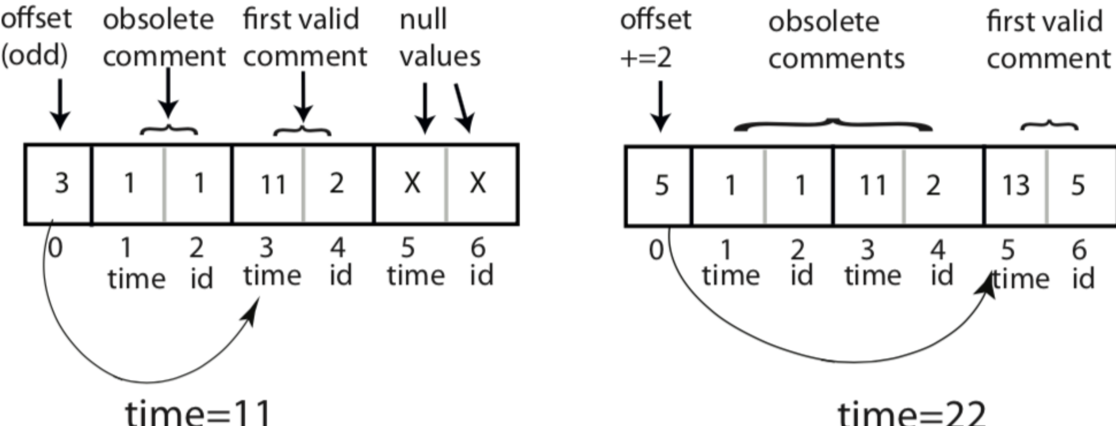


Figure 5.3: Storage of the comments related to a post. The offset gives the index of the first valid comment.

5.4.1.4 Top-3

Query 1 retains only the top three elements, whereas Query 2 has k as an input parameter. Thus, we use the low value of $k = 3$ to our advantage and use a special implementation that is efficient for a low value of k: a top-k structure using a primitive array of three pointers to the top three posts. We use an in-place insertion sort algorithm to sort the array after removal/insertion. Insertion has a very high computational complexity ($\mathcal{O}(n^2)$). However, for small arrays, it is the most efficient sorting algorithm. Its space complexity is constant, therefore there is no object creation/destruction that overloads the garbage collector (automatic memory management). The small value of k implies that the complete array fits in a cache line, thus guaranteeing high performance. Since there are many insertions/deletions, at each clock time, the performance gain is quite noticeable.

5.4.2 Algorithms

5.4.2.1 Workflow

Our solution for Query 1 consists of the following three steps: *first*, we update the internal timestamp; *second*, we process the post or the comment to update the internal structures; *third*, we remove expired values, and update the top-3 list if required. The first and last steps are independent of the nature of the input, while the second step depends on whether a post or a comment has arrived.

For the arrival of a new post, we add this post to both `potentialScore` and `expdate`. If a comment arrives, we first ensure that the post it relates to is valid. If the related post is valid, we add the commenter to the list described in Section 5.4.1.3. We increase the upper-bound of this post by 10, thus updating the state of `potentialScore`. We also update the expiration date of the post; resulting in a change in the state of `expdate`.

5.4.2.2 Top-3 entrance

As stated in the previous section, we delay the computation of the exact score using an upper-bound to filter out useless computations.

In order to determine the top-3 candidates, we iterate over the posts stored in `potentialScore` until the current post's potential score – an upper-bound on the exact score – is smaller than the third element in the top-3. If a post is a potential candidate, we compute its exact score, update the value of the potential score to this score and, if the score is larger than the smallest one in the top-3, the post is added to the top-3 array.

A trivial upper-bound can be computed by simply adding 10 to a post value every time a new related comment arrives. However, such a bound, while correct, is not efficient enough to avoid useless computation. We refine this upper limit by calculating when the post will see its score decrease. This value, called `nextDecay`, is initialized a day and a millisecond after the timestamp of the post. An associated value, called `maxDecay`, indicates by how much the upper-bound can be refined. These values are used to refine the upper-bound in the following way: at the current timestamp `ts`, we use `nextDecay` to compute the number of days `nDays` elapsed between `ts` and `nextDecay`. We can then safely decrease the value of the upper-bound (`potentialScore`) while determining the minimum value between `nDays` and `maxDecay`. `maxDecay` is used to avoid the following pitfall: the current timestamp being 15 days after `nextDecay` (`nDays` = 15). It is impossible that the decrease in the score, inputed to a single comment or to the post, be larger than the days this comment/post has before expiration: `nextDecay` is updated during the exact score computation. The next decay is computed for the post and each of its valid comments, the minimum value is then kept.

5.4.2.3 Lazy commenters evaluation

The most costly operation, during the computation of the exact score of a post, is to compute the set of commenters. A naive approach would use a multiset to store the commenters of the post and to maintain this structure throughout the life-time of a post. This method proves to be ineffective in practice due to unnecessary computations. As we presented before in Section 5.4.1.3, we maintain an array-list of `long` to store the id of the commenters along with their timestamp. Hence, we use a lazy evaluation strategy for the set of commenters. The set (a hashset), is initialized only when the score computation takes place and there is at least one valid comment. On this first call to this set building method, all valid commenters are added to the set. On subsequent calls, if some comments have been flagged as expired by changing the offset during preceding steps, the set is cleared. Otherwise, we add the newly added comments.

In addition to this, we have to clear the set if some comments have expired, since removing the commenter from the existing set could corrupt the data. Thus, if a commenter has commented twice on the same post and his/her first comment expires, he/she remains a commenter of the post.

5.5 Query 2 Solution

5.5.1 Data Structures

Given the Query 2 complexity, it is quite evident that several data structures are required to provide a tractable solution – each catering for different level of indexes.

5.5.1.1 Dense Numbering and main indices

To store comment and user identifiers, we opt for a dense numbering scheme, which has been proven to be very effective for real-time applications especially for the streaming databases [Sub+16; Che+15]. This technique is summarized here for completeness. Query 2 maintains two counters initialized with the value of 1 – one for users and one for comments – which are the two dense number generators for comments and users `long` identifier mapping to `int` values.

Using a dense numbering scheme not only prevents the explosion of memory footprint (it's upper bound is 8 Gigabyte for the setting of Grand Challenge), but also serves the purpose of an array-based data structure for storing comments liked by each user. Such details are discussed in subsection 5.5.1.5.

Table 5.1 presents the preliminaries of our two dense numbering schemes, and a list of the main data structures utilised for Query 2.

5.5.1.2 Friendship graph

The friendship graph is a crucial data structure for Query 2 as it grows monotonically (friendship links are added and never deleted), and its size evolves with time as new users enter the system. It is highly solicited for testing friendships between users and to determine who is affected by the arrival of a new event. Thus, it has to scale well with respect to the number of users while maintining an acceptable memory footprint.

We tested several solutions, each providing a different space/time trade-off. We recall first that the users are identified by a dense number, which could be used as an index in a data structure. A first naive solution is to store the similarity matrix of friendship between users as a two-dimensional array of boolean values. Each cell is initialized to zero, and one entry encodes a friendship relation for the

Table 5.1: Main data structure used in Query 2

Name	Type	Description
`comments2dense`	Map<Long, Integer>	`long` ids of comments are mapped to an `int` using dense numbering.
`dense2comments`	Map<Integer, Q2Comment>	comments dense ids are mapped to Query 2 comments instances.
`userLikedComments`	Map<Integer, OffsetIntArrayList>	Map a user dense id to the array of dense comments it has replied on. These array are tailored for out tasks and described at Section 5.5.1.5.
`commentLikes`	Map<Integer, List<Integer> >	Inverse index of `userLikedComments` : comment dense ids a remapped to the user who like this comment.
`dense2users`	LongArrayList	Array whose index is a dense user id and the corresponding value the actual `long` original id of the user.
`users2dense`	Map<Long, Integer>	Reverse index of `dense2users`

user at the *i-th* line and the user at the *j-th* column[1]. Such a solution would be lightning-fast given the nature of primitive arrays and their storage as contiguous elements in the memory, but it would be very sparse: when the number of users keeps growing, this does not scale in terms of memory footprint – even when only considering the triangular superior matrix of friendships since the friendship property is symmetric in this task.

In order to address sparsity and scalability, a common solution is to use bitmaps. For each user, we can associate a vector of n bits, where n is the current number of known users, or a sufficiently large number in order to scale with the task at hand. For a user u_1, when the *i-th* bit of its bitset is set, this encodes that u_1 is a friend of u_2. This is a fairly better solution when the number of users is small. With the increase in the number of users, however, it reaches the same memory footprint of the first solution in the situation of a full dataset as provided by the organizers.

We therefore came up with two competitive alternatives, and selected the one with the best performance.

5.5.1.3 Roaring bitmap friendship graph

Roaring bitmaps [Cha+15] are compressed bitmaps. Unlike their alternatives [LKA10], roaring bitmaps provide random accesses without the extra overhead of uncompressing the entire compressed bitset. That is, to check if the *i-th* value is set, roaring bitmap splits the entire bitmap in chunks, where each chunk is

[1] providing our dense numbering scheme starts at 1, there is a -1 translation to be performed when using their dense identifier as indexes in such array. This is also true for other solutions relying on the dense numbers as indexes.

an uncompressed bitmap. This is a valuable property that can be utilised in our case, since the friendship graph is accessed frequently. At the same time, the compression offered by the roaring bitmaps makes it possible to scale in terms of memory footprint.

5.5.1.4 Friendship graph with hashmap and bloom-filter

As an alternative to roaring bitmaps, we utilise a combination of hashmaps and a bloom filter for the friendship graph. The basic idea is as follows: the friendship graph is a hashmap with `int` to `Friend` mappings, where the object `Friend` contains a set of users that are friends of a user identified by a dense number (`int`) $i = dense(u_k)$. This enables the addition of new friendship relations and to test the friendship relation between two users. This may sound counter-intuitive to map primitive data-types to an object (given objects instantiation time and garbage collection), but keep in mind that we would only instantiate this object once per user in the entire timeframe of the system. The interesting point therefore is how the `Friend` object handles the collection of user's friendship. In our implementation, each `Friend` object holds a hashset to group the set of friends – each identified with a dense generated number – for a specific user u_k. This structure handles the insertion of users's new friends. To prevent frequent costly access to the hashset and to test whether a user is a member of the hashset, we employ a bloom filter. Since users are more likely not to know each other, bloom Filter presents an interesting data structure increase in speed.

We evaluated them using the test set provided by the organizers[2]. The value for parameter d (sliding window size) was set to 12 hours (as recommended by the organizers) and k to 10. It follows that the roaring bitmap approach, on the average of 10 runs, is significantly outperformed by the combination of hashmaps, hashset, and the bloom filter approach – this makes it the solution of choice for the friendship graph data structure in our submission.

5.5.1.5 Comments liked by each user

The friendship graph provides the basic building block for processing Query 2. However, to compute the maximal friendship clique (i.e. *community*) for a comment posted by a user, we require an additional data structure. The new data structure stores comments liked by each user and is subjected to high insertion rates (each time a `Like` event is received) with frequent read operations; the score of a comment is updated by determining the neighbourhood it has influenced. For this task, we employ an array-based data structure. The only issue is that the comments

[2] `http://bit.ly/1QRZkU3`

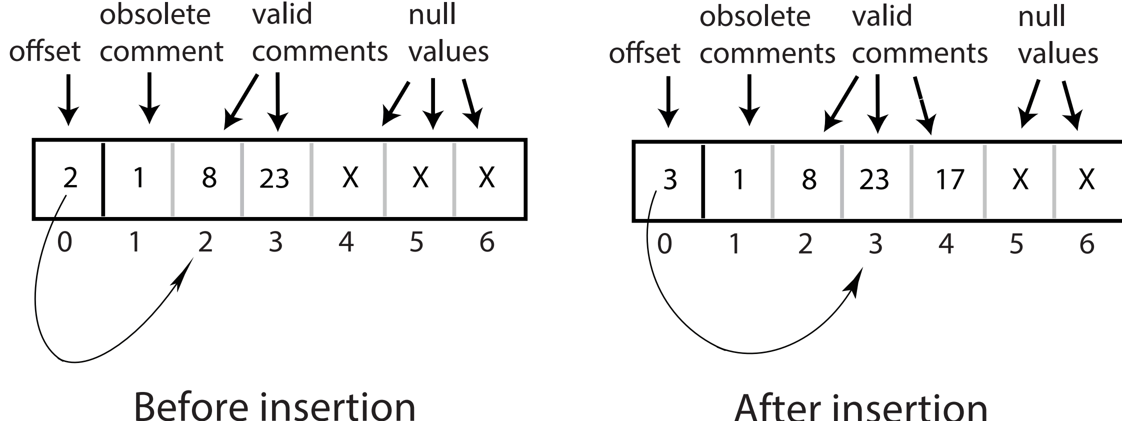


Figure 5.4: A sample array of sorted dense comment ids for a user. On the left is the state of the array before an insertion. On the right, is the new state of the array after an insertion occurs at a timestamp for which the older comment in the window has a dense id value of 9.

become obsolete when they leave the sliding window. We hereby devise a solution similar to that used in Query 1. That is, using an offset to indicate the position of the first valid comment in the array.

The offset utilised by the array to locate the first valid comments requires careful management. It is not desirable to update this offset each time the window slides, as in this case we need to update the offset for each newly arrived event for a user. Thus, we utilize a lazy update technique that is executed on two different occasions. The first update is on insertion in a specific array. Each time a new value is added to the table by a right-to-left traversal of the array until an obsolete comment is encountered. The second update is executed each time we actually read the array to compute the intersection of a given array with another array (actually computing the comments two users have in common). When computing this intersection we will inevitably encounter obsolete comments but only once since we will then update the offset of both arrays as we are traversing both of them to compute the intersection. This approach provided us a factor of improvement between 2 and 3 of the throughput in our solution.

5.5.2 Algorithms

5.5.2.1 Outline of Query 2 processing

Algorithm 5 illustrates the execution of Query 2 for each newly arrived event (`Comment`, `Friendship`, or `Like` events). It has the following steps.

1. A preprocessing step updates the set of comments to be taken into account (the method is named `updateWindow`). This methods returns the dense id

of the oldest comment in the window (any comment whose id is lower then this threshold will be discarded). This is useful in preventing an unnecessary full scan of users for each iteration. Simply, we remove the comments' Id a user has replied to, if the returned comment Id is lower than the threshold. However, this maintenance would be too expensive, and thus is only done periodically. Due to memory footprint reasons, we cannot afford to keep all comments' Id in the other data structures. Hence, all the obsolete comment ids are removed from the remaining data structures using a call to the `clean()` method. This is also a mandatory step as the obsolete comments may have been inserted as a top-k item. Thus, the method `clean` ensures that the top-k list no longer contains obsolete items for this new clock tick.

2. A second step implements the main procedure for processing events from `Comment`, `Friendship` or `Like` by employing the corresponding process. The call to `processComment()` is very straightforward; a new dense Id is associated to the new comment, the dense numbering indexes are updated accordingly and the comment is added to the current window. The remaining processes (`processFriendship()` and `processLike()`) encompass a complex behaviour and are presented in the aforementioned discussion.

3. The last step is performed regardless of the type of event (e.g. `Comment`, `Friendship` or `Like`). We first insert new items in the top-k if the cardinality of the list is less than k. This happens if we removed comments in top-k in the preprocessing step, and where the processing of the event did not allow for the top-k to complete. In this case, older comments – which are still in valid – should be promoted in the top-k. We first identify those comments and then insert them in the top-k. This process is triggered when calling `updateTopkIfNecessary()`. It is slightly tedious. It in fact relies on a sorted list of the minimal size of the largest clique for each valid comment, See Section 5.5.2.4 for detail on the lower bound used to minimize computation. The second post-processing task is straightforward; if there is a change in the top-k values/order, the results are passed on to the result-serialization thread, as previously described in the global architecture (see Figure 5.1).

5.5.2.2 Process likes

Whenever a `like` is processed, we add the `like` to the list of integers in `commentLikes`. We then recompute the friendship subgraph for the curren comment. If the number of edges is larger than the lower-bound (see Section 5.5.2.4), the largest clique is computed using the implementation of McCreesh [MP13], based on Tomita's algorithm [TS03], which is very efficient on dense numbered nodes.

Algorithm 5: Outline of query 2 processing an event

Input: *DebsRecord* : *event*
1 ▷ preprocessing *threshold* ← *updateWindow*(*event*)
2 *clean*(*event.getTimeStamp*())
3 ▷ actual event processing w.r.t its type
4 **if** *the event type is comment* **then**
5   processComment(event)
6 **else if** *the event type is friendship* **then**
7   processFriendship(event)
8 **else if** *the event type is like* **then**
9   processLike(event)
10 ▷ postprocessing
11 *updateTopkIfNecessary*()
12 *writeResult*()

5.5.2.3 Process friendship

An incoming friendship modifies the state of the global friendship graph, and consequently the subgraphs of the comments that are liked by the two users who are now friends. We therefore compute the list of comments concerned and apply for each of them the same process as the comment concerned by `Process like`.

5.5.2.4 Lower bound on clique computation

Turan's Theorem [Tur41] gives an upper bound on the number of edges for a n nodes graph to contain a r-clique, i.e. a clique of r nodes.

theorem 2: Turan's Theorem

To find k-clique in G(V,E) with n vertices : k ⪯ n(n-1)/2 and k⪯E Theorem of Turan: Let G(V,E), n vertices without k-cliques, then |E|⪯ [(k-2)n2]/ [2(k-1)]

Whenever a change occurs and a comment is a candidate to enter the top-k, we first validate that its number of edges is sufficient to contain a clique of r nodes, where r is larger than the clique of the last comment from the top-k. This provides a very quick way to determine if it is worth computing the size of the maximal clique for this graph.

5.6 Evaluation

A global system evaluation was performed by the organizers and our system exhibits the following performance: 142K events/second and 1 and 0.7 millisecond average latency for the first and second queries, respectively. We therefore use this section to detail query specific evaluations, with a particular focus on the impact of the message-passing queues.

d	*k*	LinkedBlockingQueue			BufferedLinkedBlockingQueue			LockFreeBlockingQueue		
		T	L	time	T	L	time	T	L	time
1	1	352	1.06	158486	500	2.98	111714	505	0.95	110666
	3	362	0.99	154107	520	2.91	107426	469	1.00	119125
	10	359	0.89	155634	515	2.91	108476	514	0.97	108747
	30	365	0.96	152904	522	2.87	107018	470	0.86	118808
60	1	384	0.24	145545	514	1.95	108716	505	0.25	110573
	3	381	0.23	146472	504	2.04	110821	501	0.20	111453
	10	386	0.24	144686	511	2.17	109369	487	0.21	114608
	30	365	0.24	153101	513	1.99	108802	492	0.23	113483
720	1	394	0.40	141720	505	3.11	110500	495	0.47	112855
	3	407	0.31	137499	502	2.52	111170	513	0.30	108969
	10	373	0.41	149838	516	2.07	108231	496	0.30	112546
	30	400	0.31	139762	496	3.44	112657	488	0.27	114480
1440	1	405	0.80	137996	496	3.72	112670	490	0.70	114036
	3	396	0.58	141140	491	6.25	113733	471	0.59	118764
	10	410	0.68	136200	501	19.70	111407	478	0.74	116778
	30	400	1.01	139680	493	246.61	113184	471	1.17	118644

Table 5.2: Performance of our solution on the large dataset for Query 2 only provided by the organizers with respect to variation of window size (d) in minutes, number of elements in top-k, and different event passing queues (LinkedBlockingQueue, BufferedLinkedBlockingQueue and LockFreeQueue). Throughput values (T column) are expressed in kilo-events per seconds, latencies (L column) in 10^{-4} seconds, and execution time (time column) in miliseconds.

5.6.1 Experimental Settings

Metrics (Query 1). For Query 1, we varied the number of processed events while employing different kinds of queues to evaluate its effect on performance. As the window is of a non-linear type, the number of events determines the granularity with which the window slides. We utilise different kinds of queues: the standard blocking queue, the buffered blocking queue, and the lock-free queue.
Metrics (Query 2). For Query 2, we varied a number of different parameters including: the size of the window in seconds, the value of the k, and the type of the queue. These parameters have varying effects on performance and provide useful insights for Query 2 performance.

Dataset. We use the same dataset that which was provided by the Grand Challenge committee for the evaluation of both queries. The dataset contains four streams, namely `Comment`, `Post`, `Like` and `Friendship` stream. In total the dataset contains about 55 million events.

Configurations. All the experiments were performed on an Intel Xeon E3 1246v3 processor with 8MB of L3 cache, and we report averages of over 10 runs. The system is equipped with 32GB of main memory and a 256Go SSD hard disk drive. It runs a 64-bit Linux 3.13.0 kernel with Oracle's JDK 8u05.

5.6.2 Queues implementation

The architecture of our solution, described in Section 5.3 highlights the fact that the various threads communicate via message-passing. We therefore heavily rely on queues that are used to pass raw events from parser to query processors, and we processed top-k from query processors to writer threads. We used three different queues in this configuration, Java's `LinkedBlockingQueue` (LBQ), a buffered version of our own that holds n events in a single slot (BLBQ). We also use a lock-free implementation, with the single producer single consumer Lock Free Queue (LFQ) from JCJT [3], an implementation of [GMV08]. The lock free queue is expected to have a lower latency than other queues, at the cost of a higher CPU usage. Since there are no locks, threads keep running in loop, waiting for an event to arrive.

5.6.3 Analysis of Query 1 Performance

Figures 5.5, 5.6 and 5.7 showcase the effect of queues on the latency and throughput values. From these figures, we can see that the BLBQ entails the highest throughput. However, it does result in higher latency values. This phenomenon can be explained as follows. The BLBQ queue works on a batch-based mode where a batch of events are added at once. Conventionally, this is done using mutual exclusion; processes modify the queue with a batch of events only and the process is guaranteed exclusive access to the queue. This increases overall throughput, however, with dire effects on the latency. The LFQ, contrary to LBQ and BLBQ, provides concurrent access to the queue among multiple processes. Slow or stopped processes do not prevent other processes from accessing the queue. Thus, the processes are not locked on the queue and there are a fair gains in terms of latency, without compromising the system throughput. For these reasons, we opt to use LFQ for the submission of the Grand Challenge solution.

The second observation that can be inferred from the Figure 5.5, 5.6 and 5.7 is as follows. With the increase in the number of events, there is no sharp drop in throughput values. That is, the throughput value do not follow a linear behaviour pattern per se. This is because of the lazy evaluation of the posts' score. Thus, if a post has a significant upper bound (i.e., probability) to enter the top-3 list; only then is its exact score calculated. If its score is greater than the posts' score in the top-3 list, it enters the top-3 list. This strategy greatly reduces computation overheads.

[3] `http://jctools.github.io/JCTools`

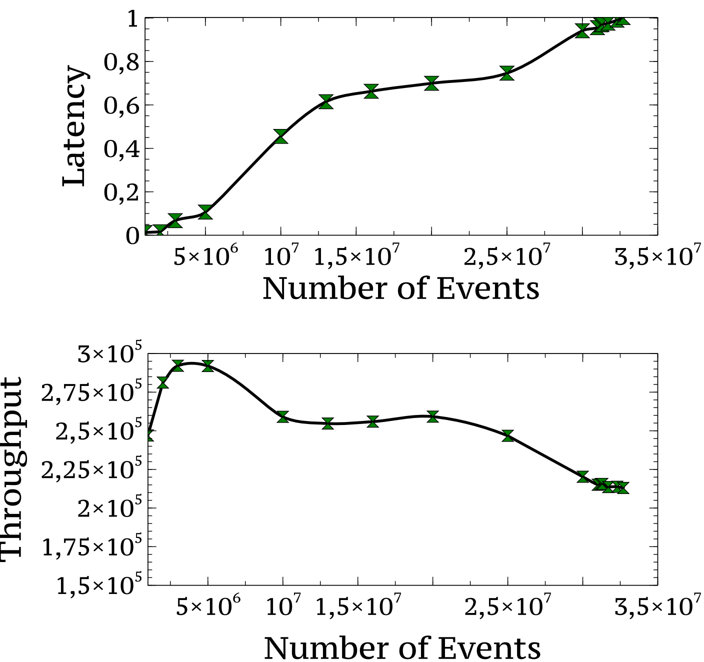


Figure 5.5: Buffered Linked Blocking Queue.

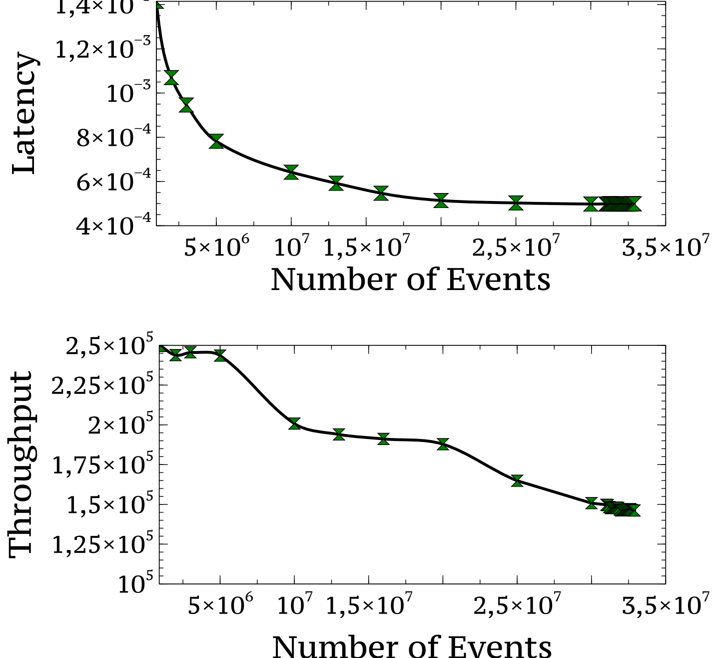


Figure 5.6: Linked Blocking Queue.

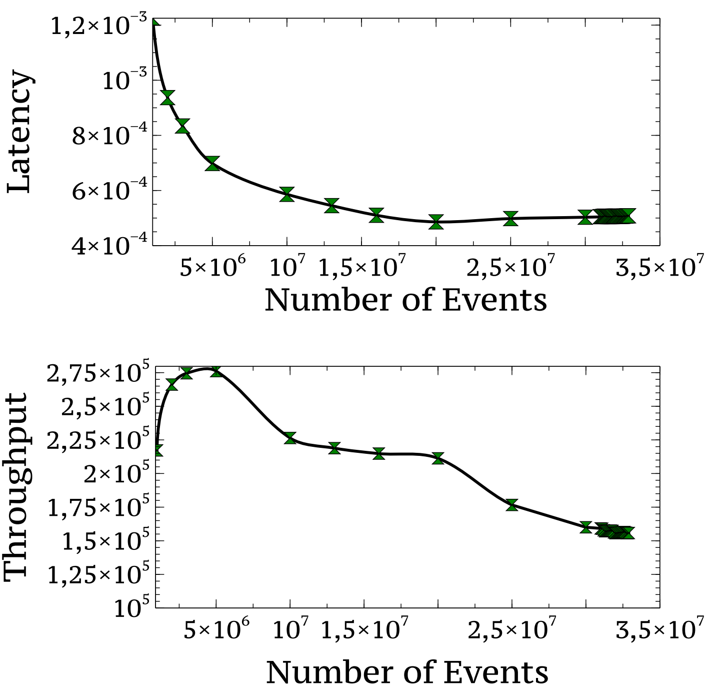


Figure 5.7: Lock Free Blocking Queue.

5.6.4 Analysis of Query 2 Performance

Table 5.2 presents the performance of Query 2 with respect to the changes of values k and the size of the window. Different queue choices also had an effect on Query 2 performance, and the observations of different queues can be directly borrowed from the analysis of Query 1. For example, consider the performance measures of BLBQ in Table 5.2. It provides the best throughput as compared to other queues. It results in higher latency values however. We also use LFQ for Query 2 in the submission of Grand Challenge solution. Second, the value of k does not have any significant effect on the query performance. The most noteworthy, however, of the values of k are the lowest ones, as the query processing is faster for the small values of k. This is because for a small value of k, each newly arrived comment can become a contender for the top-k spot. This results in the reshuffling of comments and computations of cliques resulting in the worst-case in terms of performance. Third, the query performance does not degrade linearly with the increase in the window size. Such behaviour is attributed to our lazy evaluation technique for clique computations. Thus, deferring the complete computation of a clique does not waste extra computer resources and results in lower latency values and higher throughput. Clique is calculated if either some elements are removed from the top-k data structure or if the changes in the friendship graph surpass the threshold.

5.7 Conclusion

In this chapter, we described our solution to the 2016 edition of the DEBS Grand Challenge. In the design of our solution, we carefully took care of optimizing every part of the system. Fast data parser, efficient message-passing queues as well as devising efficient fast lower and upper bounds to avoid costly computation, were the keys to the success of our approach. Similarly, the choices and designs behind the most frequently used data structures (sorted elements, friendship graph) largely contributed to overall system performance. On the basis of this experience, we have realized that Programming languages have evolved to answer the need for data stream processing. Be it with Domain Specific Languages [Bos+14; Su+14; TKA02], language extensions [SB13; EJ09] or with the developement of standard APIs like *Stream* for Java. This field demonstrated many advances in the last few years. However, the existing data structures of these languages have been designed for static data processing and their correct use with evolving data is cumbersome – top-k query processing requires maintaining sorted collections. We identified that maintaining sorted collections of dynamic data is particularly error-prone and leads to hard-to-detect bugs. Thus, in the Appendix A, we tackle the issue of maintaining dynamically sorted collections in Java in a safe and transparent manner for the application developer. For this purpose, we developed an annotation-based approach called `Upsortable` – a portmanteau of update and sort – that uses compilation-time abstract syntax tree modifications and runtime bytecode analysis. `Upsortable` is fully compatible with standard Java and is therefore available to the greatest number of developers.

6
A Scalable Framework for Accelerating Situation Prediction over Spatio-temporal Event Streams

Contents

6.1 Introduction 123
6.2 Input Data Streams and Query definitions 125
6.2.1 Input Data Streams 125
6.2.2 Query 1: Predicting destinations of vessels 126
6.2.3 Query 2: Predicting arrival times of vessels 126
6.3 Preliminaires 127
6.4 The Framework 127
6.5 Experimental Evaluation 129
6.5.1 Evaluation [Gul+18b] 129
6.5.2 Results and GC Benchmark 130
6.6 Conclusion 131

6.1 Introduction

The tremendous increase in the use of cellular phones, GPS-like devices and RFIDs has resulted in the rapid increase in new spatiotemporal applications. Examples of these applications include traffic monitoring, supply chain management, enhanced 911 services, etc. These applications continuously receive data from mobile objects in the form of event streams that are processed in a continuous manner. Another extension of these applications is to enable predictive analysis to envision proactive

functionality. This adds to the existing challenges of processing event streams, which are of high volume and velocity, with real-time responses.

This chapter deals with the DEBS Grand Challenge 2018, where we provide a generic solution to a naval transportation problem of situation prediction over spatiotemporal event streams. Our solution is an adjustment in the approach previously presented in chapter 4, used for the prediction of complex events in CEP systems. Note that the main common feature between the two problems is the use of multidimensional data in a stream environment. The grand challenge is composed of two queries. Query 1 (**Q1**) predicts the correct destinations of vessels given information such as vessel id, timestamp and the current position of the vessel. Query 2 (**Q2**) predicts the arrival time at a destination port given a vessel's respective bounding boxes of coordinates. Given these queries, the traditional solution would involve techniques such as All-K-Order-Markov [PP99] and Probabilistic Suffix Tree [BYY04], Naive Bayes [MKN10], Support Vector Machines [Par+17], etc. Some of these techniques are based on the Markov property, where k recent events from the training datasets are used to perform the prediction. Increasing k would exponentially increase the state complexity. Therefore, these techniques forget older events and they are not taken into account for prediction. Furthermore, these techniques require customise tweaking for each use cases and datasets, hence cannot be generalised.

Our work has two aims: first to provide a generic solution that is not only applicable for DEBS Grand Challenge 2018, but can also be extended for other use cases for spatiotemporal predictions. Second, we wanted to take care of older events in the event streams that result in optimised precision. To materialise these aims, we designed an optimised storage framework for storing historical information from the training dataset and event streams. Our storage framework employs space-filling curve (Z-order) to reduce the number of N dimensions to one dimension. This enables us to use a linear data structure, i.e. ph-tree[ZZN14], for optimised continuous insertions and querying over the event streams.

Our contributions are described as follows:

- Instead of relying on existing machine learning algorithms, we employ novel multi-dimensional indexing optimised for event streams.
- Our proposed indexing solution enables optimised online learning from the event streams.
- Based on our solution, the prediction is not just based on recent events but also the older and important events.
- Our proposed system is memory and CPU efficient, hence can be used in resource constraint environment.

6.2 Input Data Streams and Query definitions

The challenge was co-organized by MarineTraffic, the BigDataOcean project and the HOBBIT [1] project represented by AGT International [2]. Grand Challenge data was provided by MarineTraffic and hosted by Big Data Ocean while the automated evaluation platform was provided by the HOBBIT project.

6.2.1 Input Data Streams

The data is an anonymized collection of sensor data from vessels' Automated Identification System (AIS), the internationally accepted standard for self-identification of big vessels. AIS transponders periodically send time-series data points to coastal-, aircraft- or satellite-based AIS receivers. Data points of the provided dataset contain static information (ship identifier, ship type) as well as dynamic one (coordinates, speed, heading, course, draught, departure port name).Figure 6.1, shows the vessels' trajectories in graphical form, where each colour represents a starting port and each circle is a port. The schema of the tuples in the dataset is composed by attributes:

- *SHIP ID*, the anonymized id of the ship
- *SHIP TYPE*, the vessel type
- *SPEED*, speed measured in knots
- *LON*, the longitude of the current ship position
- *LAT*, the latitude of the current ship position
- *COURSE*, the direction in which the ship moves
- *HEADING*, the cardinal direction in which the ship is to be steered [3]
- *TIMESTAMP*, the time at which the message was sent (UTC)
- *DEPARTURE PORT NAME*, the name of the last port visited by the vessel
- *REPORTED DRAUGHT*, the vertical distance between the waterline and the bottom of the ship's hull

[1] https://project-hobbit.eu/
[2] http://www.agtinternational.com/
[3] https://en.wikipedia.org/wiki/Course_(navigation)

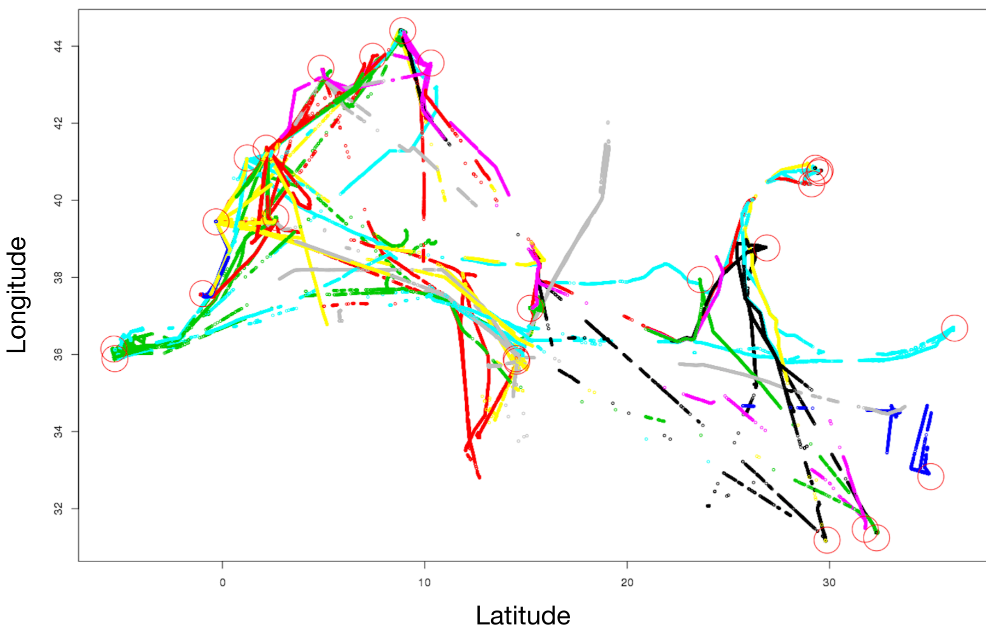


Figure 6.1: Vessels trajectories between ports

6.2.2 Query 1: Predicting destinations of vessels

Predicting the correct destination of a vessel is a relevant problem for a wide range of stakeholders including port authorities, vessel operators and many more. The prediction problem is to generate a continuous stream of predictions for the destination port of any vessel given the input data streams. The above data is provided as a continuous stream of tuples and the goal of the system is to provide for every input tuple, one output tuple containing the name of the destination port.

A solution is considered correct at time stamp t if the predicted destination port matches the actual destination port for a tuple with this timestamp as well as for all subsequent tuples. The goal of any solution is not only to predict the correct destination port but also to predict it as soon as possible, counting from the moment when a new port of origin appears for a given vessel. From port departure to arrival, the solution must emit one prediction per position update.

6.2.3 Query 2: Predicting arrival times of vessels

There is a set of ports defined by respective bounding boxes of coordinates. Once a ship leaves a port (i.e. the respective bounding box), the task is to predict the

arrival time at its destination port (i.e. when the next defined bounding boxes will be entered). Like for query 1, from port departure to arrival, the solution must emit one prediction per position update.

6.3 Preliminaires

In this section, we introduce the notion used in the paper and provide the background on the datasets; queries used to evaluate our framework and the space-filling curves for indexing.

Notations. Let $E = [e_1, e_2, \ldots e_i]$ donates a discrete stream of events and e_i denotes an individual event. Each event is a tuple (A, t), where A is a set of attributes, such as vessel id, vessel location, port, etc., and t is the associated timestamp of the event. Let $H = \{A_1 \times A_2 \times, \ldots, A_n\}$ is an N-D *lattice* for the universe of all the events in the training dataset, where an n-tuple $X = (x_1, x_2, \ldots, x_n)$ defines a point in $\mathcal{H}$ and $x_i \in A_i \; \forall i \in n$. This N-D lattice is used to find the nearest points given the input tasks for both queries for prediction.

Space-filling Curve (Z-order). As discussed in chapter 3, the Z-order curve preserves the proximity properties among the points while leveraging the effectiveness of linear data structures (such as B+tree) for range queries. The construction of the Z-order curve, i.e, Z-values generation, is accomplished by the simple process of *bit-shuffling* and this is one of its main advantages over other space-filling curves. Considering this, we have employed the Z-order curve to map all the attributes assigned to an event. The resulted mapping is added in the ph-tree for efficient storage and processing of the given queries.

6.4 The Framework

In this section, we present our framework and the evaluation of DEBS Grand Challenge 2018 queries. Fig.6.2 shows all the components of our framework. The input manager parses the training dataset and incoming event streams before indexing it. The historical store employs the Ph-Tree to index the main attributes including longitude, latitude, heading, course, speed and reported draught. The score index is used to prioritise the vessels that are custom to follow the same routine. That is, if we are asked to predict the destination of a vessel A, yet vessel A is usually following a certain path, then we give priority to this vessel rather than ones which usually deviate from their path. From Fig.6.3, $P1$ is the new event to find the destination of a vessel. Then from the historical index, we have three possible destinations for such vessel. Hence, we update the destination and

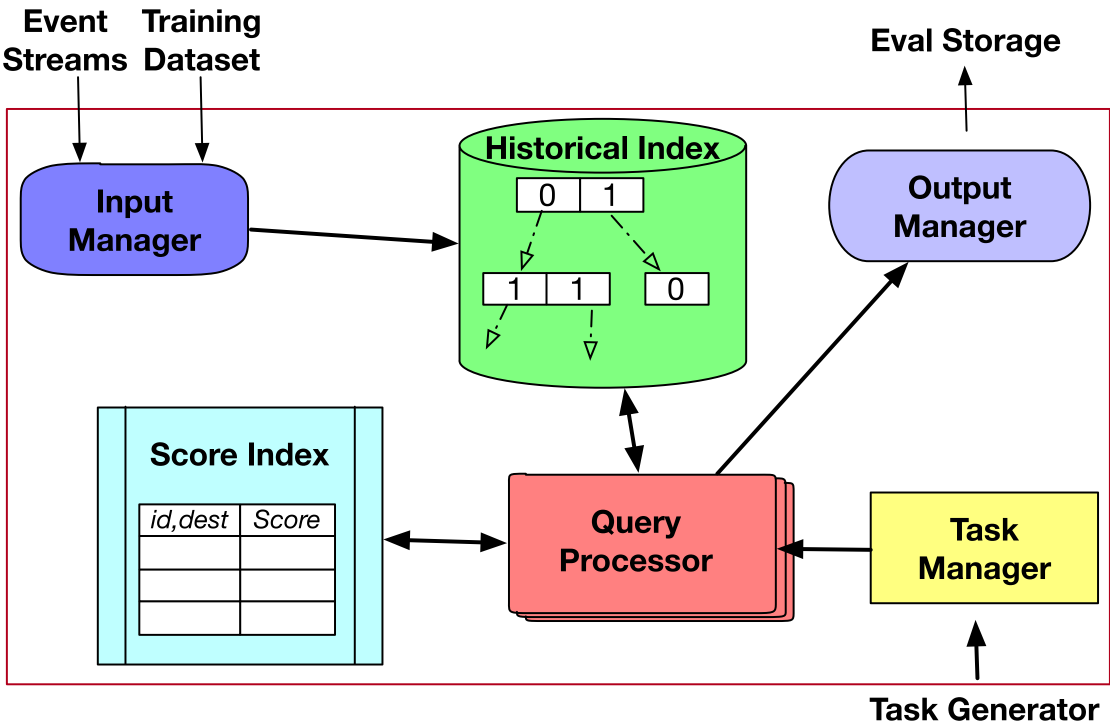
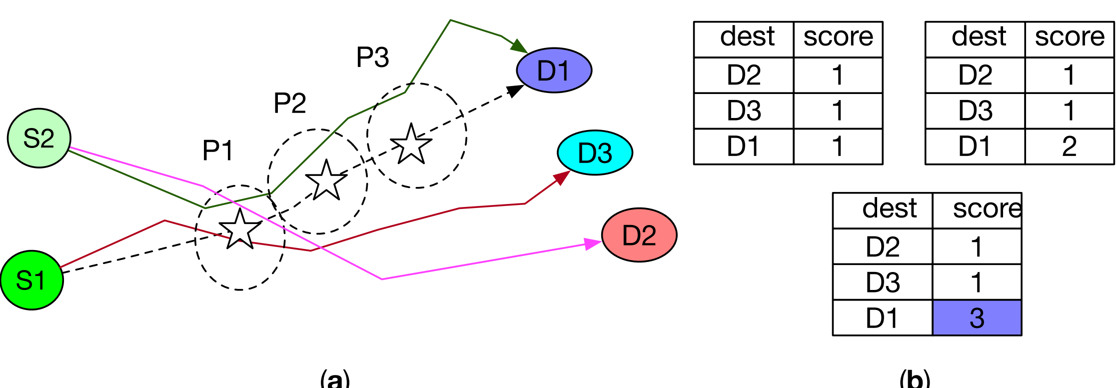


Figure 6.2: The use of thescore and historical index to predict the destination of a vessel

score table. At $P2$, the destination $D1$ persists from the initial computation and is updated, hence it is more reliable than others. Finally, at $P3$, we are sure that the vessel is heading for the destination $D1$ in our index table. Moreover, for **Q2**, we use the time difference at nearest found vessels (at $P1$, $P2$, or $P3$) to predict the arrival time for a vessel. Using this we can even predict the destination of



dest	score
D2	1
D3	1
D1	1

dest	score
D2	1
D3	1
D1	2

dest	score
D2	1
D3	1
D1	3

Figure 6.3: System Design for predicting vessels' destinations and arrival time

the vessel, even if it has not been visited before by that particular vessel with a departure from the same port. The query processor is in-charge of taking tasks from the task manager and producing a range query over the historical index to

find the nearest points of interest. Based on the resulting set of points, it employs the score index to predict the destination and time of arrival for a vessel. The predicted output is fed to the output manager which uses the eval storage of the hobbit framework to determine the precision of the computed results.

Herein, we briefly describe the inner workings of our indexing and query processing to extract the nearest point of interest for prediction. Given the training dataset and event streams, we need to create an N-D lattice. To accomplish this, we first convert the N-D points to single dimension points using Z-order curve as described earlier. These points are added into a ph-tree, where the bit locations of the points are used to cluster the N-D points in the tree. With the arrival of a new task, we create a point query and then find the nearest neighbours of these points using the Euclidean distance of each dimension. This task can be efficiently done by comparing N-D points at bit-level.

6.5 Experimental Evaluation

Herein, we describe the set of evaluation metrics defined in this challenge and then discuss the results of the participating solutions.

6.5.1 Evaluation [Gul+18b]

The Q1 score is calculated based on both how soon the correct predictions are made (rank A1) and how long the system will take to complete (rank B1). Rank A1 is the average time between a prediction and arrival at port. Only correct predictions are taken into account. Arrival at a port is defined by the first event that is reported inside the corresponding delimitation box. More formally:

$$Average\ Earliness\ Rate = \sum_{k=0}^{N\ Trips} \frac{\frac{length(of\ last\ correct\ sequence)}{length(of\ trip\ sequence)}}{N\ trips}$$

Score A1 = offset of the first tuple of the last correctly predicted sequence before trip ends / total trip duration. Offset = tripEndTimestamp − FirstCorrectTupleTimestamp.

Example Score A1:

- Time:01, Predicted Destination: A (Start of Trip)
- Time:02, Predicted Destination: B
- Time:03, Predicted Destination: A
- Time:04, Predicted Destination: B
- Time:05, Predicted Destination: B

- Time:06, Predicted Destination: B (Arrival at B)

Score A1: (06-04)/(06-01) [higher is better] The overall ranking for query 1 (Rank Q1) is then computed as Rank Q1 = 0.75*Rank A1 + 0.25*Rank B1.

The Q2 evaluation is calculated based on the accuracy of predictions (Rank A2) and the total runtime (Rank B2). Score A2 is the mean average error of all predicted arrival times while Rank B2 ranks according to the total runtime. More formally:

$$Mean\ Average\ Error = \sum_{k=0}^{Ntuples} \frac{|error\ per\ tuple|}{Ntuples}$$

The overall ranking for Query 2 (Rank Q2) is then computed as Rank Q2 = 0.75*Rank A2 + 0.25*Rank B2. Finally, The final ranking is given by the sum of ranks Rank Q1 and Rank Q2.

6.5.2 Results and GC Benchmark

We use the given dataset and the two queries from the DEBS Grand Challenge 2018. Herein, we showcase how we can use a subset of the training dataset to predict the destination port of the vessels with the arrival of new tasks. Fig.6.4 shows that general accuracy and average earliness rate in relation with the percentage of data used from the training dataset for prediction task. Thus, even with a small percentage of training dataset, we are able to have a higher average earliness rate and general accuracy.

$$General\ Average = \frac{Correct\ predictions}{totalnumber\ of\ predictions}$$

The following table 6.1 summarizes the best participants' results, where two types of systems are found, in which machine learning approaches[NVA18; Bac+18; Ros+18; Bod+18] and indexing [Ama+18] are used. Various experiments have been carried out using different real datasets. [NVA18] (MT Detector) uses a sequence-to-sequence model based on a Long Short Term Memory (LSTM) network, [Ros+18] (KNN) uses a nearest neighbour search to find the training routes that are closer to the AIS query point based on LAT/LONG only, [Bod+18] (VEL) uses a Voting Ensemble Learning based on Random Forest: Gradient Boosting Decision Trees (GBDT), XGBoot Trees and Extremely Randomized Trees (ERT), and the [Bac+18] (Venilia) prediction mechanism is based on a variety of machine learning techniques including Markov models and supports on-line continuous training. The two best systems were based on indexing techniques, where the first (CellGrid) was based on a sequence of hash tables specifically built for the targeted use case and the second was our system (TrajectPM).

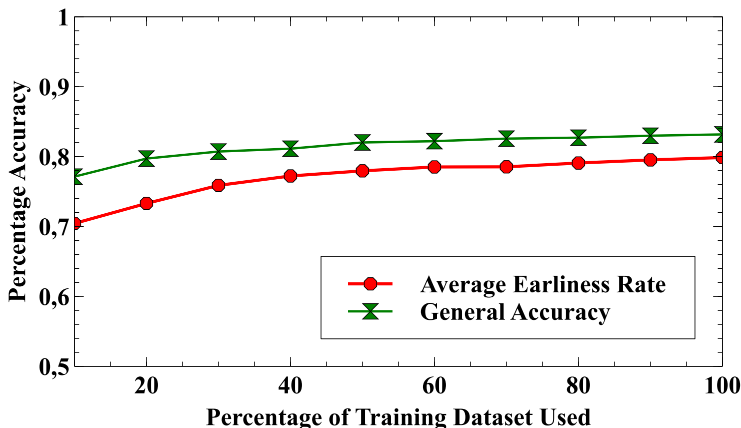


Figure 6.4: Comparing average earliness rate and general accuracy with the percentage of training dataset used

	MT Detector [NVA18]	CellGrid [Ama+18]	KNN [Ros+18]	Venilia [Bac+18]	VEL [Bod+18]	TrajectPM
A1	0,54	0,685	0,647	0,5	0,653	0,668
B1 seconds	86	99	157	129	102	102
AVG Latency ms	36	ns	ns	ns	ns	ns
A2	5193	894	2193	1482	1328	1002
B2 seconds	95	108	180	147	109	165
AVG Latency ms	41	117	51	38	48	93

Table 6.1: DEBS GC Results from Hobbit platform (ns: nano seconds)

6.6 Conclusion

This Challenge focused on spatio-temporal streaming data focusing on the combination of event processing and predictive analytics.

We employed novel multi-dimensional indexing optimised for event streams instead of relying on existing machine learning. In the results we outperformed many approaches which used on-shelf data mining and machine learning algorithms.

In conclusion, we believe that the main strengths of our architecture is its more generic approach, its suitability for a range of diverse real-life scenarios and its ability to support on-line continuous training.

Part IV

Conclusion

7 Conclusion and Future Works

In this dissertation, we focused on enhancing the capabilities of Complex Event Processing (CEP) systems, whose goal is to detect patterns over incoming event streams. Existing CEP systems suffer from poor performance due to high memory and high computation resource consumption for expensive pattern query operators such as Kleene+ and Skip-Till-Any. In addition most of them cannot provide predictive behaviour. The main contributions of this thesis is to propose a new breed of CEP systems that can handle higher amounts of data for detecting complex patterns and integrate more intelligent predictive behaviour. The main results are described in the following text.

7.1 Enhancing CEP performance

In order to enhance the performance of CEP systems, we focused on optimizing the memory and CPU performance for expensive queries. We provided multiple optimization techniques. Our approaches leveraged concepts and methods from multidimensional indexing techniques, dominance relationship analysis and batch processing.

We provided a novel recomputation approach called RCEP for processing expressive CEP queries. RCEP eliminates the need for storing partial matches which were identified as the bottleneck of existing incremental approaches.

To efficiently store and process multi-attribute events, we employed space-filling curves combined with B+tree indexing techniques. This enabled us to use efficient multidimensional indexing in streaming environments. We then exploited multidimensional indexing properties by constructing range queries to perform a range search and retrieve potential events that can be part of complete matches.

To produce final matches from range query results, we proposed a heuristic based join algorithm. Our algorithm leverages the dominance property between set of events and joins the clustered events in batches to reduce the cost of pair-wise joins. Additionally, dominance property allows to reuse previously computed joins to further economize on recomputation of matches.

We proposed a transformation of online window management to batch-based window management. Where instead of using traditional incremental deletions of events outside the sliding window, we implemented batch deletion of expired events. We called it CPU Friendly method which avoids the rebuilding cost of the B+Tree in the streaming settings. This CPU friendly approach results in a slight increase in memory consumption because of having to keep the older events, nevertheless, it is still far better in memory performance as compared to when maintaining the partial matches. Various experiments using various datasets have demonstrated that our system outperforms existing approaches by several orders of magnitude and consume less memory.

Perspectives: *To go further, the work can be extended by exploring multi-query optimizations, by exploiting sharing opportunities such as sharing the results of one sub-query by another to reduce computational cost or by decomposing queries. Another future idea is to treat the case of lossy data or out of order data. The out of order problem can be easily solved by our system as our approach is based on lazy approach. Our system waits till the final matching event arrives before processing the relevant events. Thus, any out of order event will eventually arrive and will be enqueued for being processed at a later time when the final matching event arrives.*

7.2 Predictive CEP

Continuing in the direction of improving the functionalities of the CEP systems, we investigated an open issue of designing a predictive CEP which permits better understanding and proactively acting in response to potential upcoming future complex events.

We addressed this issue from a new perspective, with the main objective of bringing the real-time predictive CEP capabilities to the database layer. For this objective, we extended the existing data structures, query execution and optimization techniques.

Predictive CEP problem has a parallel in the domain of sequence prediction. Some existing approaches for sequence prediction based on Markov Models depend only on the previous state. Other models, including also Markov models, suffer from catastrophic forgetting of older sequences, where only some recent items of

training sequences are used to perform predictions. This makes these approaches as lossy as some of the past information relevant for prediction is lost. Such lossy techniques impact the prediction accuracy. Prediction with multidimensional sequences becomes even worse.

In order to design an efficient predictive CEP, we used the following approaches: First, we used a memory efficient data structure to index all historic sequences. This data structure allowed our predictor to be updated, in real time, at each step of new sequence detection. Secondly, we designed algorithms to perform fast as well as accurate sequence prediction and handle complex data, i.e., multidimensional sequences. Thirdly, in order to be memory efficient for storing the historical sequences we proposed a summarisation approach, that results on minor loss of information. Nevertheless, our algorithm looses only non important information by tracking the less frequently retrieved or used historical sequences. We thus proposed a generic solution for incorporating Predictive Analytics into any Complex Event Processing system.

Preliminary results, show that our system outperforms an existing recent sequence prediction approach in terms of accuracy and execution time.

Perspectives: *In future, we would like to explore the integration of machine learning techniques with our approach. Machine learning can help CEP in automatisation of some processes like automatic query-pattern generation. Constructing query-patterns is a difficult task which needs the intervention of an expert. Moreover, we only output the future possible matches, but we do not provide the probability of their occurrence. We would like to explore machine learning techniques to output the occurrence probability.*

7.3 Real world use cases and challenges

As a final contribution, we showed the ability of our approaches to be applied on real use cases and different challenges proposed by the research community. Such challenges provided a common ground for us to evaluate and compare our event-driven systems with other systems of other researchers.

We participated to the 2016 DEBS Grand Challenge, which focused on reasoning over social networks data to drive meaningful insights from it in real-time. One novelty was the non-linearity of the expiration of the elements. We proposed several efficient and effective techniques to enable a lazy evaluation of non-linear window elements. In another part of the problem, we had to manage a graph of ever growing social network data. Due to the centrality of this structure, a careful design was therefore required.

In the design of our solution, we carefully took care of optimizing every part of the system. Fast data parser, efficient message-passing queues as well as devising efficient fast lower and upper bounds to avoid costly computation, were the keys to the success of our approach. Similarly, the choice and designs behind the most frequently data structures (sorted elements, graph) largely contributed to the overall system performance.

The 2018 DEBS Grand Challenge focused on spatio-temporal streaming data. This challenge was about the combination of event processing and predictive analytics. The goal of the challenge was to perform spatio-temporal prediction.

We employed novel multi-dimensional indexing optimized for event streams instead of relying on existing machine learning. Our proposed solution had following advantages: it performed optimized online learning, prediction, the prediction was not just based on the recent events but also the older and important events, our solution was memory and CPU efficient.

In addition, the main strengths of our architecture was its suitability for many different real-life scenarios and its ability to support on-line continuous training.

In conclusion, we believe that real-time data processing is finally becoming an important part of data management domain. Our work has widespread impacts on a variety of applications: fraud detection, water management, event processing in different types of networks. The results of our research have been published in a number of international conferences.

Appendices

Upsortable an Annotation-Based Approach

Contents

A.1 Introduction **141**
A.2 The Case For `Upsortable` **142**
A.3 Upsortable solution **143**
A.3.1 AST modifications 144
A.3.2 Bookkeeping 144
A.3.3 Garbage Collection 146
A.4 Discussion **147**

A.1 Introduction

Top-k queries over data streams are well studied problems. There are numerous systems allowing the continuous processing of queries over sliding windows. At the opposite, non-append only streams call for ad-hoc solutions, e.g. tailor-made solutions implemented in a mainstream programming language. In the meantime, the *Stream* API and lambda expressions have been added in Java 8, thus gaining powerful operations for data stream processing. However, the Java *Collections Framework* does not provide data structures to safely and conveniently support sorted collections of evolving data. In this chapter, we explain `Upsortable`, an annotation-based approach to allow the use of existing sorted collections from the standard Java API for dynamic data management. Our approach relies on a combination of pre-compilation abstract syntax tree modifications and the runtime

analysis of bytecode. `Upsortable`[1] offers the developer a safe and time-efficient solution for developing top-k queries on data streams while maintaining a full compatibility with standard Java.

A.2 The Case For `Upsortable`

The standard Java *Collections* API contains three implementations of sorted data structures: the *java.util.TreeSet* backed by a Red-Black tree, the *java.util.PriorityQueue* that implements a priority heap and, for thread-safety purposes, the "*java.util.concurrent. ConcurrentSkipListSet*" implements a concurrent variant of Skip List. These structures especially implement `add` and `remove` primitives, as well as methods to navigate within these collections. These structures are therefore well-suited for the implementation of exact top-k queries: elements are kept sorted according to either a comparator provided at the creation time of the data structure or by the natural ordering of the elements. In both cases, a pairwise comparison method is used to sort the objects and this method must provide a total ordering. When dealing with data streams, the value of some fields of an object are subject to evolution and this may require a reordering within the collections this object belongs to. With the aforementioned sorted data structures – as well as third-party Java Collections APIs such as Guava[2] or Eclipse Collections[3] – the developer must first remove the object from each sorted collections, update its internal fields and reinsert the object in these collections. The sorted collections may otherwise become irredeemably broken. Figure A.1 depicts such an example. Hence, this `remove, update` and then *insert* process is very error-prone, especially in large codebases where objects belong to different sorted collections, depending on the state of the application. Broken sorted collections are also hard to identify at runtime and may go undetected for a while. This is typical for the top-k queries, where the collections might be broken after the k-th element. The behaviour of the corrupted data structure is unpredictable, it ranges from inconsistent results to wrong inserts and impossible removals – as depicted in Figure A.1 where the removal of D is impossible since it cannot be reached.

To circumvent this issue, the standard solution is to rely on the *Observer* design pattern. This pattern implies that the objects must keep track of the collections they belong to. This requires adding an extra data structure within the objects to store pointers to the collections they belong to. The field setters must be updated to remove, update and insert the object (acting as the *notify* in the pattern). Using

[1] `https://github.com/jsubercaze/Upsortable`
[2] `https://github.com/google/guava`
[3] `https://www.eclipse.org/collections/`

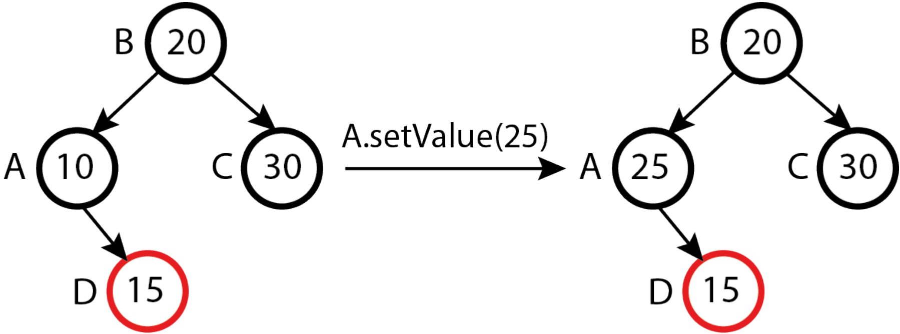


Figure A.1: Example of a corrupted Red-Black tree after update of Object *A* via call to its setter.

a dynamic array to store the pointer is the most compact way, however it may lead to useless removals and updates if the modified field does not participate in the comparison of some sorted structures that the objects belong to. Using a Hashmap circumvents this issue by mapping fields to the structures where the object belongs and where the fields participate in the comparison. However in both cases, when dealing with millions/billions of objects that are created and destroyed during the application lifetime, this solution has a very high memory cost. Moreover, it still requires heavy modifications of the source code by the application developer who must handcraft these routines for each object definition and for each setter.

Listing A.1: Annotation based solution

```
@Upsortable
public class MyObject {

    private int firstField;
    private String secondField;

}
```

A.3 Upsortable solution

Our solution proposes an alternative to the *Observer* pattern that does not require any other source code modification than adding an annotation and has a restricted memory fingerprint. The developer simply uses the `@Upsortable` annotation at the class level to declare that the internal fields are subject to modification and that the sorted collections it belongs to must be dynamically updated – such as depicted in Listing A.1. Our framework performs all the required updates to maintain the collections correctly sorted when setters update values in the object fields.

The underlying idea of our solution is that in real-time applications the number of sorted collections is very small compared to the number of objects that are sorted

within these collections – dozens against millions in practice. We leverage this imbalance to devise an approach that does not require any extra data structure to be added to the objects definition. Instead of linking objects to the collections they belong to, as in the *Observer* pattern, a global map links each field definition to the list of collections where it participates in the comparison process.

To relieve the developer from the burden of implementing this process, our framework consists of two parts: a transparent source code injection during the compilation phase and an encapsulation of the standard API sorted collections to automatically manage the global collection.

A.3.1 AST modifications

The Java compilation is a two-step process. The first step parses and compiles the source code and the second one processes the annotations. The Lombok project[4] has demonstrated the feasibility of modifying and recompiling the Abstract Syntax Tree (AST) during the second step, allowing annotations to transparently inject source code. Our framework, based on Lombok, injects setter methods for the classes annotated *@Upsortable*. The pseudo code of the setter method is given in Algorithm 6. The setter retrieves the sorted collections associated to the current field name – obtained via reflection – and performs the remove, update a ndinsert operations. The algorithm keeps track of the sets the current object participates in (by contract, *remove()* returns true if the object was present). As a consequence, the insertion of the updated object in the correct collections is guaranteed. Usage of WeakReference is detailed in Section A.3.3. Figure A.2 depicts the source code injection via AST modification during the annotation processing phase.

A.3.2 Bookkeeping

To keep track of the mappings between field names and the sorted collections, we encapsulate the creation of the sorted collections using the static factory pattern.

Listing A.2: Collection instantiation with upsortable

```
//Without upsortable
TreeSet<MyObject> mySet = new TreeSet<>(comparator);

//With upsortable
UpsortableSet<MyObject> mySet = Upsortables.newTreeSet(comparator);
```

We created a class called *Upsortables* which exposes static methods to create sorted structures backed by the standard Java API ones: *TreeSet*, *ConcurrentSkipList* and *PriorityQueue*. These static factory methods require the usage

[4] `https://projectlombok.org/`

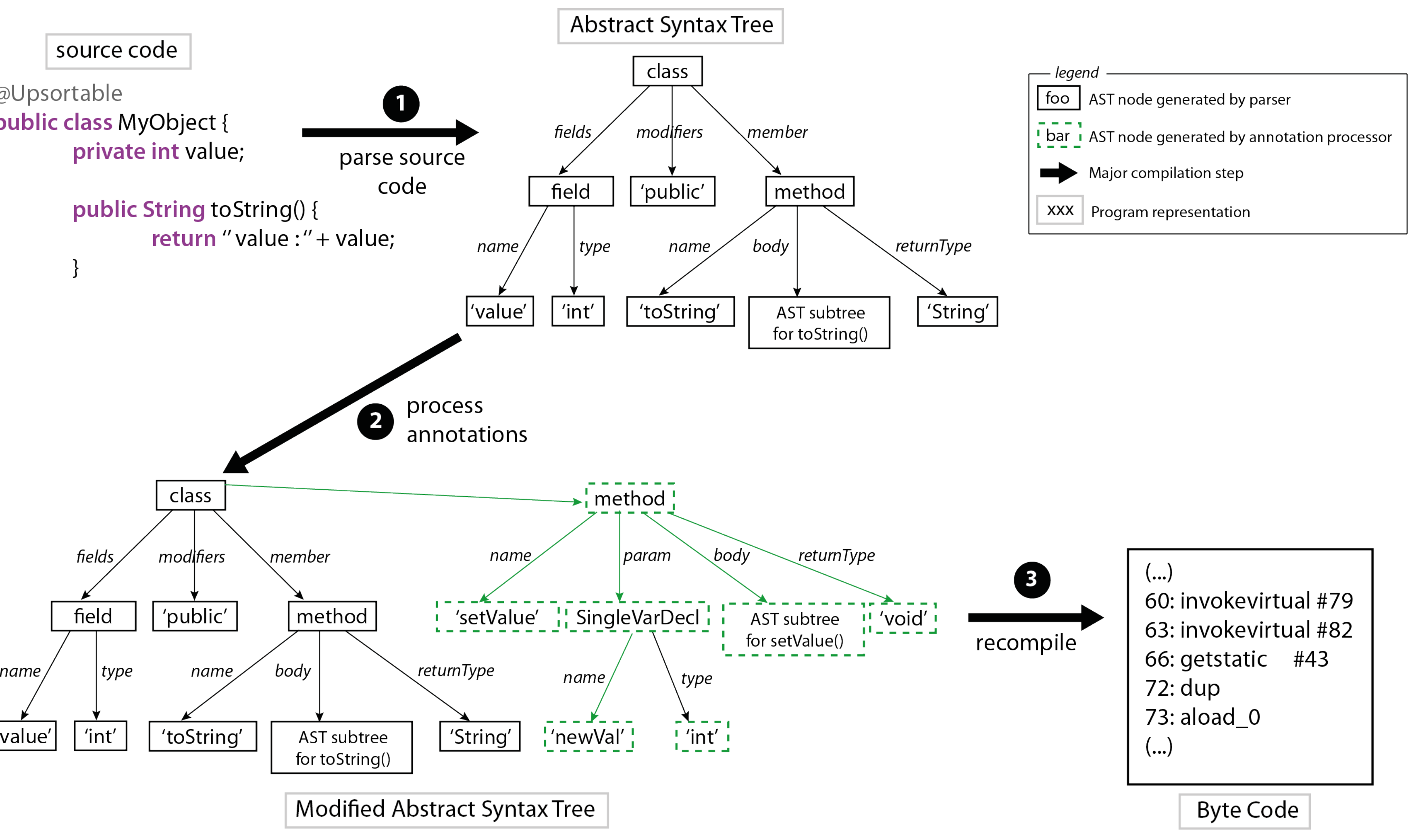


Figure A.2: Upsortable Abstract Syntax Tree modifications at annotation processing time.

Algorithm 6: Injected Setter code during annotation processing

Input: *newValue*: the new value of the field
1 ▷ Fails fast if unchanged **if** *this.field == newValue* **then**
2 return;
3 ▷ List of references to the collections concerned by this field
4 $refsList \leftarrow refMap.get(currentFieldName)$;
5 ▷ Remove *this* from the collections, remove cleaned references
6 $participatingCollections = newArrayList()$;
7 **for** $ref \in refsList$ **do**
8 **if** ref *is cleaned* **then**
9 remove from $refList$;
10 **if** $ref.deref().remove(this)$ **then**
11 participatingCollections.add($ref.deref()$);
12 ▷ Update the value
13 $this.field \leftarrow newValue$;
14 ▷ Reinsert in the right collections
15 **for** $collection \in participatingCollections$ **do**
16 $collection.add(this)$;

of comparators for the creation of sorted collections, disallowing the usage of natural ordering. The comparator implements per definition a *compare(MyObject o1, MyObject o2)* method. The static factory methods analyze the content of the *compare* method via runtime bytecode analysis in order to extract the fields of *MyObject* that participates in the comparison. For this purpose, we use Javassist, a common bytecode manipulation library. The extracted field names are then associated to the sorted collection that is being created in the global map. For performance reasons, we provide two versions of this global collection, one being thread-safe, the other not. On the developerside, besides the usage of the annotation, the sorted collection instantiation is the only modification, albeit minor, that is required to use `Upsortable`. Listing A.2 depicts the minor changes that this encapsulation implies. The burden on the developer side is therefore very limited and does not bring any particular difficulty.

A.3.3 Garbage Collection

Sorted collections may be created and deleted during the lifecycle of the application. Our framework shall therefore not interfere with the lifetime of these collections and shall especially not prevent them from being collected by the garbage collector (GC). To prevent the Hashmap that maps fields definitions to the Upsortable collections to hold a reference to these collections that would prevent their collection by the GC, we use a *WeakReference*. Contrary to soft references, weak ones do not interfere with the garbage collection of the objects they refer to. 'The injected setters' code takes care of removing weak references that have been cleaned up by the garbage

collector. By relying on the *ListIterator*, we are able to both process valid references and remove cleaned ones in a single iteration over the list of weak references.

A.4 Discussion

The *Upsortable* approach offers a convenient and safe solution to manage dynamically sorted collections. Naturally, safety and convenience have a performance impact. Keeping track of the relation between fields and sorted collections in *Upsortable* has a very limited memory fingerprint – especially compared to the *Observer* design pattern – and the CPU impact is also limited. Since we leverage the imbalance between the number of objects and collections, this leads to very few useless removes (a $O(log(n))$ operation for three data structures) and has a very limited impact of several percents ($< 5\%$) of the runtime in the practice, depending on the input data.

B
DETAILED ANALYSIS

Contents

B.1 Evaluating Kleene+ Operator 149
B.2 Proof Sketches 150
B.3 Optimising Z-address Comparison 153
B.4 Operations over Virtual Z-addresses 154
B.4.1 Correctness of Comparing Virtual Z-addresses 154
B.4.2 NextJumpIn and NextJumpOut Computation 155

In this appendix as mentioned before we provide detailed analysis on algorithms and proofs of the different theorem and lemmas defined in chapter3.

B.1 Evaluating Kleene+ Operator

The Kleene+ operator is executed after the joins evaluation and it employs the bitvector B_u and Banker's sequence for the set bits in B_u. An example of a Bankers sequence for three set bits for an events sequence $b^+ / \langle b_1, b_2, b_3 \rangle$ is shown in Table B.1.

The creation of sequences from the generated binary number requires mapping to the B_u when there are unset bits in B_u. That is, the join size is less than the total number of events in an events sequence. For instance, Fig. B.1 shows a bitvector $B_1 = 0100101_2$, where only three bits are set. Hence we need to map the index of the set bit with the generated Banker's sequence. Fig. B.1 shows such mapping for two sequences. Such mapping is part of the Algorithm 3, where at *line 9* a NextSetBit function is employed to skip the unset bits during the generation of the matches.

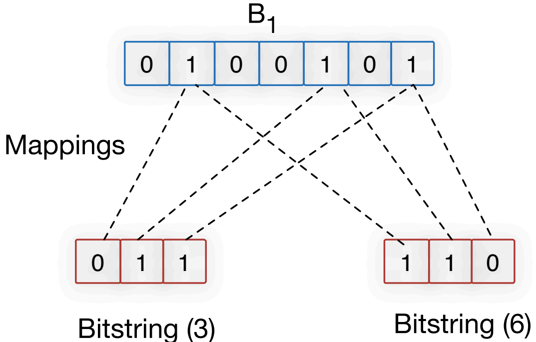


Figure B.1: Mapping of the Banker's sequences and a bitvector

Decimal	Binary	Sequence
1	001	$\langle b_3 \rangle$
2	010	$\langle b_2 \rangle$
3	011	$\langle b_2, b_3 \rangle$
4	100	$\langle b_1 \rangle$
5	101	$\langle b_1, b_3 \rangle$
6	110	$\langle b_1, b_2 \rangle$
7	111	$\langle b_1, b_2, b_3 \rangle$

Table B.1: Banker Sequence for an events sequence $\langle b_1, b_2, b_3 \rangle$

B.2 Proof Sketches

theorem 3

The total number of matches for n events with a CEP query Q are as follows [Agr+08]:

1. Polynomial n^k, where $k = |P|$ and $p^+ \notin P$
2. Exponential (2^n) for a $p^+ \in P$

Proof Sketch. For a given k variable bindings and their respective events sequences $\{p_1/\vec{e_1}, \ldots, p_k/\vec{e_k}\}$, we need to perform a cross-product (in the worst case) to extract all the possible matches with no strict time predicate. Hence for n events within a window divided into k events sequences, the possible number of matches amounts to n^k. The Kleene+ operator determines all the possible combinations or permutation of events within a sequence. Hence, the total number of such combinations would amount to 2^n for n events within a window for a qualified variable binding (p^+). □

lemma: 1

Given two Z-regions r and r' and a set of binary relations $\{\phi_1, \dots \phi_d\}$ from the variable predicates Θ^v in Q over d-dimensional events (Z-addresses), r dominates r' ($r \vdash r'$) iff

1. $\forall_i g^i_{d-1}(max(r))\ \phi_i\ g^i_{d-1}(min(r'))$ if $\phi_i = \{<, \leq\}$
2. $\forall_i g^i_{d-1}(min(r))\ \phi_i\ g^i_{d-1}(max(r'))$ if $\phi_i = \{>, \geq\}$

Proof. We prove the aforementioned points case by case. Let us assume a point z from region r and a point z' from region r'. Furthermore, for brevity, we use $z^l = min(r)$ and $z^h = max(r)$ The following is true:

$$\forall_i g^i_{d-1}(z^h_r) \leq g^i_{d-1}(z) \leq g^i_{d-1}(z^l_r) \tag{B.1}$$

$$\forall_i g^i_{d-1}(z^h_{r'}) \leq g^i_{d-1}(z') \leq g^i_{d-1}(z^l_{r'}) \tag{B.2}$$

since by definition, z^h_r and z^l_r represent the extremities of region r having maximum and minimum values in all dimensions and we follow the transitivity property and monotonic ordering of Z-addresses in a Z-region.

Case 1. In the case $i \in \Phi_{<,\leq}$ meaning $\phi_i = \{<, \leq\}$ and thus, from the assumed condition in the lemma, $g^i_{d-1}(z^h_r)\ \phi_i\ g^i_{d-1}(z^l_{r'})$ true. Which in turn means that $g^i_{d-1}(z^h_r) \leq g^i_{d-1}(z^l_{r'})$ or $g^i_{d-1}(z^h_r) < g^i_{d-1}(z^l_{r'})$ respectively depending on ϕ_i. Combining this with eq.(B.1) and (B.2), it follows that $g^i_{d-1}(z) \leq g^i_{d-1}(z')$ or $g^i_{d-1}(z) < g^i_{d-1}(z')$ depending on ϕ_i. Which means that for these i,

$g^i_{d-1}(z)\ \phi_i\ g^i_{d-1}(z')$ is true.

Case 2. Lets now take the case where $i \in \Phi_{>,\geq}$ meaning $\phi_i = \{>, \geq\}$ and thus, from the assumed condition in the lemma, $g^i_{d-1}(z^l_r)\ \phi_i\ g^i_{d-1}(z^h_{r'})$ is true. Which means that $g^i_{d-1}(z^l_r) \geq g^i_{d-1}(z^h_{r'})$ or $g^i_{d-1}(z^l_r) > g^i_{d-1}(z^h_{r'})$ respectively depending on ϕ_i. Combining this with eq.(B.1) and (B.2), it follows that $g^i_{d-1}(z) \geq g^i_{d-1}(z')$ or $g^i_{d-1}(z) > g^i_{d-1}(z')$ depending on ϕ_i. Which means that for these i, $g^i_{d-1}(z)\ \phi_i\ g^i_{d-1}(z')$ is true.

Thus, after combining cases 1 and 2, it follows that

$\forall_i\ g^i_{d-1}(z)\ \phi_i\ g^i_{d-1}(z')$,

which in turn forms Definition 5, which defines dominance, means that for any given point z from region r and z' from region r', given the above conditions, $z \vdash z'$ and hence $r \vdash r'$.

lemma: 2

Given two Z-regions r and r' and a set of binary relations from Lemma 1 r partially dominates r' ($r \sim r'$) iff Lemma 1 does not holds and only following conditions hold

1. $\forall_i g^i_{d-1}(min(r))\ \phi_i\ g^i_{d-1}(max(r'))$ if $\phi_i = \{<, \leq\}$
2. $\forall_i g^i_{d-1}(max(r))\ \phi_i\ g^i_{d-1}(min(r'))$ if $\phi_i = \{>, \geq\}$

PROOF. We prove the lemma as follows.
Case 1. In the case $i \in \Phi_{<,\leq}$ meaning $\phi_i = \{<, \leq\}$ and thus, from the assumed condition in the lemma, $g^i_{d-1}(z^l_r)\ \phi_i\ g^i_{d-1}(z^h_{r'})$ is true. Now lets take the region defined by the range having the following bits same as the extremities $g^i_{d-1}(z^l_r)$ and $g^i_{d-1}(z^h_{r'})$. Lets arbitrarily divide this region into two sub-regions s_1 and $s^{'}_1$. This is done by arbitrarily choosing 4 range values, lying inside the above range, such that the following is true:
$g^i_{d-1}(z^l_r)\ \phi_i\ g^i_{d-1}(z^l_{s_1})\ \phi_i\ g^i_{d-1}(z^h_{s_1})\ \phi_i\ g^i_{d-1}(z_{s^{'}_1})^l \phi_i\ g^i_{d-1}(z^h_{s^{'}_1})\ \phi_i\ g^i_{d-1}(z^h_{r'})$.

Also, the division is done in a way that the sub-regions are not empty and they only contain points from r and r'. There exists at least one division where sub-regions are not empty as, in the extreme case, the two regions can have at least a single common point each with the bits $g^i_{d-1}(z^l_r)$ and $g^i_{d-1}(z^h_{r'})$, respectively. From the above way of dividing, it is ensured that $g^i_{d-1}(z^h_{s_1})\ \phi_i\ g^i_{d-1}(z^l_{s^{'}_1})$.
Case 2. Now for other remaining $i \in \Phi_{>,\geq}$ meaning $\phi_i = \{>, \geq\}$ and thus, from the assumed condition in the lemma, $g^i_{d-1}(z^h_r)\ \phi_i\ g^i_{d-1}(z^l_{r'})$ is true. Again lets take the region defined by the range having the following bits same as the extremities $g^i_{d-1}(z^l_{r'})$ and $g^i_{d-1}(z^h_r)$. Lets arbitrarily divide it into two sub-regions s_2 and $s^{'}_2$. This is done by arbitrarily choosing 4 range values, lying inside the above range, such that the following is true:
$g^i_{d-1}(z^l_{r'})\ \phi_i\ g^i_{d-1}(z_{s^l_{2'}})\ \phi_i\ g^i_{d-1}(z_{s^h_{2'}})\ \phi_i\ g^i_{d-1}(z^l_{s_1})$
$\phi_i\ g^i_{d-1}(z^h_{s_1})\ \phi_i\ g^i_{d-1}(z^h_r)$.

Again, the division is done in a way that the sub-regions are not empty and they only contain points from r and r'. There exists at least one division where sub-regions are not empty as, in the extreme case, the two regions can have at least a single common point each with bits $g^i_{d-1}(z^h_r)$ and $g^i_{d-1}(z^l_{r'})$, respectively. From the above way of dividing, it is ensured that $g^i_{d-1}(z^l_{s_2})\ \phi_i\ g^i_{d-1}(z^h_{s^{'}_2})$.

Now, let's define a sub-region $s = s_1 \cap s_2$ and another sub-region $s' = s^{'}_1 \cap s^{'}_2$. The sets s and s' are non empty. As from the last lines of the above cases, s at least has a single point having bits $g^i_{d-1}(z^l_r) \forall i \in \Phi_{<,\leq}$ and $g^i_{d-1}(z^h_r) \forall i \in \Phi_{>,\geq}$. Similarly, s' at least has a single point having bits $g^i_{d-1}(z^h_{r'}) \forall i \in \Phi_{<,\leq}$ and $g^i_{d-1}(z^l_{r'}) \forall i \in \Phi_{>,\geq}$.

From the above 2 cases:

$\forall i,\ g^i_{d-1}(z^h_{s_1})\ \phi_i\ g^i_{d-1}(z^l_{s'_1})$ and $g^i_{d-1}(z^l_{s_2})\ \phi_i\ g^i_{d-1}(z^h_{s'_2})$ are true.

As s consists in the common points of s_1 and s_2, and s' consists in common points of s'_1 and s'_2, it implies that $\forall i,\ g^i_{d-1}(z^h_s)\ \phi_i\ g^i_{d-1}(z^l_{s'})$ and $g^i_{d-1}(z^l_s)\ \phi_i\ g^i_{d-1}(z^h_{s'})$ are true. Using lemma 2 $s \vdash s'$. Thus, there are some points in r that dominate some points in r' and hence $r \sim r'$.

B.3 Optimising Z-address Comparison

Herein, we describe an optimisation technique to save on the cost of the bit interleaving and comparison of Z-addresses during insertion, as well as range query processing over an event tree.

The aim of the bit-interleaving process is to compute the binary relations over Z-addresses while inserting and querying in an event tree. That is, locating the MSB difference between the Z-addresses in questions. For instance, if $z_1 = 001\mathbf{0}10$ and $z_2 = 001\mathbf{1}10$ then $z_2 > z_1$. This means we have to (i) generate a Z-address using bit interleaving for each event; (ii) de-interleave the Z-addresses to find the difference in the MSB between them; and (iii) decode the Z-address to get their original values. The first and third tasks can be performed using bit-level parallelism if the resulting bitstring (of a Z-address) fits in the registers of the CPU at hand, e.g. $|z| < 64$ bits on a 64-bit CPU. However, if $|z| > 64$, we need to iterate over the bits within each dimension in a traditional manner. Our aim is to avoid this by getting rid of bit interleaving and de-interleaving while still retaining the properties of Z-order curve.

Examining the comparison process between two Z-addresses (keys) reveals that we do not have to actually interleave bits to determine the MSB difference. Instead, we can directly compute and compare the MSB difference of each dimension (in keys) independently and then following up by comparing just the dimension that won the comparison, i.e. which has the leftmost MSB. The MSB comparison between a dimension from each key can be performed using a bitwise XOR operation. We call this process *virtual* Z-address comparison and, for brevity, Algorithm 7 presents this process for 2-dimensional keys.

Algorithm 7 takes the two keys and first calculates the bitwise XOR (denoted as $\veebar$) between the respective dimensions of both keys (*lines 1,2*). This provides the means to determine the position of the MSB contributed by both dimensions. It then finds out which dimension is responsible for the MSB bit in the virtual Z-addresses (*line 3*). If the XOR between the first dimensions of the keys is less than the other dimension, we are sure that the second dimension contributed to the MSB in the virtual Z-addresses (*lines 4-7*). Otherwise, it is the first dimension (*lines 9 -12*). Depending on the comparison result, it then checks which key's dimension (first or second) is greater or less than the other. This would provide us with the final results

Algorithm 7: Comparing two virtual Z-addresses

```
Input: Two keys X = (x1, x2) and Y= (y1, y2), each with two dimensions
1  MSB_d1 ← x1 ⊻ y1 ;                                  // bitwise XOR (⊻) operation
2  MSB_d2 ← x2 ⊻ y2 ;                                  // bitwise XOR (⊻) operation
3  if MSB_d1 < MSB_d2 And MSB_d1 < (MSB_d1 ⊻ MSB_d2) then
                // the second dimension contributes to the MSB;
4      if x2 < y2 then
5          Y is greater than X
6      else
7          X is greater than Y
8  else
                // the first dimension contributes to the MSB;
9      if x1 < y1 then
10         Y is greater than X
11     else
12         X is greater than Y
```

of which virtual Z-address is greater/less than the other. Note that Algorithm 7 can be extended to multiple dimensions by recursively checking all the dimensions to find out which dimension contributes to the MSB in the virtual Z-address. The use of virtual Z-addresses results in skipping the expensive process of bit interleaving and de-interleaving. Follows, the details of the correctness of Algorithm 7.

B.4 Operations over Virtual Z-addresses

Herein, first, we describe how Algorithm 7 correctly compares two virtual Z-addresses (keys) without going through the process of bit interleaving and de-interleaving. Second, we show how NextJumpIn and NextJump are calculated for false positive points in a range query over the Z-order curve.

B.4.1 Correctness of Comparing Virtual Z-addresses

Without loss of generality, let's discuss how we can compare the two integer values x and y using the position of the MSBs. If $x = y$ then both have the same location of MSB. However, if $x < y$ then y has the highest MSB set than x. Now consider how to compare x and y without comparing and calculating the position of the MSBs directly. Without loss of generality, let us assume that $x \leq y$. We have the two following cases.

1. If $x = y$ then both have the same MSB position. If we compute $x \veebar y$, the result is that it will have the same MSB set and $x < x \veebar y$ will be *false*.

2. If $x < y$ then x does not have the same position of the MSB than y, and the position of y MSB will be higher than x. Then if we compute $x \veebar y$, it will have same MSB of y and $x < x \veebar y$ will be true.

The logic of the above-mentioned cases is used to describe the working of Algorithm 7. In particular, *line 3* of Algorithm 7 which determines the dimension to compare in the virtual Z-address. Hence the first bitwise XOR ($\veebar$) operations between each of the dimensions in Algorithm 7 (*lines 1,2*) are used to compare the dimensions itself. That is, which dimension in the virtual Z-address would have contributed to the MSB: this comparison is done at the first part of the *if* statement on *line 3*. The second part of the *if* statement at *line 3* is then used to compare MSB difference between them, i.e. which dimensions have the highest MSB between the two virtual Z-addresses.

B.4.2 NextJumpIn and NextJumpOut Computation

Due to the nature of the Z-order curve, it contains jumps and can go out of the bounding box. Hence when calculating the range queries, we can have some false positive points, i.e. Z-addresses that are not part of the result, within the answer set. To resolve this problem, cursor-driven approaches are proposed to skip those false positive points. This includes calculating the boundaries of next set of elements that can be reached while skipping the false positive. Please refer to [OM84; Ram+00a] for the detailed discussion on this problem. Herein, we discuss how to calculate such boundaries without going through the process of bit interleaving and de-leaving. These boundaries are called NextJumpIn (NJI) and NextJumpOut (NJO).

Algorithm 8: NextJumpIn and NextJumpOut Computation

Input: A Range query with $q = [X, Y]$, X = (x_1, x_2) and Y= (y_1, y_2), each with two dimensions

1 **NJI** $\leftarrow X$;
2 **NJO** $\leftarrow Y$;
3 $dim \leftarrow$ DimToCompare(**NJI**, **NJO**);
4 $cmsb \leftarrow$ CommonMSB(**NJI**[dim], **NJO**[dim]);
5 $mask \leftarrow ((\neg(1 << (cmsb + 1)) - 1)$ & $0xFF$;
6 **NJI**[dim] $\leftarrow Y[dim]$ & $mask) \mid (1 << cmsb))$ `// bitwise OR (|) and shif operator (<<);`
7 **NJO**[dim] $\leftarrow Y[dim]$ & $mask) \mid (1 << cmsb) - 1)$

Algorithm 8 presents the procedure of calculating NJI and NJO for a range query. Without loss of generality, it shows the operations on 32 bits integers. It first initialises the NJI and NJO points with that of the range query boundaries. When it finds a value that comes out of the bounding boX. It then determines the dimension in both NJI and NJO that contributes to the MSB. Then computes the common MSB

for such dimension using both NJI and NJO. This common MSB is used to create a mask, which later will be used to create the new values of the dimension in question. Using the mask and common MSB, it changes the values of the dimension in both NJI and NJO. This results in the new range value that can skip the false positives.

Works Cited

[Aba+03] Daniel J Abadi et al. "Aurora: a new model and architecture for data stream management". In: *VLDBJ* (2003), pp. 120–139.

[Adi+06] Asaf Adi et al. "Complex event processing for financial services". In: *Services Computing Workshops, 2006. SCW'06. IEEE*. IEEE. 2006, pp. 7–12.

[Agr+08] Jagrati Agrawal et al. "Efficient Pattern Matching over Event Streams". In: *Proceedings of the 2008 ACM SIGMOD International Conference on Management of Data*. SIGMOD '08. Vancouver, Canada: ACM, 2008, pp. 147–160. URL: http://doi.acm.org/10.1145/1376616.1376634.

[AÇT08] Mert Akdere, Ugur Çetintemel, and Nesime Tatbul. "Plan-based complex event detection across distributed sources". In: *Proceedings of the VLDB Endowment* 1.1 (2008), pp. 66–77.

[AÇU10] Mert Akdere, Uur Çetintemel, and Eli Upfal. "Database-support for Continuous Prediction Queries over Streaming Data". In: *VLDB* (2010).

[AAP17] Elias Alevizos, Alexander Artikis, and George Paliouras. "Event Forecasting with Pattern Markov Chains". In: *Proceedings of the 11th ACM International Conference on Distributed and Event-based Systems*. DEBS '17. Barcelona, Spain: ACM, 2017, pp. 146–157. URL: http://doi.acm.org/10.1145/3093742.3093920.

[Ama+18] Ciprian Amariei et al. "Cell Grid Architecture for Maritime Route Prediction on AIS Data Streams". In: *Proceedings of the 12th ACM International Conference on Distributed and Event-based Systems, DEBS 2018, Hamilton, New Zealand, June 25-29, 2018*. 2018, pp. 202–204. URL: https://doi.org/10.1145/3210284.3220503.

[Ani+12] Darko Anicic et al. "Stream Reasoning and Complex Event Processing in ETALIS". In: *Semant. web* 3.4 (Oct. 2012), pp. 397–407. URL: http://dl.acm.org/citation.cfm?id=2590208.2590214.

[ABW06] Arvind Arasu, Shivnath Babu, and Jennifer Widom. "The CQL continuous query language: semantic foundations and query execution". In: *The VLDB Journal* 15.2 (2006), pp. 121–142.

[Ara+03] Arvind Arasu et al. "STREAM: the stanford stream data manager (demonstration description)". In: *Proceedings of the 2003 ACM SIGMOD international conference on Management of data*. ACM. 2003, pp. 665–665.

[Art+12] Alexander Artikis et al. "Event processing under uncertainty". In: *Proceedings of the 6th ACM International Conference on Distributed Event-Based Systems*. ACM. 2012, pp. 32–43.

[Art+17a] Alexander Artikis et al. “A Prototype for Credit Card Fraud Management: Industry Paper”. In: *DEBS*. 2017, pp. 249–260.

[Art+17b] Alexander Artikis et al. “A Prototype for Credit Card Fraud Management: Industry Paper”. In: *Proceedings of the 11th ACM International Conference on Distributed and Event-based Systems*. DEBS ’17. Barcelona, Spain: ACM, 2017, pp. 249–260. URL: http://doi.acm.org/10.1145/3093742.3093912.

[AH00] Ron Avnur and Joseph M Hellerstein. “Eddies: Continuously adaptive query processing”. In: *ACM sigmod record*. Vol. 29. 2. ACM. 2000, pp. 261–272.

[Bab+02] Brian Babcock et al. “Models and issues in data stream systems”. In: *Proceedings of the twenty-first ACM SIGMOD-SIGACT-SIGART symposium on Principles of database systems*. ACM. 2002, pp. 1–16.

[BW01] Shivnath Babu and Jennifer Widom. “Continuous queries over data streams”. In: *ACM Sigmod Record* 30.3 (2001), pp. 109–120.

[Bac+18] Moti Bachar et al. “Venilia, On-line Learning and Prediction of Vessel Destination”. In: *Proceedings of the 12th ACM International Conference on Distributed and Event-based Systems, DEBS 2018, Hamilton, New Zealand, June 25-29, 2018*. 2018, pp. 209–212. URL: https://doi.org/10.1145/3210284.3220505.

[BGH87] Joel Bartlett, Jim Gray, and Bob Horst. “Fault tolerance in tandem computer systems”. In: *The Evolution of Fault-Tolerant Computing*. Springer, 1987, pp. 55–76.

[BYY04] Ron Begleiter, Ran El-Yaniv, and Golan Yona. “On Prediction Using Variable Order Markov Models”. In: *J. Artif. Int. Res.* 22.1 (Dec. 2004), pp. 385–421. URL: http://dl.acm.org/citation.cfm?id=1622487.1622499.

[BF79] Jon Louis Bentley and Jerome H. Friedman. “Data Structures for Range Searching”. In: *ACM Comput. Surv.* 11.4 (Dec. 1979), pp. 397–409. URL: http://doi.acm.org/10.1145/356789.356797.

[BS80] Jon Louis Bentley and James B. Saxe. “Decomposable Searching Problems I: Static-to-Dynamic Transformation.” In: *Journal of Algorithms* 1.4 (1980), pp. 301–358. URL: http://dblp.uni-trier.de/db/journals/jal/jal1.html#BentleyS80.

[Bey+99] Kevin S. Beyer et al. “When Is ”Nearest Neighbor” Meaningful?” In: *ICDT*. 1999, pp. 217–235.

[BFP10] Silvia Bianchi, Pascal Felber, and Maria Gradinariu Potop-Butucaru. “Stabilizing distributed r-trees for peer-to-peer content routing”. In: *IEEE Transactions on Parallel and Distributed Systems* 21.8 (2010), pp. 1175–1187.

[Blo+10] Marion Blount et al. “Real-time analysis for intensive care: development and deployment of the artemis analytic system”. In: *IEEE Engineering in Medicine and Biology Magazine* 29.2 (2010), pp. 110–118.

[Bod+18] Oleh Bodunov et al. “Real-time Destination and ETA Prediction for Maritime Traffic”. In: *Proceedings of the 12th ACM International Conference on Distributed and Event-based Systems, DEBS 2018, Hamilton, New Zealand, June 25-29, 2018*. 2018, pp. 198–201. URL: https://doi.org/10.1145/3210284.3220502.

[Bos+14] Jeffrey Bosboom et al. “StreamJIT: A commensal compiler for high-performance stream programming”. In: *OOPSLA*. 2014, pp. 177–195.

[Bre+07] Lars Brenna et al. “Cayuga: A High-performance Event Processing Engine”. In: *SIGMOD*. 2007, pp. 1100–1102.

[Bui09] Hai-Lam Bui. “Survey and comparison of event query languages using practical examples”. In: *Ludwig Maximilian University of Munich* (2009).

[Cas+02] Miguel Castro et al. “SCRIBE: A large-scale and decentralized application-level multicast infrastructure”. In: *IEEE Journal on Selected Areas in communications* 20.8 (2002), pp. 1489–1499.

[Cha+14] Samy Chambi et al. “Better bitmap performance with Roaring bitmaps”. In: *CoRR* (2014).

[Cha+15] Samy Chambi et al. “Better bitmap performance with Roaring bitmaps”. In: *Software: Practice and Experience* (2015).

[CS94] Rakesh Chandra and Arie Segev. “Active databases for financial applications”. In: *Research Issues in Data Engineering, 1994. Active Database Systems. Proceedings Fourth International Workshop on*. IEEE. 1994, pp. 46–52.

[Cha+00] Yuan-Chi Chang et al. “The onion technique: indexing for linear optimization queries”. In: *ACM Sigmod Record*. Vol. 29. 2. ACM. 2000, pp. 391–402.

[Che+15] Jules Chevalier et al. “Slider: an Efficient Incremental Reasoner”. In: *SIGMOD*. ACM. 2015, pp. 1081–1086.

[CW84] John G. Cleary and Ian H. Witten. “Data Compression using Adaptive Coding and Partial String Matching”. In: *IEEE Transactions on Communications* 32.4 (1984), pp. 396–402.

[Cor11] Graham Cormode. “Sketch techniques for approximate query processing”. In: *Foundations and Trends in DB*. 2011.

[CDF01] Gianpaolo Cugola, Elisabetta Di Nitto, and Alfonso Fuggetta. “The JEDI event-based infrastructure and its application to the development of the OPSS WFMS”. In: *IEEE transactions on Software Engineering* 27.9 (2001), pp. 827–850.

[CM12a] Gianpaolo Cugola and Alessandro Margara. “Low latency complex event processing on parallel hardware”. In: *Journal of Parallel and Distributed Computing* 72.2 (2012), pp. 205–218.

[CM12b] Gianpaolo Cugola and Alessandro Margara. “Processing flows of information: From data stream to complex event processing”. In: *ACM Computing Surveys (CSUR)* 44.3 (2012), p. 15.

[CM15] Gianpaolo Cugola and Alessandro Margara. "The complex event processing paradigm". In: *Data Management in Pervasive Systems*. Springer, 2015, pp. 113–133.

[DM17] Tiziano De Matteis and Gabriele Mencagli. "Parallel patterns for window-based stateful operators on data streams: an algorithmic skeleton approach". In: *International Journal of Parallel Programming* 45.2 (2017), pp. 382–401.

[DIG07] Yanlei Diao, Neil Immerman, and Daniel Gyllstrom. "Sase+: An agile language for kleene closure over event streams". In: *UMass Technical Report* (2007).

[EE11] Yagil Engel and Opher Etzion. "Towards Proactive Event-driven Computing". In: *DEBS*. 2011, pp. 125–136.

[ESP] ESPER. http://www.espertech.com/esper/.

[EJ09] Patrick Eugster and KR Jayaram. "EventJava: An extension of Java for event correlation". In: *ECOOP*. Springer. 2009, pp. 570–594.

[Fli] Apache Flink. https://flink.apache.org/.

[FB12] Lajos Jenő Fülöp and Beszédes. "Predictive Complex Event Processing: A Conceptual Framework for Combining Complex Event Processing and Predictive Analytics". In: *BCI*. 2012.

[GUW02] Hector Garcia-Molina, Jeffrey D. Ullman, and Jennifer Widom. *Database systems - the complete book (international edition).* Pearson Education, 2002, pp. I–XXVII, 1–1119.

[GB01] Dimitrios Georgakopoulos and Alexander Buchmann, eds. *Proceedings of the 17th International Conference on Data Engineering, April 2-6, 2001, Heidelberg, Germany.* IEEE Computer Society, 2001. URL: http://ieeexplore.ieee.org/xpl/mostRecentIssue.jsp?punumber=7304.

[GMV08] John Giacomoni, Tipp Moseley, and Manish Vachharajani. "FastForward for efficient pipeline parallelism: a cache-optimized concurrent lock-free queue". In: *SIGPLAN*. ACM. 2008, pp. 43–52.

[Gue+15] Ted Gueniche et al. "CPT+: Decreasing the Time/Space Complexity of the Compact Prediction Tree". In: *PAKDD*. 2015, pp. 625–636. URL: http://dx.doi.org/10.1007/978-3-319-18032-8_49.

[Gul+16a] Vincenzo Gulisano et al. "The DEBS 2016 Grand Challenge". In: *to be published in the DEBS 2016 proceedings.* ACM. 2016, pp. 1–8.

[Gul+16b] Vincenzo Gulisano et al. "The DEBS 2016 grand challenge". In: *Proceedings of the 10th ACM International Conference on Distributed and Event-based Systems, DEBS '16, Irvine, CA, USA, June 20 - 24, 2016.* 2016, pp. 289–292. URL: http://doi.acm.org/10.1145/2933267.2933519.

[Gul+18a] Vincenzo Gulisano et al. "The DEBS 2018 Grand Challenge". In: *Proceedings of the 12th ACM International Conference on Distributed and Event-based Systems, DEBS 2018, Hamilton, New Zealand, June 25-29, 2018.* 2018, pp. 191–194. URL: http://doi.acm.org/10.1145/3210284.3220510.

[Gul+18b] Vincenzo Gulisano et al. “The DEBS 2018 Grand Challenge”. In: *Proceedings of the 12th ACM International Conference on Distributed and Event-based Systems, DEBS 2018, Hamilton, New Zealand, June 25-29, 2018*. 2018, pp. 191–194. URL: `https://doi.org/10.1145/3210284.3220510`.

[Hil91] David Hilbert. *Ueber stetige abbildung einer linie auf ein flachenstuck. Mathematische Annalen.* 1891.

[Hil+08] Matthew Hill et al. “Event detection in sensor networks for modern oil fields”. In: *Proceedings of the second international conference on Distributed event-based systems*. ACM. 2008, pp. 95–102.

[Hir+18] Martin Hirzel et al. “Stream Query Optimization”. In: (2018).

[HKP01] Vagelis Hristidis, Nick Koudas, and Yannis Papakonstantinou. “PREFER: A system for the efficient execution of multi-parametric ranked queries”. In: *ACM Sigmod Record*. Vol. 30. 2. ACM. 2001, pp. 259–270.

[Huf52] David A Huffman. “A method for the construction of minimum-redundancy codes”. In: *Proceedings of the IRE* 40.9 (1952), pp. 1098–1101.

[Kha+17] Zuhair Khayyat et al. “Fast and Scalable Inequality Joins”. In: *The VLDB Journal* (2017), pp. 125–150.

[Kir+17] James Kirkpatrick et al. “Overcoming catastrophic forgetting in neural networks”. In: *Proceedings of the national academy of sciences* (2017), p. 201611835.

[KS18a] Ilya Kolchinsky and Assaf Schuster. “Efficient Adaptive Detection of Complex Event Patterns”. In: *arXiv preprint arXiv:1801.08588* (2018).

[KS18b] Ilya Kolchinsky and Assaf Schuster. “Join Query Optimization Techniques for Complex Event Processing Applications”. In: *arXiv preprint arXiv:1801.09413* (2018).

[KSK16] Ilya Kolchinsky, Assaf Schuster, and Danny Keren. “Efficient Detection of Complex Event Patterns Using Lazy Chain Automata”. In: *arXiv preprint arXiv:1612.05110* (2016).

[KSS15a] Ilya Kolchinsky, Izchak Sharfman, and Assaf Schuster. “Lazy Evaluation Methods for Detecting Complex Events”. In: *DEBS*. 2015, pp. 34–45.

[KSS15b] Ilya Kolchinsky, Izchak Sharfman, and Assaf Schuster. “Lazy evaluation methods for detecting complex events”. In: *Proceedings of the 9th ACM International Conference on Distributed Event-Based Systems*. ACM. 2015, pp. 34–45.

[KS09] Jürgen Krämer and Bernhard Seeger. “Semantics and implementation of continuous sliding window queries over data streams”. In: *ACM Transactions on Database Systems (TODS)* 34.1 (2009), p. 4.

[Lee+07] Ken C. K. Lee et al. “Approaching the Skyline in Z Order”. In: *VLDB*. 2007, pp. 279–290.

[LKA10] Daniel Lemire, Owen Kaser, and Kamel Aouiche. “Sorting improves word-aligned bitmap indexes”. In: *Data & Knowledge Engineering* 69.1 (2010), pp. 3–28.

[LG16] Zheng Li and Tingjian Ge. “History is a mirror to the future: Best-effort approximate complex event matching with insufficient resources”. In: *Proceedings of the VLDB Endowment* 10.4 (2016), pp. 397–408.

[Lii91] Helmut Liitkepohl. “Introduction to multiple time series analysis”. In: *Berlin et al* (1991).

[Lin+05] Xuemin Lin et al. “Stabbing the sky: Efficient skyline computation over sliding windows”. In: *Data Engineering, 2005. ICDE 2005. Proceedings. 21st International Conference on.* IEEE. 2005, pp. 502–513.

[LvHS00] J. Loughry, J.I. van Hemert, and L. Schoofs. “Efficiently Enumerating the Subsets of a Set”. In: *Applied-math* (2000).

[Luc08] David Luckham. “The power of events: An introduction to complex event processing in distributed enterprise systems”. In: *International Workshop on Rules and Rule Markup Languages for the Semantic Web.* Springer. 2008, pp. 3–3.

[LF98] David C Luckham and Brian Frasca. “Complex event processing in distributed systems”. In: *Computer Systems Laboratory Technical Report CSL-TR-98-754. Stanford University, Stanford* 28 (1998).

[MCT14] Alessandro Margara, Gianpaolo Cugola, and Giordano Tamburrelli. “Learning from the past: automated rule generation for complex event processing”. In: *Proceedings of the 8th ACM International Conference on Distributed Event-Based Systems.* ACM. 2014, pp. 47–58.

[MKN10] Steven Mascaro, Kevin B Korb, and Ann E Nicholson. “Learning abnormal vessel behaviour from ais data with bayesian networks at two time scales”. In: *Tracks a Journal of Artists Writings* (2010), pp. 1–34.

[May18] Ruben Mayer. “Window-based data parallelization in complex event processing”. In: (2018).

[MD89] Dennis McCarthy and Umeshwar Dayal. “The architecture of an active database management system”. In: *ACM Sigmod Record.* Vol. 18. 2. ACM. 1989, pp. 215–224.

[MP13] Ciaran McCreesh and Patrick Prosser. “Multi-threading a state-of-the-art maximum clique algorithm”. In: *Algorithms* 6.4 (2013), pp. 618–635.

[MM09] Yuan Mei and Samuel Madden. “ZStream: A Cost-based Query Processor for Adaptively Detecting Composite Events”. In: *SIGMOD.* 2009, pp. 193–206.

[MRT12] Mehryar Mohri, Afshin Rostamizadeh, and Ameet Talwalkar. *Foundations of Machine Learning.* MIT Press, 2012.

[Mor66] G. M. Morton. “A Computer Oriented Geodetic Data Base; and a New Technique in File Sequencing”. In: *IBM.* 1966.

[MBP06] Kyriakos Mouratidis, Spiridon Bakiras, and Dimitris Papadias. “Continuous monitoring of top-k queries over sliding windows”. In: *SIGMOD.* 2006, pp. 635–646.

[MTZ17] Raef Mousheimish, Yehia Taher, and Karine Zeitouni. "Automatic learning of predictive cep rules: bridging the gap between data mining and complex event processing". In: *Proceedings of the 11th ACM International Conference on Distributed and Event-based Systems*. ACM. 2017, pp. 158–169.

[MZZ12] Barzan Mozafari, Kai Zeng, and Carlo Zaniolo. "High-performance Complex Event Processing over XML Streams". In: *Proceedings of the 2012 ACM SIGMOD International Conference on Management of Data*. SIGMOD '12. Scottsdale, Arizona, USA: ACM, 2012, pp. 253–264. URL: `http://doi.acm.org/10.1145/2213836.2213866`.

[MP12] Christopher Mutschler and Michael Philippsen. "Learning event detection rules with noise hidden markov models". In: *Adaptive Hardware and Systems (AHS), 2012 NASA/ESA Conference on*. IEEE. 2012, pp. 159–166.

[NCB15] Parth Nagarkar, K. Selçuk Candan, and Aneesha Bhat. "Compressed Spatial Hierarchical Bitmap (cSHB) Indexes for Efficiently Processing Spatial Range Query Workloads". In: *VLDB* (2015).

[NVA18] Duc-Duy Nguyen, Chan Le Van, and Muhammad Intizar Ali. "Vessel Destination and Arrival Time Prediction with Sequence-to-Sequence Models over Spatial Grid". In: *Proceedings of the 12th ACM International Conference on Distributed and Event-based Systems, DEBS 2018, Hamilton, New Zealand, June 25-29, 2018*. 2018, pp. 217–220. URL: `https://doi.org/10.1145/3210284.3220507`.

[NC07] Charles Nyce and A Cpcu. "Predictive analytics white paper". In: *American Institute for CPCU. Insurance Institute of America* (2007), pp. 9–10.

[OM84] J. A. Orenstein and T. H. Merrett. "A Class of Data Structures for Associative Searching". In: *PODS-SIGMOD*. 1984, pp. 181–190.

[Pal+18] Saravana Murthy Palanisamy et al. "Preserving Privacy and Quality of Service in Complex Event Processing through Event Reordering". In: *Proceedings of the 12th ACM International Conference on Distributed and Event-based Systems*. ACM. 2018, pp. 40–51.

[Par+17] I Parolas et al. "Prediction of vessels' estimated time of arrival (ETA) using machine learning–a port of Rotterdam case study". In: *The 96th Annual Meeting of the Transportation Research Board January*. 2017.

[PP99] James Pitkow and Peter Pirolli. "Mining Longest Repeating Subsequences to Predict World Wide Web Surfing". In: *USENIX Symposium on Internet Technologies and Systems*. 1999, pp. 13–13. URL: `http://dl.acm.org/citation.cfm?id=1251480.1251493`.

[Pop+17] Olga Poppe et al. "Complete Event Trend Detection in High-Rate Event Streams". In: *SIGMOD*. 2017, pp. 109–124.

[QGT16] Miao Qiao, Junhao Gan, and Yufei Tao. "Range Thresholding on Streams". In: *Proceedings of the 2016 International Conference on Management of Data*. SIGMOD '16. San Francisco, California, USA: ACM, 2016, pp. 571–582. URL: `http://doi.acm.org/10.1145/2882903.2915965`.

[REG11] Ella Rabinovich, Opher Etzion, and Avigdor Gal. "Pattern rewriting framework for event processing optimization". In: *Proceedings of the 5th ACM international conference on Distributed event-based system*. ACM. 2011, pp. 101–112.

[Ram+00a] Frank Ramsak et al. "Integrating the UB-Tree into a Database System Kernel". In: *VLDB*. 2000, pp. 263–272.

[Ram+00b] Frank Ramsak et al. "Integrating the UB-Tree into a Database System Kernel". In: *VLDB*. 2000.

[RLR16] Medhabi Ray, Chuan Lei, and Elke A. Rundensteiner. "Scalable Pattern Sharing on Event Streams*". In: *Proceedings of the 2016 International Conference on Management of Data*. SIGMOD '16. San Francisco, California, USA: ACM, 2016, pp. 495–510. URL: `http://doi.acm.org/10.1145/2882903.2882947`.

[Ray+13] Medhabi Ray et al. "High-performance complex event processing using continuous sliding views". In: *Proceedings of the 16th International Conference on Extending Database Technology*. ACM. 2013, pp. 525–536.

[RED12] Suchet Rinsurongkawong, Mongkol Ekpanyapong, and Matthew N Dailey. "Fire detection for early fire alarm based on optical flow video processing". In: *Electrical engineering/electronics, computer, telecommunications and information technology (ecti-con), 2012 9th international conference on*. IEEE. 2012, pp. 1–4.

[Ros+18] Valentin Rosca et al. "Predicting Destinations by Nearest Neighbor Search on Training Vessel Routes". In: *Proceedings of the 12th ACM International Conference on Distributed and Event-based Systems, DEBS 2018, Hamilton, New Zealand, June 25-29, 2018*. 2018, pp. 224–225. URL: `https://doi.org/10.1145/3210284.3220509`.

[Sam05] Hanan Samet. *Foundations of Multidimensional and Metric Data Structures*. Morgan Kaufmann Publishers Inc., 2005.

[SB13] Gereon Schueller and Andreas Behrend. "Stream fusion using reactive programming, LINQ and magic updates". In: *FUSION*. 2013, pp. 1265–1272.

[SMP09] Nicholas Poul Schultz-Møller, Matteo Migliavacca, and Peter Pietzuch. "Distributed complex event processing with query rewriting". In: *Proceedings of the Third ACM International Conference on Distributed Event-Based Systems*. ACM. 2009, p. 4.

[SSS10] Sinan Sen, Nenad Stojanovic, and Ljiljana Stojanovic. "An approach for iterative event pattern recommendation". In: *Proceedings of the Fourth ACM International Conference on Distributed Event-Based Systems*. ACM. 2010, pp. 196–205.

[SS04] Alex Spokoiny and Yuval Shahar. "A knowledge-based time-oriented active database approach for intelligent abstraction, querying and continuous monitoring of clinical data." In: *Medinfo*. 2004, pp. 84–88.

[SÇZ05] Michael Stonebraker, Ugur Çetintemel, and Stan Zdonik. "The 8 requirements of real-time stream processing". In: *ACM Sigmod Record* 34.4 (2005), pp. 42–47.

[Su+14] Xueyuan Su et al. "Changing engines in midstream: A Java stream computational model for big data processing". In: *PVLDB* (2014), pp. 1343–1354.

[Sub+16] Julien Subercaze et al. "Inferray: fast in-memory RDF inference". In: *PVLDB* 9.6 (2016), pp. 468–479.

[Sub+17] Julien Subercaze et al. "Upsortable: programming top-k queries over data streams". In: *Proceedings of the VLDB Endowment* 10.12 (2017), pp. 1873–1876.

[Ter+92] Douglas Terry et al. *Continuous queries over append-only databases.* Vol. 21. 2. ACM, 1992.

[TKA02] William Thies, Michal Karczmarek, and Saman Amarasinghe. "StreamIt: A language for streaming applications". In: *International Conference on Compiler Construction.* 2002, pp. 179–196.

[TS03] Etsuji Tomita and Tomokazu Seki. "An efficient branch-and-bound algorithm for finding a maximum clique". In: *Discrete mathematics and theoretical computer science.* 2003.

[Tur41] Paul Turán. "On an extremal problem in graph theory". In: *Mat. Fiz. Lapok* 48.436-452 (1941), p. 137.

[TGW09] Yulia Turchin, Avigdor Gal, and Segev Wasserkrug. "Tuning complex event processing rules using the prediction-correction paradigm". In: *Proceedings of the Third ACM International Conference on Distributed Event-Based Systems.* ACM. 2009, p. 10.

[Was+12] Segev Wasserkrug et al. "Efficient processing of uncertain events in rule-based systems". In: *IEEE Transactions on Knowledge and Data Engineering* 24.1 (2012), pp. 45–58.

[WDR06] Eugene Wu, Yanlei Diao, and Shariq Rizvi. "High-performance Complex Event Processing over Streams". In: *SIGMOD.* 2006.

[Yan+11] Di Yang et al. "An optimal strategy for monitoring top-k queries in streaming windows". In: *Proceedings of the 14th International Conference on Extending Database Technology.* ACM. 2011, pp. 57–68.

[Yi+03] Ke Yi et al. "Efficient maintenance of materialized top-k views". In: *Data Engineering, 2003. Proceedings. 19th International Conference on.* IEEE. 2003, pp. 189–200.

[ZZN14] Tilmann Zäschke, Christoph Zimmerli, and Moira C. Norrie. "The PH-tree: A Space-efficient Storage Structure and Multi-dimensional Index". In: *Proceedings of the 2014 ACM SIGMOD International Conference on Management of Data.* SIGMOD '14. Snowbird, Utah, USA: ACM, 2014, pp. 397–408. URL: `http://doi.acm.org/10.1145/2588555.2588564`.

[ZDI14a] Haopeng Zhang, Yanlei Diao, and Neil Immerman. "On Complexity and Optimization of Expensive Queries in Complex Event Processing". In: *SIGMOD.* 2014, pp. 217–228.

[ZDI14b] Haopeng Zhang, Yanlei Diao, and Neil Immerman. "On complexity and optimization of expensive queries in complex event processing". In: *Proceedings of the 2014 ACM SIGMOD international conference on Management of data*. ACM. 2014, pp. 217–228.

[Zhu17] Ruiand others Zhu. "SAP: Improving Continuous Top-K Queries over Streaming Data". In: *TKDE* (2017).

[Zhu+17] Rui Zhu et al. "Approximate Continuous Top-k Query over Sliding Window". In: *Journal of Computer Science and Technology* 32.1 (2017), pp. 93–109.

[Zhu+01] Shelley Q Zhuang et al. "Bayeux: An architecture for scalable and fault-tolerant wide-area data dissemination". In: *Proceedings of the 11th international workshop on Network and operating systems support for digital audio and video*. ACM. 2001, pp. 11–20.