Event Stream Analytics

by

Olga Poppe

A Dissertation

Submitted to the Faculty

of the

WORCESTER POLYTECHNIC INSTITUTE

In partial fulfillment of the requirements for the

Degree of Doctor of Philosophy

in

Computer Science

by

January 6, 2018

APPROVED:

Professor Elke A. Rundensteiner
Worcester Polytechnic Institute
Advisor

Professor Dan Dougherty
Worcester Polytechnic Institute
Committee Member

Professor Mohamed Y. Eltabakh
Worcester Polytechnic Institute
Committee Member

Professor David Maier
Portland State University
External Committee Member

Professor Craig Wills
Worcester Polytechnic Institute
Head of Department

Abstract

Advances in hardware, software and communication networks have enabled applications to generate data at unprecedented volume and velocity. An important type of this data are event streams generated from financial transactions, health sensors, web logs, social media, mobile devices, and vehicles. The world is thus poised for a sea-change in time-critical applications from financial fraud detection to healthcare analytics empowered by inferring insights from event streams in real time. Event processing systems continuously evaluate massive query workloads to detect and aggregate event trends of interest. Examples of these trends include check kites in financial fraud detection, irregular heartbeat in healthcare analytics, and vehicle trajectories in traffic control. These trends can be of any length. Worst yet, their number may grow exponentially in the number of events. State-of-the-art systems do not offer practical solutions for trend analytics and thus suffer from long delays and high memory costs.

In this dissertation, we first propose the Complete Event Trend detection (CET) approach. Due to common event sub-sequences in CETs, either the responsiveness is delayed by repeated computations or an exorbitant amount of memory is required to store partial results. To overcome these limitations, we define the CET graph to compactly encode all matched CETs. Based on the graph, we define the spectrum of CET-detection algorithms from time-optimized to space-optimized. We find the middle ground between these two

extremes by partitioning the graph into time-centric graphlets and caching partial CETs per graphlet to enable effective reuse of these intermediate results. We reveal cost monotonicity properties of the search space of graph partitioning plans. Our CET optimizer leverages these properties to prune significant portions of the search to produce a partitioning plan with minimal latency yet within the available memory.

Second, we propose the Graph-based Real-time Event Trend Aggregation (GRETA) approach that dynamically computes event trend aggregation without first constructing these trends. We define the GRETA graph to compactly encode all trends. Our GRETA runtime incrementally maintains the graph, while dynamically propagating aggregates along its edges. Based on the graph, the final aggregate is incrementally updated and instantaneously returned at the end of each query window. Our GRETA runtime represents a win-win solution, reducing both the time complexity from exponential to quadratic and the space complexity from exponential to linear in the number of events compared to state-of-the-art techniques. Beyond computing aggregates at the fine granularity of each matched event, we design an optimized strategy that maintains aggregates at the coarse granularity levels – per event type or even per pattern. This strategy minimizes the number of aggregates and is shown to further reduce time and space complexity for event trend aggregation.

Third, our Shared Online Event Sequence Aggregation (SHARON) approach shares intermediate aggregates among multiple queries while avoiding the expensive construction of event sequences. Our SHARON optimizer faces two challenges. One, a sharing decision is not always beneficial. Two, a sharing decision may prevent other sharing opportunities. To guide our SHARON optimizer, we compactly encode sharing candidates, their benefits, and conflicts among candidates into the SHARON graph. Based on the graph, we map our problem of finding an optimal sharing plan to the Maximum Weight Independent Set (MWIS) problem. We then use the guaranteed weight of a greedy algorithm for the MWIS problem to prune the search of our sharing plan finder without sacrificing its optimality.

In several comprehensive experimental studies, namely, one for each part of this dissertation, we demonstrate the superiority of the proposed strategies over the state-of-the-art techniques with respect to latency, throughput, and memory costs.

To my parents

Acknowledgements

I sincerely thank my adviser Prof. Elke A. Rundensteiner for multiple path-breaking discussions, prompt feedback, patience, and sleepless nights. She always believed in me and encouraged me to complete my Ph.D. thesis. She played the key role in every step of my Ph.D. studies from my application for graduate program at WPI, through every talk, publication, internship, or milestone. She was both supportive and inspiring while supervising my research from a vague idea to a polished publication at a top-level venue. Without her guidance, I would not be where I am now in my professional and personal development. Her excellence in tackling challenging research problems will always be my inspiration to continue my research in the future.

I would like to thank my committee members Prof. David Maier, Prof. Dan Dougherty, and Prof. Mohamed Y. Eltabakh for their valuable feedback during all milestones of my Ph.D. research. I am very grateful to Prof. David Maier for his attention to detail that improved the quality of several top-level conference publications. I would like to thank Prof. Dan Dougherty for his guidance during my research qualification. His feedback helped me to simplify the CAESAR model to focus on the most interesting idea of context-aware event stream analytics. I am grateful to all members of DSRG group – especially, Prof. Mohamed Y. Eltabakh, Dr. Lei Cao, and Prof. Xiangnan Kong – for their critical comments during my talks. I thank Prof. François Bry for his supervision at an early stage of my Ph.D. studies.

I sincerely thank Dr. Chuan Lei for his support and multiple discussions on a daily basis. His critical comments significantly improved every publication we have produced together. I enjoyed the internship at NEC Labs in summer 2016 under his supervision. It was a pleasure to work with him and I am looking forward to our future joint publications.

I was fortunate to collaborate with multiple bright students including Sandro Giessl, Soni Dang, Allison Rozet, Salah Ahmed, Goutham Deva, Nicholas Fajardo, James Owens, Thomas Schweich, MaryAnn VanValkenburg, Sarun Paisarnsrisomsuk, and Pitchaya Wiratchotisatian. I appreciate their hard work to get both theory and implementation completed on time.

I am endlessly grateful to my family for being always there for me. This dissertation is dedicated to my parents. I would like to thank Gerhard Poppe for always believing in me and for his support at the very beginning of my sudies of computer science. I sincerely thank my friends and colleagues Harald Zauner, Simon Brodt, Dr. Dmitry Sinyukov, Dr. Hardik Kabaria, Dr. Norbert Eisinger, Ingeborg von Troshke, Prof. Hans-Jürgen Ohlbach, Dr. Jakub Kotowski, Dr. Tim Furche, and Dr. Benedikt Linse, for their support during tough days of my Ph.D. studies. I was fortunate to be a member of the Russian-Speaking Students and Scholars Association and the Alpine Ski Team at WPI. Thanks to these clubs I felt well-rested before each new accomplishment.

I thank Mark Taylor, Michael Voorhis, and John Leveillee for helping me to become familiar with cluster usage. I am also grateful to Mary Racicot and Refie Cane for resolving bureaucratic issues. Last but not least, I am thankful for the financial support I have received from my adviser Prof. Elke A.

Rundensteiner and Computer Science department at WPI. I was supported as a teaching assistant during the first year in graduate school. Afterwards, my research has been funded by the National Science Foundation grants, namely, IIS 1018443, IIS 143620, IIS 1560229, and CRI 1305258.

My Publications

Publications Contributing to this Dissertation

Part I: Event Trend Detection

Part I of this dissertation is devoted to the efficient detection of event trends.

1. Olga Poppe, Chuan Lei, Salah Ahmed, and Elke A. Rundensteiner, *Complete Event Trend Detection in High-rate Event Streams*, **International Conference on Management of Data (SIGMOD)**, 2017, pages 109-124.

   Relationship to this dissertation: Part I is based on the publication above in which we propose the Complete Event Trend (CET) detection approach. First, we compactly capture all CETs as a graph. Based on the graph, we propose the spectrum of CET detection algorithms that range from CPU-optimized to memory-optimized. Lastly, we find the middle ground between CPU costs and memory usage of CET detection in high-rate event streams.

Part II: Event Trend Aggregation

Part II of this dissertation defines real-time, light-weight aggregation of event trends.

2. Olga Poppe, Chuan Lei, Elke A. Rundensteiner, and David Maier, *GRETA: Graph-based Real-time Event Trend Aggregation*, **International Conference on Very**

Large Data Bases (VLDB), 2018, pages 80-92.

Relationship to this dissertation: Chapters 10–13 are based on the publication above in which we present the GRETA approach. It avoids the event trend construction step for the purpose of event trend aggregation. GRETA encodes all trends as a graph and propagates the aggregates from previous events to more recent events in a dynamic programming fashion. Thus, GRETA reduces the time complexity of event trend aggregation from exponential to quadratic in the number of events in the worst case compared to the state-of-the-art approaches.

3. Olga Poppe, Chuan Lei, Elke A. Rundensteiner, and David Maier, *Event Trend Aggregation Under Rich Event Matching Semantics*, 2018, **in submission**.

   Relationship to this dissertation: Chapters 14–20 are based on the publication above in which we propose the COGRA approach. It supports rich event matching semantics and implements incremental event trend aggregation at multiple granularity levels that range from fine (per matched event), to medium (per event type), to coarse (per pattern). Our COGRA approach maintains trend aggregates at the coarsest possible granularity level. Thus, it minimizes the number of maintained aggregates – further reducing time and space complexity of event trend aggregation.

Part III: Shared Event Sequence Aggregation

Part III of this dissertation explores sharing opportunities among event sequence aggregation queries to reduce the workload latency.

4. Olga Poppe, Allison Rozet, Chuan Lei, Elke A. Rundensteiner, and David Maier, *Sharon: Shared Online Event Sequence Aggregation*, 2018, **in submission**.

Relationship to dissertation: Part III is based on the publication above in which we propose the SHARON approach. It shares intermediate aggregates among multiple queries while avoiding the expensive construction of event sequences. To this end, it encodes sharing candidates, their benefits, and conflicts among candidates into the SHARON graph. Based on the graph, we map our problem of finding an optimal sharing plan to the Maximum Weight Independent Set (MWIS) problem. We then use the guaranteed weight of a greedy algorithm for MWIS to prune the search of our sharing plan finder without sacrificing its optimality.

Other Publications

During my Ph.D. research, I contributed to the following publications in event stream analytics.

5. Olga Poppe, Chuan Lei, Elke A. Rundensteiner, Dan Dougherty, Goutham Deva, Nicholas Fajardo, James Owens, Thomas Schweich, MaryAnn VanValkenburg, Sarun Paisarnsrisomsuk, Pitchaya Wiratchotisatian, George Gettel, Robert Hollinger, Devin Roberts, and Daniel Tocco, *CAESAR: Context-Aware Event Stream Analytics for Urban Transportation Services*, Demonstration, **International Conference on Extending Data Base Technology (EDBT)**, 2017, pages 590-593.

6. Olga Poppe, Chuan Lei, Elke A. Rundensteiner, and Dan Dougherty, *Context-aware Event Stream Analytics*, **International Conference on Extending Data Base Technology (EDBT)**, 2016, pages 413-424.

7. Salah Ahmed, Olga Poppe, and Elke A. Rundensteiner, *Event Sequence Detection over Interval-Based Event Streams*, **International Conference on Advances in Big Data Analytics (ABDA)**, 2016, pages 17-23.

8. Elke A. Rundensteiner, Olga Poppe, Chuan Lei, Medhabi Ray, Lei Cao, Yingmei Qi, Mo Liu, and Di Wang. *Exploiting Sharing Opportunities for Real-time Complex Event Analytics*, **IEEE Data Engineering Bulletin**, 2015, pages 82-93.

9. Olga Poppe, Sandro Giessl, Elke A. Rundensteiner, and Francois Bry. *The HIT Model: Workflow-aware Event Stream Monitoring*, **Transactions on Large-Scale Data- and Knowledge-Centered Systems**, 2013, pages 26-50.

10. Michael Eckert, Francois Bry, Simon Brodt, Olga Poppe, and Steffen Hausmann. *A CEP Babelfish: Languages for Complex Event Processing and Querying Surveyed*, **Reasoning in Event-Based Distributed Systems**, 2011, pages 47-70.

11. Michael Eckert, Francois Bry, Simon Brodt, Olga Poppe, and Steffen Hausmann. *Two Semantics for CEP, no Double Talk: Complex Event Relational Algebra and its Application to XChangeEQ*, **Reasoning in Event-Based Distributed Systems**, 2011, pages 71-97.

Contents

List of Figures

List of Tables

1

Introduction

Complex Event Processing (CEP) is a technology for supporting streaming applications such as healthcare analytics, financial fraud detection, and urban transportation services. CEP systems continuously evaluate event queries against high-rate streams of primitive events to detect higher-level event trends. In contrast to traditional event sequences of *fixed* length [4], event trends have an *arbitrary* length [5, 6]. They are expressed by Kleene patterns. Various event matching semantics were defined in the CEP literature [3, 7, 8] to determine the contiguity of event trends. For example, *phases of contiguously increasing heartbeat* are detected in healthcare analytics, while *non-contiguous check kites* are of interest for financial fraud detection. Aggregation functions may be applied to these event trends to provide summarized insights about them. For example, the *count of trips on a route* serves as a measure of route popularity. CEP applications must react to critical changes of these aggregates in real time.

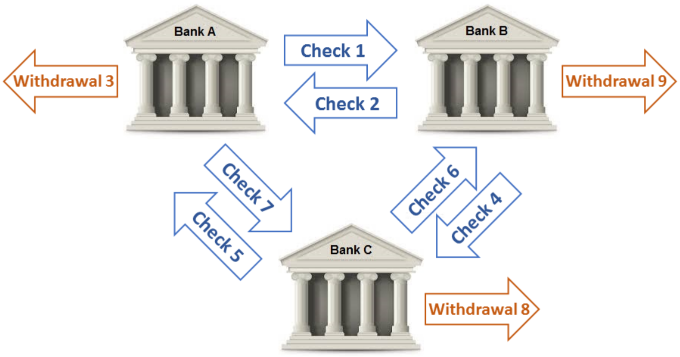


Figure 1.1: Circular check kiting

1.1 Motivating Examples

We now describe three use case scenarios of time-critical event trend analytics that require alternative event matching semantics.

• ***Financial fraud detection***. Circular check kiting is an example of event-trend detection for financial fraud. In a simple case, it involves writing a check for a value greater than the account balance from an account in Bank A, then writing a check from another account in Bank B, also with insufficient funds, with the second check serving to cover the non-existent funds from the first account. Fraudsters take advantage of the float and withdraw funds from the account before the banks can detect the scheme (Figure 1.1).

Complex versions of this scheme have occurred involving multiple fraudsters posing as large businesses, thereby masking their activity as normal business transactions. In this way they coax banks to waive the limit of available funds [9]. To implement this scheme, fraudsters transfer millions among banks using complex webs of worthless checks. As just one example, in 2014, 12 people were charged in a large-scale "bustout" scheme, costing banks over $15 million [10].

Query q detects a chain (or circle) of any length formed by uncovered check deposits during a time window of 1 day that slides every 10 minutes. The pattern of the query is the Kleene closure on check deposit events, denoted *Check C*+. The predicates require

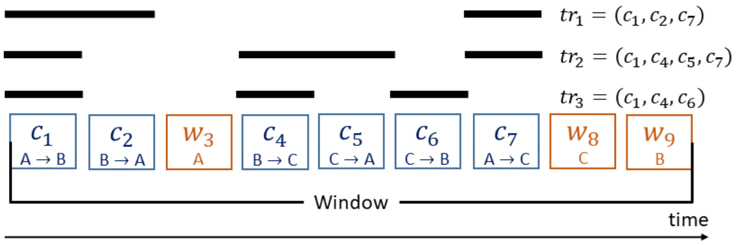


Figure 1.2: Three check kite trends detected by query q

the checks in a chain to be uncovered. The destination of a check C must be the same as the source of the next check NEXT(C) to form a chain.

q : PATTERN Check C+

WHERE C.type = "uncovered" AND C.destination = NEXT(C).source

SEMANTICS skip-till-any-match

WITHIN 1 day SLIDE 10 minutes

Since arbitrary many fraudsters, financial transactions and banks worldwide can be involved in this scheme, detection of circular check kites is a computationally expensive problem. To prevent cash withdrawal from an account that is involved in at least one check kiting scheme, the query continuously analyzes high-rate event streams with thousands of financial transactions per second and detects *all complete* check kiting trends in real time.

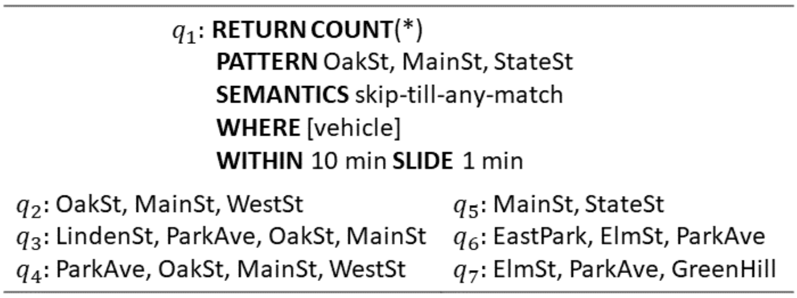
In Figure 1.2, $(c_1 : A \rightarrow B)$ denotes an uncovered check deposit event from Bank A into Bank B at time 1 and $(w_3 : A)$ denotes a cash withdrawal event from Bank A at time 3. Three check kiting trends are detected in this example. They are shown as black lines above the event stream: (c_1, c_2, c_7), (c_1, c_4, c_5, c_7) and (c_1, c_4, c_6). Note that check c_2 is part of the first trend but is skipped to detect the second and third event trends. Such flexible way of finding all matches is called skip-till-any-match event matching semantics [7].

• ***Healthcare analytics***. Cardiac arrhythmia is a serious heart disease in which the heartbeat is too fast, too slow, or irregular. It can lead to life-threatening complications causing about 325K sudden cardiac deaths in US per year [11]. Thus, an abnormal heartbeat must be promptly detected to enable immediate lifesaving measures.

```
q′ : RETURN patient, MIN(M.rate), MAX(M.rate)
     PATTERN Measurement M+
     SEMANTICS contiguous
     WHERE [patient] AND M.rate < NEXT(M).rate AND M.activity = passive
     GROUP-BY patient
     WITHIN 10 minutes SLIDE 30 seconds
```

Query q' detects extreme differences in heartbeat during passive physical activities (e.g., reading, watching TV). The query consumes a stream of heart rate measurements of intensive care patients. Each event carries a time stamp in seconds, a patient identifier, an activity identifier, and a heart rate. For each patient, q' detects phases of a contiguously increasing heart rate despite passive physical activity during a time window of 10 minutes that slides every 30 seconds. The query computes the *minimal* and the *maximal* heart rate measurements during such phases. No measurements may be skipped in between matched events per patient, as expressed by the *contiguous* semantics.

• ***Urban transportation services***. With the growing popularity of ridesharing services such as Uber and Lyft, their systems face multiple challenges including real-time analysis of vehicle trajectories, geospatial prediction, and alerting. These systems evaluate massive workloads of event queries against high-rate streams of drivers' position reports and riders' requests to infer the current supply and demand situation on each route. They incorporate traffic conditions to compute the best route for each trip. They then instantaneously react to critical changes to prevent time waste, reduce costs and pollution, and increase riders' satisfaction and drivers' profit. With thousands of drivers and over 150 requests per minute in New York City [12, 13, 14], real-time traffic analytics and ride management is a challenging task.



```
          q1: RETURN COUNT(*)
              PATTERN OakSt, MainSt, StateSt
              SEMANTICS skip-till-any-match
              WHERE [vehicle]
              WITHIN 10 min SLIDE 1 min
q2: OakSt, MainSt, WestSt                 q5: MainSt, StateSt
q3: LindenSt, ParkAve, OakSt, MainSt      q6: EastPark, ElmSt, ParkAve
q4: ParkAve, OakSt, MainSt, WestSt        q7: ElmSt, ParkAve, GreenHill
```

Figure 1.3: Traffic monitoring workload Q

Queries q_1–q_7 in Figure 1.3 compute the count of trips on a route as a measure of route popularity. They consume a stream of vehicle position reports. Each report carries such attribute values as a time stamp in seconds, a car identifier and its position. Here, event type corresponds to a vehicle position. For example, a vehicle on Main Street sends a position report of type *MainSt*. Each trip corresponds to a sequence of position reports from the same vehicle (as required by the predicate *[vehicle]*) during a 10-minute-long time window that slides every minute. The predicates and window parameters of q_2–q_7 are identical to q_1 and thus are not shown for compactness. The sub-pattern *(OakSt,MainSt)* appears in queries q_1–q_4. Sharing the aggregation of common patterns among multiple similar queries is vital to speed up the system responsiveness.

1.2 State-of-the-Art Approaches

We first describe the relationship of this dissertation to the broader class of data management approaches. We then summarize the shortcomings of existing streaming approaches. Chapters 9, 20, and 26 discuss related work for each part of this dissertation in more detail.

1.2.1 Broader Context

The optimization techniques of Kleene closure and event sequence queries proposed in this dissertation loosely relate to static databases and recursive queries.

- **Static Databases**. Traditional SQL queries [15, 16, 17] do not support streaming operators such as event sequence and Kleene closure that treat the order of events by their time stamps as first-class citizen. While static sequence data bases extend traditional SQL queries by order-aware join operators [18, 19], these approaches work with data that is statically stored and indexed prior to processing. Thus, they do not tackle the challenges of dynamically streaming data such as event expiration and real-time reactions.

- **Recursive Queries**. Kleene closure can be considered as a special class of recursive queries [20, 21, 22, 23, 24, 25, 26, 27]. However, these approaches have a different focus. Namely, extending expressive power for recursive queries [26] or ensuring correctness of such queries [20, 23]. These approaches do not support multiple event matching semantics [3, 7, 8] that are required to express diverse streaming application scenarios (Section 1.1).

1.2.2 Event Trend Detection

While existing event processing approaches recognize the importance of Kleene closure computation over event streams [3, 7, 8, 28, 29], some of them, namely, Cayuga [28] and ZStream [29] do not support the skip-till-any-match semantics required to express the use cases above. While SASE [3] supports Kleene closure computation under the skip-till-any-match semantics, it stores single events and forms matches at the end of each window. Since an event sub-sequence can be part of multiple matches, SASE re-computes a common event sub-sequence for each match that contains it. This approach suffers from repeated computations. For example, our experiments in Chapter 8 demonstrate that

	Event sequences	**Event trends**
Two-step	Flink [30], Esper [31], Oracle Stream Analytics [32]	SASE [3], Cayuga [28], ZStream [29]
Online	A-Seq [2]	GRETA, COGRA

Table 1.1: Event sequence versus event trend aggregation approaches

the CPU processing time of SASE is 38 minutes when the event rate is 50k per second and the query window is 30 minutes. Such a long processing delay is unacceptable for time-critical applications that require high responsiveness, such as within a minute. In summary, the existing approaches do not fully address the challenges of real-time event trend detection over high-rate event streams.

1.2.3 Event Trend Aggregation

State-of-the-art approaches to event aggregation can be divided into the following groups (Table 1.1).

• ***CEP approaches*** such as SASE [3], Cayuga [28], and ZStream [29] support Kleene closure. However, Cayuga and ZStream do not consider event matching semantics defined in the literature [3, 7, 8]. While their languages include aggregation, they do not provide any optimization techniques to compute aggregation on top of Kleene patterns. They utilize the two-step approach that constructs all trends prior to their aggregation. This approach suffers from long delays or even fails to terminate due to the exponential cost of event trend construction (Chapters 13 and 19). A-Seq [2] computes aggregation of *fixed-length event sequences* without constructing these sequences. However, A-Seq supports neither Kleene closure nor event matching semantics. Thus, it does not tackle the exponential complexity of event trends – which now is the focus of our work.

• ***Streaming systems***. Industrial streaming systems such as Flink [30], Esper [31], and Oracle Stream Analytics [32] only support fixed-length event sequences aka sequential

	Non-Shared	**Shared**
Two-step	Flink [30], SASE [3], Cayuga [28], ZStream [29]	SPASS [38], ECube [4]
Online	A-Seq [2]	SHARON

Table 1.2: Shared versus non-shared event sequence aggregation approaches

join. They do not support Kleene closure queries. They construct all sequences prior to their aggregation and thus follow the prohibitively expensive two-step approach.

Streaming approaches [33, 34, 35, 36, 37] evaluate Select-Project-Join queries with windows, i.e., their execution paradigm is set-based. They support neither event sequence nor Kleene closure. These approaches require the construction of join results prior to their aggregation. Thus, they define incremental aggregation of *single raw events*.

1.2.4 Shared Event Sequence Aggregation

State-of-the-art approaches to event sequence aggregation can be divided into the following groups (Table 1.2):

- ***Non-shared two-step approaches***, including Flink [30], SASE [3], Cayuga [28], and ZStream [29], evaluate each query independently from other queries in the workload. Furthermore, these approaches do not offer optimization strategies specific for event sequence aggregation queries. Without special optimization techniques, these approaches first construct event sequences and then aggregate them. Since the number of event sequences is polynomial in the number of events [2, 3], event sequence construction is an expensive step. Our experiments in Chapter 25 confirm that such a *non-shared, two-step* approach implemented by the popular open-source streaming system Flink [30] does not terminate, even for low-rate streams of a few hundred events per second.

- ***Shared two-step approaches*** such as SPASS [38] and ECube [4] focus on *shared* event sequence construction, not on sequence aggregation. If these approaches are applied to aggregate event sequences, they would construct all sequences prior to their aggregation. This event sequence construction step degrades system performance. Our experiments in Chapter 25 confirm that such a *shared, two-step* approach implemented by SPASS [3] requires 41 minutes per query window, even for low-rate streams of a few hundred events per second. Such long delays are not acceptable for time-critical applications that require a system response within a few seconds [39].

- ***Non-shared online approaches*** such as A-Seq [2] compute event sequence aggregation *online*, i.e., without constructing the sequences. A-Seq incrementally maintains a set of aggregates for each pattern and instantaneously discards each event once it has updated the aggregates. A-Seq does not tackle the sharing optimization problem to determine which queries should share the aggregation of which sub-patterns such that the latency of a workload is minimized – which is now the focus of our work. Without an optimizer, A-Seq does not share computations among multiple queries.

1.3 Research Challenges

In this dissertation, we focus on the following open problems.

Event Trend Detection.

- ***Exponentially many trends of unbounded length***. Not only is each event trend of statically unknown and potentially unbounded length, but also the number of trends can be exponential in the number of relevant events in the worst case (Chapter 3). Without a clever design, this complexity may jeopardize real-time trend detection in high-rate event streams.

- ***CPU versus memory trade-off of trend detection***. Due to the occurrence of many common event sub-sequences in trends, either repeated computations are invoked or an exorbitant amount of memory is required to store partial trends during trend detection.

Consequently, the system may either fail to deliver the results with low latency or run out of memory due to high-rate event streams.

• ***Exponential stream-partitioning problem***. The divide and conquer principle is a common solution to the problem above of trading between CPU and memory. Namely, we could partition the stream, cache results per partition and reuse them for final result construction. However, with the search space being exponential [40], an effective lightweight stream-partitioning algorithm must be developed to guarantee prompt system responses.

Event Trend Aggregation.

• ***Real-time event trend aggregation***. Kleene closure matches an exponential number of arbitrarily long event trends in the number of events in the worst case [3]. Thus, any practical solution must aim to aggregate event trends without first constructing them to enable real-time in-memory execution. At the same time, correctness must be guaranteed. That is, the same aggregates must be returned as by the two-step approach.

• ***Nested Kleene patterns***. Kleene closure detects event trends of arbitrary length. Worse yet, Kleene closure, event sequence, and negation may be arbitrarily nested in a pattern, introducing complex interdependencies between events in an event trend. Incremental aggregation of such arbitrarily long and complex event trends is an open problem.

• ***Rich event matching semantics*** were defined in the CEP literature [3, 7, 8] to enable expressive event queries required for different applications. These semantics range from the most restrictive contiguous semantics (query q' above) to the most flexible skip-till-any-match semantics (query q). Their execution strategies differ significantly, making the seamless support of online event trend aggregation on top of these diverse semantics a challenging task.

Shared Event Sequence Aggregation.

• ***Online yet shared event sequence aggregation***. Unfortunately, these optimization techniques (that allow computing event sequence aggregation online and at the same time share the intermediate results) cannot be simply combined with each other because they impose contradictory constraints on the underlying execution strategy. For example, if query q_4 in Figure 1.3 *shares* the aggregation results of patterns $p = (ParkAve, OakSt)$ and $p' = (MainSt, WestSt)$ with other queries, the aggregates for p and p' must be combined to form the final results for q_4. To ensure correctness, this result combination must be aware of the temporal order between sequences matched by p and p' and their expiration. To be able to analyze these temporal relationships, event sequences must be constructed. This requirement contradicts the key idea of the *online* approaches that avoid the expensive event sequence construction step.

• ***Benefit of sharing***. Sharing the aggregation computation for a pattern p by a set of queries Q_p containing p is not always beneficial, since this sharing may introduce considerable CPU overhead of combining shared intermediate aggregates to form the final results for each query in Q_p. Thus, an accurate sharing benefit model is required to assess the quality of a sharing plan.

• ***Intractable sharing plan search space***. The search space for a high-quality sharing plan is exponential in the number of sharing candidates. Since the event rate may fluctuate, the benefit of sharing a pattern may change over time. To achieve a high sharing benefit, the sharing plan may have to be dynamically adjusted. Hence, an effective yet efficient optimization algorithm for sharing plan selection is required.

1.4 Proposed Solutions

To tackle the open challenges above, we extend state-of-the-art techniques with event trend analytics capabilities. In particular, we focus on event query optimizer and runtime stream analytics (Figure 1.4). Our proposed techniques are compatible with the existing

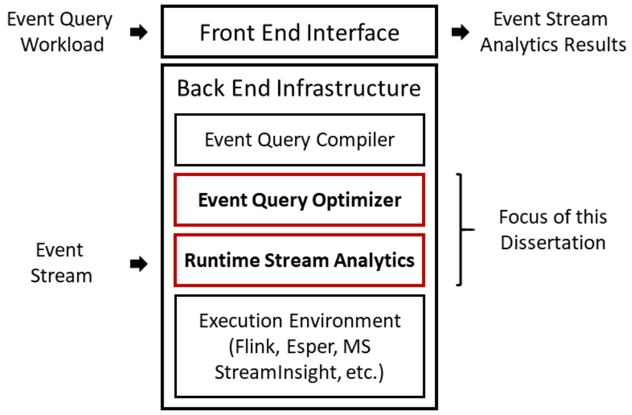


Figure 1.4: Event stream analytics framework

streaming engines such as Flink [30], Esper [31], Microsoft SreamInsight [41], Oracle Stream Analytics [32]. Our contributions include:

Event Trend Detection. We focus on detecting Complete Event Trends (CETs), i.e., trends that are not part of other trends. Given an event query with a Kleene pattern, our CET processing paradigm first extracts events matched by the query from a stream. It then encodes their CET relationships in a compact data structure, called the CET graph. Based on the graph, we propose a family of CET detection algorithms ranging from the memory-optimized M-CET to the CPU-time-optimized T-CET solution. M-CET avoids excessive storage of intermediate results and thus invokes repeated computations. T-CET accelerates CET detection by incrementally maintaining intermediate results but unfortunately requires an exorbitant amount of memory.

To trade off between CPU and memory costs, we develop the Hybrid CET detection algorithm H-CET which is a middle ground between these two extremes. Namely, we partition the CET graph into smaller graphlets. Based on the partitioned graph, H-CET caches partial CETs per graphlet using T-CET, and then stitches these partial CETs together to form the final CET results using M-CET. Partitioning faces the trade off that fine-grained partitioning plans reduce the memory consumption (since CETs are stored per graphlet) while increasing the execution time due to the overhead of partial result combination. On the other hand, coarse-grained partitioning plans have high memory costs while the overhead of partial result combination is low. We thus design a cost-driven CET optimizer that finds an optimal partitioning plan with minimal CPU execution costs yet within the available memory limit. Our key innovations are the following:

1) We define the problem of real-time CET detection over high-rate event streams under memory constraints. We prove that the number of CETs is exponential in the number of relevant events in the worst case.

2) We introduce a compact data structure, called a CET graph, to encode relevant events and their CET relationships. The spectrum of CET detection algorithms ranging from the CPU-time-optimized algorithm T-CET to the memory-optimized algorithm M-CET is introduced.

3) To trade-off between CPU and memory costs, we develop the Hybrid CET-detection algorithm H-CET. We first partition the CET graph into time-centric graphlets. Then, H-CET computes CETs per graphlet using T-CET and reuses these partial results to detect all CETs using M-CET.

4) We establish a cost model for CET detection. Our analysis reveals cost monotonicity properties of the search space of candidate partitioning plans. We then design a CET optimizer that leverages these properties to effectively prune sub-optimal plans. Our optimizer is guaranteed to produce a graph-partitioning plan with minimal execution time within a given memory bound.

5) We conduct an extensive performance evaluation of our CET approach using both synthetic and real data sets [42, 43]. Our CET solution achieves up to a 42–fold speed-up compared to the state-of-the-art strategies including Flink [30] and SASE [3].

Event Trend Aggregation. Given an event trend aggregation query q and a stream I, our Graph-based Real-time Event Trend Aggregation (GRETA) approach compactly encodes all event trends matched by the query q in the stream I into a GRETA graph. During graph construction, aggregates are propagated from previous events to newly arrived events along the edges of the graph following the dynamic programming principle. This propagation assures incremental aggregation computation without first constructing the trends. The final aggregate is also computed incrementally, such that it can be instantaneously returned at the end of each window of q.

Furthermore, we propose to optimize GRETA by maintaining Coarse-Granular aggregates, called the COGRA approach. COGRA is the first technique that defines online event trend aggregation under rich event matching semantics at multiple granularity levels. Depending on the event matching semantics and other query features, COGRA adaptively selects the *coarsest possible granularity level* at which it incrementally computes event trend aggregation. These granularity levels range from fine (per matched event), to medium (per event type), to coarse (per pattern). Thus, COGRA minimizes the number of aggregates and discards all events once they have updated the aggregates. Our COGRA approach represents a win-win solution that reduces both time and space complexity of trend aggregation compared to state-of-the-art approaches. Our key innovations include:

1) We translate a nested Kleene pattern P into a GRETA template. Based on this template, we construct a GRETA graph that compactly captures all trends matched by pattern P in the stream. During graph construction, the aggregates are dynamically propagated along the edges of the graph. We prove the correctness of the GRETA graph and the graph-based aggregation computation.

2) To handle nested patterns with negative sub-patterns, we split the pattern into positive and negative sub-patterns. We maintain a separate GRETA graph for each resulting sub-pattern and invalidate certain events if a match of a negative sub-pattern is found.

3) To avoid sub-graph replication between overlapping sliding windows, we share one GRETA graph between all windows. Each event that falls into k windows maintains k aggregates. A final aggregate is computed per window.

4) To ensure low-latency lightweight query processing, we design the GRETA runtime data structure to support dynamic insertion of newly arriving events, batch deletion of expired events, incremental propagation of aggregates, and efficient evaluation of expressive predicates.

5) We define the problem of real-time event trend aggregation under rich event matching semantics. Based on these event matching semantics and other query features, we identify the coarsest possible granularity level at which trend aggregates are maintained.

6) For each granularity level, we propose efficient data structures and processing strategies to compute event trend aggregation. We also prove the correctness of these strategies.

7) The strategies are shown to reduce both time and space complexity compared to state-of-the-art approaches.

8) Our experiments using both synthetic and real data sets [42, 43] demonstrate that our techniques achieve up to four orders of magnitude speed-up and use up to eight orders of magnitude less memory compared to the state-of-the-art strategies including Flink [30], SASE [3], and A-Seq [2] (Table 1.1).

Shared Event Sequence Aggregation. We propose the following Shared Online Event Sequence Aggregation (SHARON) optimization techniques. Since sharing a pattern p by a set of queries Q_p is not always beneficial, we develop a *sharing benefit model* to assess the quality of a sharing candidate (p, Q_p). The model compares the gain of sharing p among queries Q_p to the overhead of combining shared aggregates of p to form the final results for each query in Q_p. Non-beneficial candidates are pruned. Since a decision to share a pattern may prevent the sharing of another pattern by the same query, we define the notion of *sharing conflicts* among sharing candidates. We compactly encode sharing candidates as vertices and conflicts among these candidates as edges of the SHARON *graph* (Figure 5.1). Each vertex is assigned a weight that corresponds to the benefit of sharing the respective candidate.

Based on the graph, we map our Multi-query Event Sequence Aggregation problem to the Maximum Weight Independent Set (MWIS) problem. We then utilize the guaranteed minimal weight of the approximate algorithm GWMIN [44] for the MWIS problem to prune conflict-ridden candidates. Since conflict-free candidates always belong to an optimal sharing plan, they can also be excluded from the search early on. Based on the reduced graph, our sharing plan finder further prunes sharing plans with conflicts and returns an optimal plan (i.e., plan with minimal estimated latency) to guide our executor at runtime. In summary, SHARON seamlessly combines two optimization strategies into one integrated solution. Namely, it *shares* sequence aggregation among multiple queries, while computing sequence aggregation *online*. Our key innovations are:

1) We design the sharing benefit model to assess the quality of a sharing candidate. Non-beneficial candidates are pruned.

2) We identify sharing conflicts among candidates and encode candidates, their benefits, and conflicts among them into the SHARON *graph*.

3) We map our Multi-query Event Sequence Aggregation problem to the Maximum Weight Independent Set (MWIS) problem and utilize the guaranteed weight of the approximate algorithm for MWIS to prune conflict-ridden candidates.

4) Based on the reduced SHARON graph, we introduce the sharing plan finder that prunes sharing plans with conflicts and returns an optimal sharing plan.

5) Our performance study using real data sets [12, 39] demonstrates that sharing plans produced by the SHARON optimizer achieve up to 18-fold speed-up and use up to two orders of magnitude less memory compared to the state-of-the-art approaches including Flink [30], SPASS [38], and A-Seq [2] (Table 1.2).

1.5 Impact of This Dissertation

Our event trend analytics techniques derive valuable summarized insights about high-rate event streams in real-time. They support expressive event queries and thus are applicable to a wide range of use cases from financial fraud detection to algorithmic trading.

While the state-of-the-art industrial systems such as Microsoft Stream-Insight [41], Flink [30], Esper [31], and Oracle Stream Analytics [32] do not yet explicitly support Kleene closure computation over event streams, first steps in this direction have already been made in Flink [45] and Oracle Stream Analytics [46, 47]. Our optimization techniques can be plugged into these systems to enable scalable event trend analytics. These systems can greatly benefit from these ideas with respect to expressive power and performance optimization.

1.6 Dissertation Outline

This dissertation is organized as follows. We propose event trend detection and aggregation techniques in Parts I and II respectively. In Part III, we describe shared online event sequence aggregation methodology. Part IV concludes this dissertation and proposes future research directions.

2

Data and Query Model

In this chapter, we first define a pattern (Section 2.1). We then statically analyze a pattern to guide runtime execution (Sections 2.2 and 2.3). We define the event matching semantics and event query (Sections 2.4 and 2.5). Lastly, we summarize the assumptions of this dissertation (Section 2.6).

2.1 Pattern

Time. Time is represented by a linearly ordered *set of time points* $(\mathbb{T}, \leq)$, where $\mathbb{T} \subseteq \mathbb{Q}^+$ (the non-negative rational numbers).

Event. An *event* is a message indicating that something of interest happened in the real world. An event e has a *time stamp* $e.time \in \mathbb{T}$ assigned by the event source. An event e belongs to a particular *event type* E, denoted $e.type = E$ and described by a *schema* which specifies the set of *event attributes* and the domains of their values.

Example 2.1 *In the check kiting example, a check deposit event carries a status (covered or not), a source bank and a destination bank. Other attributes (such as account owner, balance, amount) are ignored here for simplicity.*

Event Stream. An *event stream I* is a sequence of events arriving in-order by their time stamps. Multiple events may have the same time stamp. Events are sent by event producers (e.g., ATM machines). An event consumer (e.g., financial fraud detection system) continuously monitors the stream with *event queries*. We borrow the query syntax and semantics from SASE [3, 7, 8]. Our example queries in Section 1.1 are expressed using this syntax.

Definition 2.1 (Raw Pattern) *A **raw pattern** is recursively defined as follows:*

- *An event type E is a pattern.*
- *A Kleene plus operator $P+$ applied to a pattern P is a pattern.*
- *An event sequence operator $\mathsf{SEQ}(P_1, P_2)$ applied to patterns P_1 and P_2 is a pattern.*
- *A negation operator $\mathsf{NOT}\ P$ applied to a pattern P is a pattern.*

*A **Kleene pattern** is a pattern with at least one Kleene plus operator. A pattern without a Kleene plus operator is called an **event sequence pattern**. A pattern is **positive** if it contains no negation. If an operator in a pattern is applied to the result of another operator, the pattern is **nested**. Otherwise, it is **flat**. The **length** of pattern is the number of event types and operators in it.*

Pattern. While Definition 2.1 allows arbitrarily nested patterns, we assume that a negation operator always appears within an event sequence operator and is applied either to an event sequence operator or an event type. In particular, negation may not be the outer most operator in a pattern. If a negative sub-pattern is preceded or followed by a pattern P' (as described in more details in Section 2.3), P' must be positive.

2.2 Template for a Positive Pattern

We first translate a positive pattern P into a Finite State Automaton that is then used as a template during event analytics at runtime. For example, the pattern $P = (\mathsf{SEQ}(A+, B))+$

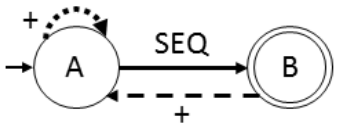


Figure 2.1: Template for the pattern *P=(SEQ(A+,B))+*

is translated into the template in Figure 2.1.

States correspond to event types in P. The initial state is labeled by the ***start type*** in P, denoted $start(P)$. The final state has label $end(P)$, i.e., the ***end type*** in P. All other states are labeled by ***middle types*** $mid(P)$. In Figure 2.1, $start(P) = A$, $end(P) = B$, and $mid(P) = \emptyset$. Since an event type may appear in a pattern at most once, state labels are distinct. Since the is no disjunction, conjunction, Kleene star, and optional sub-patterns in P, there is one $start(P)$ and one $end(P)$ event type per pattern P [1] There can be any number of event types in the set $mid(P)$. $start(P) \notin mid(P)$ and $end(P) \notin mid(P)$. An event type may be both $start(P)$ and $end(P)$, for example, in the pattern $A+$.

Transitions correspond to operators in P. They connect types of events that may be adjacent in a match of P. If a transition connects an event type E_1 with an event type E_2, then E_1 is a ***predecessor event type*** of E_2, denoted $E_1 \in P.predTypes(E_2)$. In Figure 2.1, $P.predTypes(A) = \{A, B\}$ and $P.predTypes(B) = \{A\}$.

Template Construction Algorithm. Algorithm 1 consumes a positive pattern P and returns the automaton-based representation of P, called *template* $\mathcal{T} = (S, T)$. The states S correspond to the event types in P (Line 2), while the transitions T correspond to the operators in P. Initially, the set T is empty (Line 2). For each event sequence $\mathsf{SEQ}(P_1, P_2)$ in P, there is a transition from $end(P_1)$ to $start(P_2)$ with label "SEQ" (Lines 3–5). Analogously, for each Kleene plus P_1+ in P, there is a transition from $end(P_1)$ to $start(P_1)$ with label "+" (Lines 6–8). Start and end event types of a pattern are computed by the auxiliary methods in Lines 10–19.

[1] We discuss how to drop these simplifying assumptions in Chapter 12.

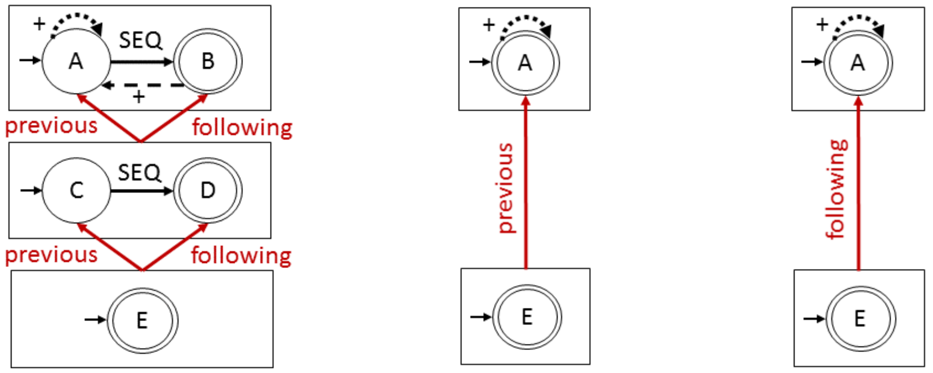
Algorithm 1 Template construction algorithm

```
Input: Positive pattern $P$
Output: Template $\mathcal{T}$
$generate(P)$ {
$S \leftarrow$ event types in $P$, $T \leftarrow \emptyset$, $\mathcal{T} = (S, T)$
for all SEQ$(P_1, P_2)$ in $P$ do
    $t \leftarrow (end(P_1), start(P_2))$, $t.label \leftarrow$ "SEQ"
    $T \leftarrow T \cup \{t\}$
for all $P_1+$ in $P$ do
    $t \leftarrow (end(P_1), start(P_1))$, $t.label \leftarrow$ " + "
    $T \leftarrow T \cup \{t\}$
return $\mathcal{T}$ }
$start(P)$ {
switch $P$ do
    case $E$ return $E$
    case $P_1+$ return $start(P_1)$
    case SEQ$(P_1, P_2)$ return $start(P_1)$ }
$end(P)$ {
switch $P$ do
    case $E$ return $E$
    case $P_1+$ return $end(P_1)$
    case SEQ$(P_1, P_2)$ return $end(P_2)$ }
```

Complexity Analysis. Let P be a pattern of length l (Definition 2.1). To extract all event types and operators from P, P is parsed once in $\Theta(l)$ time. For each operator, we determine its start and event types in $O(l)$ time. Thus, the time complexity is quadratic $O(l^2)$. The space complexity is linear in the size of the template $\Theta(|S| + |T|) = \Theta(l)$.

2.3 Template for a Pattern with Negative Sub-Patterns

We assume that negation appears within a sequence preceded and followed by positive sub-patterns. Furthermore, negation is applied either to an event sequence or a single event type. Thus, we classify patterns containing a negative sub-pattern N into the following three cases:



(a) $(\textsf{SEQ}(A+, \textsf{NOT SEQ}(C, \textsf{NOT } E, D), B))+$ (b) $\textsf{SEQ}(A+, \textsf{NOT } E)$ (c) $\textsf{SEQ}(\textsf{NOT } E, A+)$

Figure 2.2: Template for patterns with negative sub-patterns

Case 1. ***A negative sub-pattern is preceded and followed by positive sub-patterns***. A pattern of the form $P_1 = \textsf{SEQ}(P_i, \textsf{NOT } N, P_j)$ means that no trends detected by N may occur between the trends detected by P_i and P_j. A trend matched by N marks all events in the graph of the *previous* event type $end(P_i)$ as *invalid* to connect to any future event of the *following* event type $start(P_j)$. Only valid events of type $end(P_i)$ connect to events of type $start(P_j)$.

Example 2.2 *The pattern* $(\textsf{SEQ}(A+, \textsf{NOT SEQ}(C, \textsf{NOT } E, D), B))+$ *is split into one positive sub-pattern* $(\textsf{SEQ}(A+, B))+$ *and two negative sub-patterns* $\textsf{SEQ}(C, D)$ *and* E*. Figure 2.2(a) illustrates the previous and following links between the template for the negative sub-pattern and the event types in the template for its parent pattern.*

Case 2. ***A negative sub-pattern is preceded but not followed by a positive sub-pattern***. A pattern of the form $P_2 = \textsf{SEQ}(P_i, \textsf{NOT } N)$ means that no trends matched by N may occur after the trends detected by P_i until the end of I. A trend matched by N marks all previous events in the graph of P_i as *invalid*.

Case 3. ***A negative sub-pattern is followed but not preceded by a positive sub-pattern***. A pattern of the form $P_3 = \textsf{SEQ}(\textsf{NOT } N, P_j)$ means that no trends matched by N may

Algorithm 2 Pattern split algorithm

```
Input: Pattern $P$ with negative sub-patterns
Output: Set $S$ of sub-patterns of $P$
 1: $S \leftarrow \{P\}$
 2: $split(P)$ {
 3: switch $P$ do
 4:     case $P_i+$ : $S \leftarrow S \cup split(P_i)$
 5:     case SEQ$(P_i, P_j)$ : $S \leftarrow S \cup split(P_i) \cup split(P_j)$
 6:     case NOT $P_i$ :
 7:         $Parent \leftarrow S.getPatternContaining(P)$
 8:         $P_i.previous \leftarrow Parent.getPrevious(P)$
 9:         $P_i.following \leftarrow Parent.getFollowing(P)$
10:         $S.replace(Parent, Parent - P)$
11:         $S \leftarrow S \cup \{P_i\} \cup split(P_i)$
12: return $S$ }
```

occur after the start of I and before the trends detected by P_j. A trend matched by N marks all subsequent events in the graph of P_j as *invalid* until the end of I.

Example 2.3 *Figures 2.2(b) and 2.2(c) illustrate the templates for the patterns* SEQ$($ $A+$, NOT $E)$ *and* SEQ$($NOT $E, A+)$ *respectively. The first template has only a previous link, while the second template has only a following link between the template for the negative sub-pattern E and the event type A.*

Pattern Split Algorithm. Algorithm 2 consumes a pattern P, splits it into positive and negative sub-patterns, and returns the set S of these sub-patterns. Each sub-pattern in the set S has links to its previous and/or following event types in the parent pattern. At the beginning, S contains the pattern P (Line 1). The algorithm traverses P top-down. If it encounters a negative sub-pattern $P =$ NOT P_i, it finds the sub-pattern containing P, called $Parent$ pattern, using the auxiliary method *S.getPatternContaining(P)*. Then, the algorithm computes the previous and following event types of P_i using the auxiliary methods *Parent.getPrevious(P)* and *Parent.getFollowing(P)*. The algorithm removes P from $Parent$ (Lines 7–10). The pattern P_i is added to S and the algorithm is called recursively on P_i (Line 11). Since the algorithm traverses the pattern P top-down once, the time and space complexity are linear in the length of the pattern l, i.e., $\Theta(l)$.

There is always only one positive sub-pattern P from which negative sub-patterns are removed (Line 10). There can be any number of negative sub-patterns. Negative sub-patterns can be nested (Figure 2.2(a)). A pattern can have any number of children. For example, the patterns in Figure 2.2 have zero or one child. However, the pattern $\textsf{SEQ}(A, \textsf{NOT}\ B, C, \textsf{NOT}\ D, E)$ is split into one positive $P = \textsf{SEQ}(A, C, E)$ and two negative sub-patterns B and E such that B and E are children of P.

2.4 Event Matching Semantics

Event matching semantics are commonly used in the CEP literature [3, 7, 8] to express event queries for diverse streaming applications (Section 1.1). Informally speaking, these semantics differentiate between ***relevant events***, i.e., events that can extend an existing (partial) match under the most flexible skip-till-any-match semantics (see below), and ***irrelevant events*** that cannot. Relevant events either must extend existing matches or can be skipped to preserve opportunities for alternative matches. Irrelevant events either invalidate current incomplete matches or can be skipped.

Example 2.4 *In Figure 2.3, the pattern P = (SEQ(A+,B))+ is evaluated under various event matching semantics against the stream I. In the stream, letters denote types, while numbers represent time stamps, e.g., $a1$ is an event of type A with time stamp 1. Matches are depicted above the stream. They range from the shortest contiguous match $(a1, b2)$ to the longest non-contiguous match $(a1, b2, a3, a4, b6, a7, b8)$.*

Skip-Till-Any-Match Semantics (ANY for short) is the most flexible semantics that detects *all possible matches* by skipping *any* event in the stream as follows. For each event

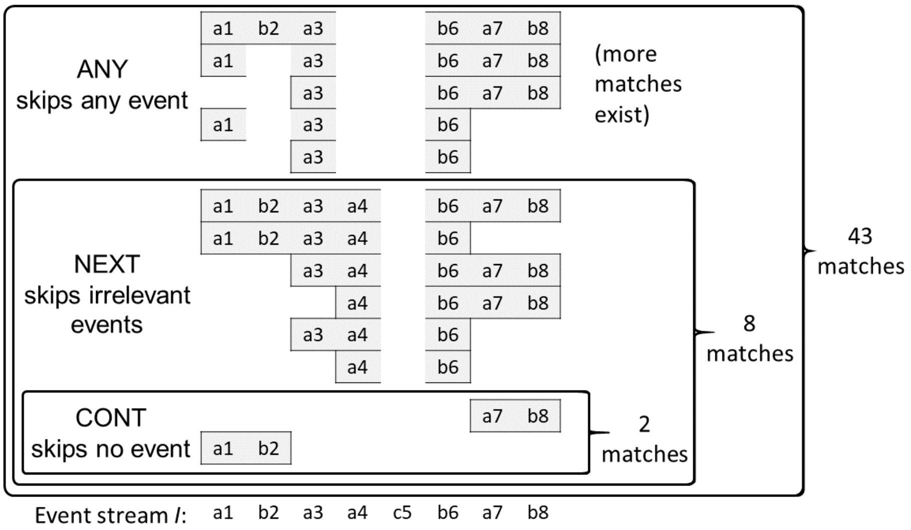


Figure 2.3: Matches of the pattern *P* = *(SEQ(A+, B))+* in the stream *I* = $\{$*a1, b2, a3, a4, b6, a7, b8*$\}$ under various event matching semantics

e and each match tr that can be extended by e, two possibilities are considered: (1) e is appended to the match tr to form a longer match $tr' = (tr, e)$, and (2) e is skipped and the match tr remains unchanged to preserve opportunities for alternative longer matches. If an event e can extend all matches, then the event e doubles the number of matches. Thus, the number of matches grows exponentially in the number of events in the worst case. Skip-till-any-match skips irrelevant events. Query q in Section 1.1 is evaluated under this semantics.

Example 2.5 *In Figure 2.3, when* $a7$ *arrives, the match* $(a3, b6)$ *is extended to* $(a3, b6, a7)$ *and the original match* $(a3, b6)$ *is also kept. Based on only 8 events in the stream, 43 matches are detected. Only some of them are shown for compactness. Irrelevant events are ignored, e.g.,* $c5$.

Definition 2.2 (Pattern Match Under Skip-Till-Any-Match) • ***Event Type****. If* $e \in I$ *and* $e.type = E$*, then the event type* E *has a match* e*, denoted* $e \in matches_{any}(E, I)$*.*

• ***Positive Event Sequence****. If* P_1 *and* P_2 *are patterns,* $(e_1, \ldots, e_m) \in matches_{any}(P_1,$ $I)$*,* $(e_{m+1}, \ldots, e_k) \in matches_{any}(P_2, I)$*, and* $e_m.time < e_{m+1}.time$*, then the event* *sequence operator* $\textsf{SEQ}(P_1, P_2)$ *has a match* $s = (e_1, \dots, e_m, e_{m+1}, \dots, e_k)$*, denoted* $s \in matches_{any}(\textsf{SEQ}(P_1, P_2), I)$*. We define the* ***START event*** $s.start = e_1$*, the* ***MID events*** $s.mid = \{e_2, \dots, e_{k-1}\}$*, and the* ***END event*** $s.end = e_k$ *of* s*.*

- ***Event Sequence with a Negative Sub-Pattern****. If* $P_1, N,$ *and* P_2 *are patterns,* $(e_1, \dots,\ e_m) \in matches_{any}(P_1, I)$*,* $(e_{m+1}, \dots, e_k) \in matches_{any}(P_2, I)$*,* $e_m.time < e_{m+1}.time$*,* $\nexists n \in matches_{any}(N, I)$ *with* $e_m.time < n.start.time$ *and* $n.end.time < e_{m+1}.time$*, then the event sequence operator* $\textsf{SEQ}(P_1,\ \textsf{NOT}\, N, P_2)$ *has a match* $s = (e_1, \dots, e_m, e_{m+1}, \dots, e_k)$*, denoted* $s \in matches_{any}(\textsf{SEQ}(P_1, \textsf{NOT}\, N, P_2), I)$*.*

If P *and* N *are patterns,* $s = (e_1, \dots, e_m) \in matches_{any}(P, I)$*,* $\nexists n \in matches_{any}(N, I)$ *with* $e_m.time < n.start.time$*, then the event sequence operator* $\textsf{SEQ}(P,\ \textsf{NOT}\, N)$ *has a match s, denoted* $s \in matches_{any}(\textsf{SEQ}(P, \textsf{NOT}\, N), I)$*.*

If N *and* P *are patterns,* $s = (e_1, \dots, e_m) \in matches_{any}(P, I)$*,* $\nexists n \in matches_{any}(N, I)$ *with* $n.end.time < e_1.time$*, then the event sequence operator* $\textsf{SEQ}(\textsf{NOT}\, N, P)$ *has a match s, denoted* $s \in matches_{any}(\textsf{SEQ}(\textsf{NOT}\, N, P), I)$*.*

START events, MID events, and END events of s are defined analogously to above.

- ***Kleene Plus****. If* P *is a pattern,* $\forall s_l \in \{s_1, \dots, s_k\}.\ s_l \in matches_{any}(P, I)$ *and* $s_l.end.time < s_{l+1}.start.time$*, then the Kleene plus operator* $P+$ *has a match* $tr = (s_1, \dots, s_k)$*, denoted* $tr \in matches_{any}(P+, I)$*. We define the* ***START event*** $tr.start = s_1.start$ *and the* ***END event*** $tr.end = s_k.end$ *of* tr*. All other events in* tr *are called* ***MID events****.*

Skip-Till-Next-Match Semantics (NEXT for short) is more restrictive than ANY because NEXT *skips only irrelevant events*, while all relevant events must be matched.

Example 2.6 *In Figure 2.3, the match* $(a3, b6)$ *does not conform to this semantics since it skipped over the relevant event* $a4$*. In contrast, the match* $(a3, a4, b6)$ *is valid since it skips no relevant events in between matched events.*

Semantics	Event sequence pattern	Kleene pattern
ANY	Polynomial	Exponential
NEXT, CONT	Linear	Polynomial

Table 2.1: Number of matches in the number of events [2, 3]

Definition 2.3 (Pattern Match Under Skip-Till-Next-Match) *If* $tr \in matches_{any}(P, I)$ *(Definition 2.2),* $\nexists tr' \in matches_{any}(P, I)$ *with* $tr.start = tr'start$*,* $tr.end = tr'.end$*, and* $tr.mid \subseteq tr'.mid$*, then* tr *is a match of* P *under the skip-till-next-match semantics, denoted* $tr \in matches_{next}(P, I)$*.*

Contiguous Semantics (CONT for short) is the most restrictive semantics since it *does not skip events*. Query q' in Section 1.1 is evaluated under the contiguous semantics.

Example 2.7 *In Figure 2.3,* $(a1, b2)$ *and* $(a7, b8)$ *are the only contiguous matches. Since the irrelevant event* $c5$ *cannot be ignored,* $a1, b2, a3,$ *and* $a4$ *cannot form contiguous matches with later events.*

Definition 2.4 (Pattern Match Under Contiguous Semantics) *If* $tr \in matches_{next}(P, I)$ *(Definition 2.3) and* $\nexists e \in I$ *such that* $tr.start.time < e.time < tr.end.time$*, then* tr *is a match of* P *under the contiguous semantics, denoted* $tr \in matches_{cont}(P, I)$*.*

Figure 2.3 illustrates the containment relationships among the sets of matches of P under different semantics.

The number of matches is determined by the following two factors (Table 2.1): (1) The presence of Kleene closure in a pattern P, which allows expressing arbitrarily long matches. (2) The event matching semantics under which the pattern P is evaluated. The number of matches of P ranges from linear to exponential in the number of matched events. While the number of matches can be exponential in the worst case, a query constrains its matches by predicates, grouping, and windows.

2.5 Event Query

Definition 2.5 (Event Query) *An event query q consists of six clauses:*

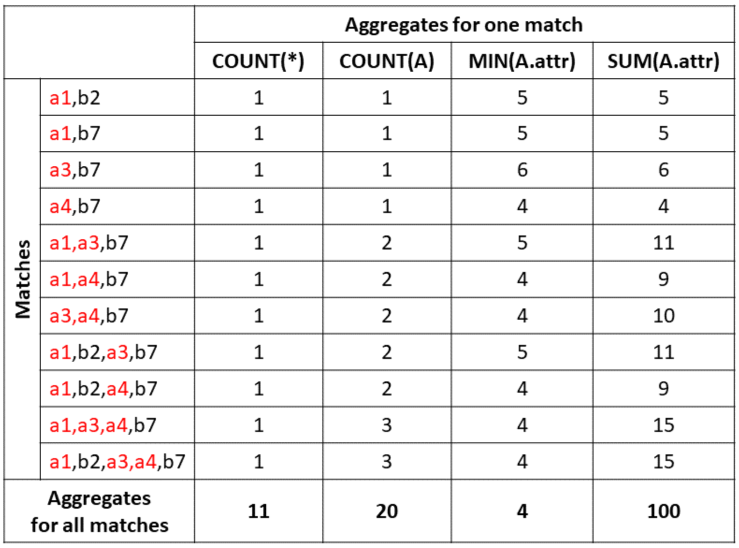
- *Aggregation result specification (optional* RETURN *clause),*
- *Pattern P (*PATTERN *clause),*
- *Event matching semantics S (*SEMANTICS *clause),*
- *Predicates θ (optional* WHERE *clause),*
- *Grouping G (optional* GROUP-BY *clause), and*
- *Window w (*WITHIN *and* SLIDE *clause).*

Predicates. We handle different classes of predicates differently. We distinguish between the following classes of predicates:

Predicates on single events either filter or partition the stream [2]. For example, the ***local predicate*** *(M.activity = passive)* in query q' in Section 1.1 selects only those heartbeat measurements that were taken during passive activities. The ***equivalence predicate*** *[patient]* in query q' requires all events in a match to have the same value of patient identifier. Thus, it partitions the stream by patient.

Predicates on adjacent events restrict the adjacency relation between events in a match (Definition 2.7). For example, the predicate $(M.rate <$ NEXT$(M).rate)$ requires heartbeat measurements to increase from one event to the next in a match of query q'.

Match Grouping and Aggregation. Within each window of query q, the matches of q are grouped by the values of grouping attributes G. Aggregates are computed per group. We focus on distributive (such as COUNT, MIN, MAX, SUM) and algebraic aggregation functions (such as AVG) since they can be computed incrementally [48]. **COUNT**$(*)$ returns the number of matches per group. Let tr be a match and *tr.*COUNT*(E)* be the number of events of type E in it. **COUNT*(E)*** corresponds to the sum of *tr.*COUNT(E) of all matches tr per group. Let *tr.*MIN*(E.attr)* be the minimal value of an attribute *attr* of events



		Aggregates for one match			
		COUNT(*)	COUNT(A)	MIN(A.attr)	SUM(A.attr)
Matches	a1,b2	1	1	5	5
	a1,b7	1	1	5	5
	a3,b7	1	1	6	6
	a4,b7	1	1	4	4
	a1,a3,b7	1	2	5	11
	a1,a4,b7	1	2	4	9
	a3,a4,b7	1	2	4	10
	a1,b2,a3,b7	1	2	5	11
	a1,b2,a4,b7	1	2	4	9
	a1,a3,a4,b7	1	3	4	15
	a1,b2,a3,a4,b7	1	3	4	15
Aggregates for all matches		**11**	**20**	**4**	**100**

Figure 2.4: Aggregation of matches of the pattern *P* = (SEQ(*A*+,*B*))+ in the stream *I* = {*a1, b2, a3, a4, b7*} under the skip-till-any-match semantics

of type E in tr. **MIN*(E.attr)*** returns the minimal value of *tr*.MIN$(E.attr)$ of all matches tr per group. **MAX*(E.attr)*** is defined analogously to MIN$(E.attr)$. Let *tr*.SUM*(E.attr)* be the sum of values of an attribute *attr* of events of type E in tr. **SUM*(E.attr)*** corresponds to the sum of *tr*.SUM*(E.attr)* of all matches tr per group. Lastly, **AVG*(E.attr)*** = SUM*(E.attr)* / COUNT*(E)* per group.

Example 2.8 *In Figure 2.4, the pattern P = (SEQ(A+,B))+ detects COUNT(*) = 11 matches in the stream I with five events under the skip-till-any-match semantics. There are COUNT(A) = 20 occurrences of a's in these matches.*

Let attr be an attribute of events of type A. Assuming that a1.attr = 5, a3.attr = 6, and a4.attr = 4, the minimal value of attribute attr in these matches is MIN(A.attr) = 4, while the maximal value of attr is MAX(A.attr) = 6. MAX(A.attr) is computed analogously to MIN(A.attr). MAX(a.attr) is not shown in Figure 2.4 for compactness.

The summation of all values of attr in all matches corresponds to SUM(A.attr) = 100. Lastly, the average value of attr in all matches is AVG(A.attr) = SUM(A.attr) / COUNT(A) = 5.

Definition 2.6 (Query Match, Event Trend, Event Sequence) *If $m = (e_1, \dots, e_k)$ is a match of the pattern P (Definition 2.1) under the event matching semantics S (Section 2.4), all events in the match m satisfy the predicates θ, carry the same values of the grouping attributes G, and are within one window w, then m is a* ***match*** *of q. We define $s = (e_i, \dots, e_j), 1 \leq i \leq j \leq k$, as a* ***sub-match*** *of P and $p = (e_1, \dots, e_l), l < k$, as a* ***prefix*** *of m of length l.*

If P is a Kleene pattern, m is called an ***event trend*** *and s is called an* ***event sub-trend****. If P is an event sequence pattern, m is called an* ***event sequence*** *and s is called an* ***event sub-sequence****.*

Definition 2.7 (Adjacent Events, Predecessor Event, Successor Event) *Let $e \in I$ be an event in a match of query q and $e' \in I$ be a new event. If q is evaluated under the* ***skip-till-any-match*** *semantics, the events e and e' are* ***adjacent*** *in a window w of q if the following conditions hold:*

- $e.time < e'.time$,
- $e.type \in P.predTypes(e'.type)$ *where P is the pattern of q,*
- *if there is a negated sub-pattern* NOT *N that is preceded by $e.type$ and followed by $e'.type$ in the pattern P, then $\nexists m \in matches_{any}(N, I)$ with $e.time < m.start.time$ and $m.end.time < e'.time$,*
- *e and e' satisfy the predicates θ of q,*
- *e and e' have the same values of grouping attributes G of q, and*
- *e and e' belong to the window w.*

If q is evaluated under the ***skip-till-next-match*** *semantics, the events e and e' are* ***adjacent*** *in a window w of q if they are adjacent in w under skip-till-any-match and*

$\nexists e'' \in I$ such that $e''.time < e'.time$ and e and e'' are adjacent in w under skip-till-any-match.

*If q is evaluated under the **contiguous** semantics, the events e and e' are **adjacent** in a window w of q if they are adjacent in w under skip-till-next-match and $\nexists e'' \in I$ such that $e.time < e''.time < e'.time$.*

*If e and e' are adjacent in w, then e is called a **predecessor event** of e', and e' is called a **successor event** of e in w.*

2.6 Assumptions

Data. For simplicity, we assume that complete and precise events arrive in-order by time stamps. Uncertain [49, 50] and out-of-order [51, 52, 53, 54] event streams are orthogonal problems that are out of the scope of this dissertation. Multiple events may have the same time stamp.

Query. We assume that:

1) A negation operator always appears within an event sequence operator and is applied either to an event sequence operator or an event type. In particular, negation may not be the outer most operator in a pattern. If a negative sub-pattern is preceded or followed by a pattern P, P must be positive.

2) A pattern does not contain Kleene star, optional sub-patterns, conjunction, nor disjunction. However, our techniques could be extended to support these pattern operators (Chapter 12).

3) An event type may appear at most once in a pattern. A straightforward extension of our approach allows dropping this assumption (Chapter 12 and Section 24.3).

4) An event query is evaluated under one of the event matching semantics defined in Section 2.4. However, other semantics are possible. They are subject to future research

described in Section 28.3.1.

5) We focus on predicates on single events and predicates on a pair of adjacent events in a match. Predicates on non-adjacent events in a match are subject for future research (Section 28.3.2). Constraints on event trend length can be supported as discussed in Chapter 12.

6) In Parts I and II, we propose optimization strategies for a single event query. Multi-query optimization techniques are left for future research (Section 28.2).

Resources. We evaluate our approach on a single multi-core machine. However, several distribution strategies are possible to further improve scalability. They are subject to future research described in Section 28.1.

Part I

Event Trend Detection

3

CET Approach Overview

In this chapter, we focus on the detection of the subset of all possible matches, namely, *Complete Event Trends (CETs)*. Informally speaking, an event trend is complete if it is not part of another trend. Expeditious CET detection is required by many streaming applications such as financial fraud detection (query q in Section 1.1).

We assume that an event query has a flat positive Kleene pattern (Definition 2.1) and is evaluated under the most flexible skip-till-any-match semantics (Section 2.4). Such a query is called a ***CET query*** in the following.

3.1 Complete Event Trend

Definition 3.1 *(**Complete Event Trend (CET)**) An event trend tr matched by a query q in the window w (Definition 2.6) is **complete** if there is no other trend tr' matched by q in w such that each event in tr is also in tr'. Otherwise, tr is called **incomplete**.*

Example 3.1 *In Figure 1.2, two trends $tr_1 = (c_1, c_2)$ and $tr_2 = (c_1, c_4, c_5)$ have been detected by query q in Section 1.1 when c_6 arrives. The event c_6 cannot be appended to these trends since the predicates of the query q (Section 1.1) would be violated (c_2.destination*

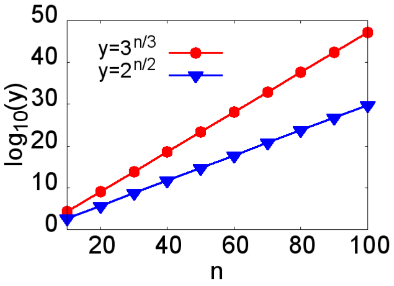


Figure 3.1: Number of CETs y for n events and the size of event groups $x = 2$ versus $x = 3$

= c_5.*destination* = *A* $\neq$ c_6.*source* = *C*.

However, c_6 *can be appended to the prefix of* tr_2 *to form a new trend* $tr_3 = (c_1, c_2, c_6)$*. Indeed,* c_6 *is matched by the pattern of* q *(*$c_6.type = Check$*), satisfies the predicate of* q *(*c_4*.destination* = c_6*.source* = *C), and is within the window of* q*. The trends* tr_1*–*tr_3 *are complete since none of them is part of another trend.*

3.2 CET Detection Optimization Problem

Many CET detection applications are time-critical (Section 1.1). Thus, our goal is to minimize the CPU processing time for CET detection.

Problem Statement. Given a CET query q, an event stream I, and an available memory limit M, the CET detection optimization problem is to detect all CETs matched by the query q in the stream I while *minimizing the CPU costs* and staying within the memory limit M.

This problem is prohibitively expensive with respect to both CPU processing time and memory consumption since the number of CETs is exponential. Below we will show that the problem of determining the maximal number of CETs that can be constructed from n events is equivalent to the problem of dividing n elements into groups such that the product of all group sizes is maximal. In particular, we will show that to maximize the product, all groups must have the same size $x = 3$.

Theorem 3.1 (Maximal Number of CETs) *An event query (Definition 2.5) detects* $3^{\frac{n}{3}}$ *CETs from* n *events in the worst case.*

Proof: Let $x_i \in \mathbb{N}$ be the number of successor events of one matched event e_i (Definition 2.6). We first show that the number of CETs to be constructed for a given set I of n events is maximal when x_i is identical for all events $e_i \in I$. Then the maximum number of CETs is given by $maximize \prod x_i$, subject to $\sum x_i = n$. The method of Lagrange multiplier in mathematical optimization [55] introduces an auxiliary function $\mathcal{L}(x_i, \lambda) = \prod x_i + \lambda \cdot \sum x_i$. Solving the following equation $\bigtriangledown_{x_i,\lambda}\mathcal{L}(x_i, \lambda) = 0$, we have $x_1 = \cdots = x_i = x_{i+1} = \cdots = x_{n/x} = x$. Thus, the maximum number of CETs that can be constructed from n events is $y = x^{\frac{n}{x}}$.

Now our goal is to determine the global maximum of this continuous function on the interval $[1, n]$. Below we derive the critical value of x following the standard approach [55]. First we take the logarithm of both sides:

$$\ln y = \ln x^{\frac{n}{x}} = \frac{n}{x} \ln x.$$

We then differentiate both sides:

$$\frac{y^t}{y} = (\frac{n}{x})' \ln x + (\ln x)' \frac{n}{x} = \frac{0 \cdot x - n \cdot 1}{x^2} \ln x + \frac{n}{x^2} = \frac{n}{x^2}(1 - \ln x).$$

Lastly, we multiply by y and substitute y with $\frac{n}{x} \ln x$.

$$y^t = y\frac{n}{x^2}(1 - \ln x) = x^{\frac{n}{x}} \frac{n}{x^2}(1 - \ln x) = x^{\frac{n}{x}-2} n(1 - \ln x).$$

We have $y^t = 0$ if $x = e$ (e is Euler's number here). Since $x \in \mathbb{N}$, we round x to its closest natural number 3. Indeed, $\forall n \in \mathbb{N},\ n > 1,\ 3^{\frac{n}{3}} > 2^{\frac{n}{2}}$ (Figure 3.1). Thus, $y = 3^{\frac{n}{3}}$ is the absolute maximum value of the function. ■

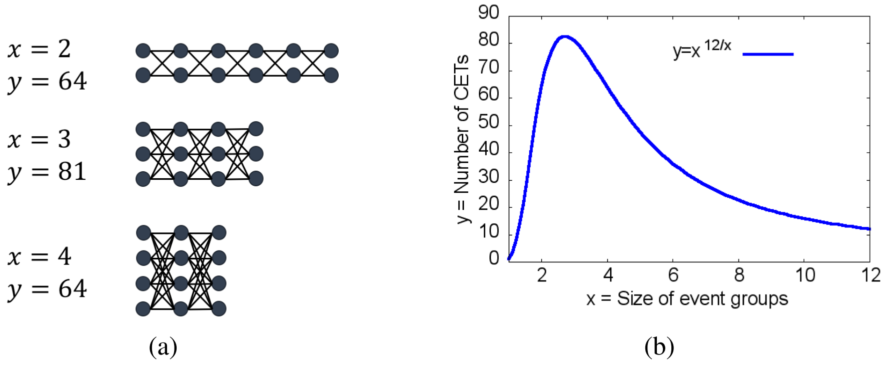


Figure 3.2: Number of CETs y for $n = 12$ events and the size of event groups x

Example 3.2 *In Figure 3.2(a), we consider three scenarios with the same number of events* $n = 12$*. These events are divided into* $\lceil \frac{n}{x} \rceil$ *groups with at most* x *events in each group where* $n, x \in \mathbb{N}$*,* $1 \leq x \leq n$*. Events in one group are not adjacent to each other. Each event in a group is adjacent to each event in the preceding and following groups. A pair of adjacent events is connected by an edge. Then the number of CETs corresponds to the product of all group sizes:* $y = \prod_{i=1}^{\frac{n}{x}} x = x^{\frac{n}{x}}$.

Given $n = 12$ *events, Figure 3.2(b) shows the number of CETs* $y = x^{\frac{n}{x}}$ *on the y-axis while varying the size of event groups* x *on the x-axis. If* $x = 3$*,* $y = 3^{\frac{n}{3}} = 81$ *is maximal. The same observation holds for any* $n \in \mathbb{N}$.

3.3 CET Framework

Given a CET query, our CET framework (Figure 3.3) extracts events matched by the query from an event stream and encodes their CET relationships in a compact data structure, called ***CET Graph***.

Based on the graph, we propose a family of ***CET Detection*** algorithms ranging from the memory-optimized M-CET to the CPU-time-optimized T-CET solution. To trade off between CPU and memory costs, we develop the Hybrid CET detection algorithm H-CET.

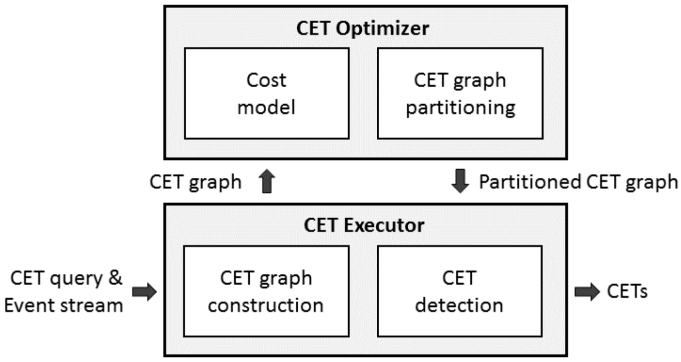


Figure 3.3: CET framework

Namely, we partition the CET graph into smaller graphlets. Based on the partitioned graph, H-CET caches partial CETs per graphlet and then stitches these partial CETs together to form the final CET results.

CET graph partitioning faces the trade-off that finer-grained partitioning plans reduce the memory consumption since CETs across graphlets are not stored while increasing the execution time, and vice versa. We thus design a cost-driven ***CET Optimizer*** that finds an optimal partitioning plan with minimal CPU execution costs yet within the available memory limit.

4

Baseline CET Detection

Our baseline CET detection algorithm (Algorithm 3) consumes a CET query q and a stream I. As new events arrive, it incrementally constructs and stores all CETs per window. For each event $e \in I$, the following cases are possible.

Case 1: The event e starts a new CET if e is matched by the pattern P and satisfies the predicates θ of the query q, it can be appended neither to an existing CET nor to its prefix (Lines 3 and 11 in Algorithm 3).

Case 2: The event e extends an existing CET t if the last event e_l in t is adjacent to e (Definition 2.6, Lines 5–7). The auxiliary method *isAdjacent(q, e_l, e)* determines whether e_l and e are adjacent.

Case 3: The event e extends the *longest compatible prefix* p of an existing CET t (Lines 8–9).

Definition 4.1 *(**Longest Compatible Prefix**.) Let q be a CET query, $t = (e_1, \dots, e_k)$ be a CET matched by q, and $e \in I$ be an event. Then, $p = (e_1, \dots, e_l), l < k$, is the **longest prefix of** t **that is compatible with** e if e_l and e are adjacent and $\nexists e_i \in \{e_{l+1}, \dots, e_k\}$ such that e_i and e are adjacent.*

Appending e to *all* compatible prefixes of t (instead of only the longest compatible

Event	Event trends	Explanation
$c_1 : A \to B$	(c_1)	Case 1: Create a new CET
$c_2 : B \to C$	(c_1, c_2)	Case 2: Append to a CET
$c_3 : B \to D$	(c_1, c_2), (c_1, c_3)	Case 3: Append to the longest compatible prefix of a CET
$c_4 : D \to E$	(c_1, c_2), (c_4), (c_1, c_3, c_4)	Eliminate incomplete trends

Table 4.1: Three cases of the baseline algorithm

prefix p) would produce incomplete trends (contained in (p, e)). Analogously, if an event e is appended to a CET t to form a longer CET (t, e) (Case 2), e is not appended to a prefix p of t (Case 3) because the trend (p, e) in contained in (t, e) and thus is not a CET. Lastly, if e is appended to a CET or its prefix (Case 2 or 3), e cannot start a new CET (Case 1). In other words, only one of the above cases applies to an event and a trend.

The longest compatible prefix p is computed by the auxiliary method *getLongestCompatiblePrefix(q, t, e)* (Lines 15–22). The auxiliary method *eliminateIncompleteTrends(T_{new})* eliminates the incomplete trends from the set of newly formed trends T_{new} (Lines 23–29). We now illustrate this baseline algorithm using Example 4.1.

Example 4.1 *Assume query q in Section 1.1 is evaluated against the events c_1–c_4 in Table 4.1. At the beginning, the set of CETs is empty. When c_1 arrives, a new CET $tr_1 = (c_1)$ is started (Case 1). When c_2 arrives, it is adjacent to c_1 and thus extends tr_1 (Case 2). When c_3 arrives, it is adjacent to c_1 but not c_2. In other words, c_1 is the longest prefix of tr_1 that is compatible with c_3. The newly formed trend is $tr_2 = (c_1, c_3)$ (Case 3). Lastly, when c_4 arrives it is compared to the existing CETs $tr_1 = (c_1, c_2)$ and $tr_2 = (c_1, c_3)$. Since c_4 can be appended neither to tr_1 nor to its prefix, it starts a new trend $tr_3 = (c_4)$ (Case 1). However, c_4 can extend $tr_2 = (c_1, c_3)$ which makes the trend tr_3 incomplete. Thus, tr_3 is eliminated.*

Theorem 4.1 (Correctness of Baseline CET Detection) *Baseline CET detection algorithm is correct.*

Proof: We prove the correctness of Algorithm 3 by induction over the number of events per window n.

Base case: $n = 1$. In this case, the set of CETs T is empty and an event e is inserted into it if e is matched by the pattern P and satisfies the predicates θ (Case 1, Line 11). Since we assumed that the query q has a positive flat Kleene pattern (Chapter 3), e is both a START and an END event. In other words, (e) is the only CET. This result is correct.

Induction assumption: T is correct after processing n events.

Induction step: $n \rightarrow n + 1$. According to the induction assumption, the set of CETs T contains all CETs that are detected from n events. A new event e arrives. Assume that after processing e, the set T is not correct. The following cases are possible.

1) There is a trend $t \in T$ which is not a CET. By Definition 3.1, there must be another trend $t' \in T$ such that each event in t appears in t'. However, incomplete trends are eliminated in Line 13. Thus, T contains only CETs.

2) A CET t is missing in T. By induction assumption, T is correct after processing n events. Thus, the trend t must have the from (t', e) where t' is either empty, or a CET detected from n events, or a longest compatible prefix of a CET detected from n events. However, these three cases are covered by Algorithm 3. Thus, no CET is missing in T. ■

Theorem 4.2 (Complexity of Baseline CET Detection) *Baseline CET detection has exponential time and space complexity in the number of events per window.*

Proof: Let n be the number of events in the query window, y be the number of CETs, l be the maximal length of a CET, and r be the number of predicates in the query q. The for-loop in Lines 2–13 is called n times. The for-loop in Lines 5–12 is executed for each CET, y times. The *isAdjacent(q, e_l, e)* method has the CPU cost $O(r)$ since each predicate needs to be executed to conclude event adjacency. The *getLongestCompatiblePrefix(q, t, e)* method has the CPU cost $O(lr)$ since in the worst case the adjacency is checked for each possible prefix of the trend t. Finally, the *eliminateIncompleteTrends(T_{new})* method has the CPU cost $O(y^2l)$ since each event trend is compared to all other event trends and the cost of one comparison is $O(l)$. Altogether, the CPU cost is $O(nyr^2l + ny^2l)$.

According to Theorem 3.1, $y = 3^{\frac{n}{3}}$ in the worst case and thus y is the dominant factor in the overall CPU complexity. Thus, the execution time of Algorithm 3 is exponential in the number of events n in the window:

$$CPU_{BL} = O(n(yr^2l + y^2l)) = O(n(3^{\frac{n}{3}}r^2l + 3^{\frac{2n}{3}}l)).$$

The memory consumption of Algorithm 3 is also exponential in n since all CETs are stored:

$$Mem_{BL} = O(yl) = O(3^{\frac{n}{3}}l).$$

■

Drawbacks of the Baseline CET Detection Algorithm. Algorithm 3 has exponential CPU and memory costs because an event sub-trend can be part of several CETs. For example, the event c_1 is part of all CETs in Table 4.1. The baseline approach replicates such sub-trends for each CET that contains them. Worse yet, when a new event arrives and is compared to the previously constructed CETs, repeated computations arise since the new event may need to be compared to a common sub-trend within each CET that contains this sub-trend.

Algorithm 3 Baseline CET detection algorithm

```
Input: CET query $q$ with pattern $P$ and predicates $\theta$, event stream $I$
Output: CETs $T_{prev}$
$T_{prev} \leftarrow \emptyset$
for all $e \in I$ do $T_{new} \leftarrow \emptyset$
    if $T_{prev} = \emptyset$ then $T_{new} \leftarrow (e)$                                // Case 1
    else
        for all $t \in T_{prev}$ do
            $e_l \leftarrow$ last event in $t$
            if $isAdjacent(q, e_l, e)$ then $T_{new} \leftarrow T_{new} \cup (t, e)$          // Case 2
            else $p \leftarrow getLongestCompatiblePrefix(q, t, e)$
                if $p.length > 0$ then $T_{new} \leftarrow T_{new} \cup t \cup (p, e)$       // Case 3
                else
                    if $e$ is matched by $P$ and satisfies $\theta$ then
                        $T_{new} \leftarrow T_{new} \cup t \cup (e)$                   // Case 1
        $T_{prev} \leftarrow eliminateIncompleteTrends(T_{new})$
return $T_{prev}$
$getLongestCompatiblePrefix(q, t, e)$ {
$i \leftarrow t.length - 1$;  $e_i \leftarrow$ event at $i^{th}$ position in $t$
while $i \geq 0$ and $!isAdjacent(q, e_i, e)$ do
    $i \leftarrow i - 1$
    if $i \geq 0$ then $e_i \leftarrow$ event at $i^{th}$ position in $t$
if $i \geq 0$ then return trend with all events in $t$ till $i^{th}$ position
else return empty trend
}
$eliminateIncompleteTrends(T)$ {
$D \leftarrow \emptyset$
for all $tr_1 \in T$ do
    for all $tr_2 \in T$ with $tr_1 \neq tr_2$ do
        if each event in $tr_1$ is in $tr_2$ then $D \leftarrow D \cup tr_1$
return $T \setminus D$
}
```

5

Graph-Based CET Detection

5.1 Compact CET Graph Encoding

The baseline algorithm is inefficient because it does not exploit sharing opportunities due to common event sub-trends in CETs. To overcome this limitation, we propose a compact data structure, called a CET graph. The graph prevents event duplication by storing each matched event exactly once. It avoids repeated computations since each new event is compared to a common event sub-trend at most once.

Given a CET query and an event stream, the vertices of the graph correspond to events in the stream that are matched by the query, while the edges connect events that are adjacent in a CET. Thus, a path in the CET graph from a vertex without ingoing edges (called ***first event***) to a vertex without outgoing edges (called ***last event***) corresponds to one CET.

Definition 5.1 *(**CET Graph**) Given a CET query q and an event stream I, the CET graph $G = (V, E)$ is a directed acyclic graph with a set of vertices V and a set of edges E. The vertices $V \subseteq I$ are events matched by q. For two events $e_1, e_2 \in V$, there is an edge $(e_1, e_2) \in E$ if e_1 and e_2 are adjacent in a CET matched by q.*

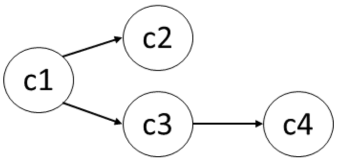


Figure 5.1: CET graph for Example 4.1

Algorithm 4 CET graph construction algorithm

```
Input: CET query $q$ with pattern $P$ and predicates $\theta$, event stream $I$
Output: CET graph $G$
 1: $sec \leftarrow 0;\ V \leftarrow \emptyset;\ E \leftarrow \emptyset;\ G \leftarrow (V, E)$
 2: for all $e \in I$ do
 3:     if $e.time > sec$ then $sec \leftarrow e.time$
 4:     $Pr \leftarrow G.getPredecessorEvents(sec, q, e)$
 5:     if $e$ is matched by $P$ and satisfies $\theta$ then $V \leftarrow V \cup e$
 6:         for all $e_l \in Pr$ do $E \leftarrow E \cup (e_l, e)$
 7: return $G$
 8: $getPredecessorEvents(sec, q, e)$ {
 9: $Pr \leftarrow \emptyset;\ L \leftarrow$ events in $G$ with time stamp $sec$
10: while $V.hasWhiteVertices()$ do
11:     for all $e_l \in L$ do
12:         if $e_l.hasBlackSuccessorEvent()$ then $e_l.color \leftarrow black$
13:         else
14:             if $isAdjacent(q, e_l, e)$ then $Pr \leftarrow Pr \cup e_l;\ e_l.color \leftarrow black$
15:             else $e_l.color \leftarrow gray$
16:     $sec \leftarrow sec - 1;\ L \leftarrow$ events in $G$ with time stamp $sec$
17: return $Pr$
18: }
```

Example 5.1 *The CET graph for Example 4.1 is depicted in Figure 5.1. The paths from the first event c_1 to the last events c_2 and c_4 correspond to two CETs.*

The CET graph construction algorithm (Algorithm 4) is analogous to the baseline algorithm (Algorithm 3), except that it updates the CET graph instead of CETs.

We assume that event time stamps are in seconds and events arrive in order by time stamps (Section 2.6). For each event e in the stream, the auxiliary method *getPredecessorEvents(sec, q, e)* returns the set of predecessor events Pr of e (Line 4). To this end, we visit all vertices in the graph G in reverse order by time stamps. Initially, all vertices in the graph are while. We color an event e_l in black if e_l (or a successor event of e_l) and e are adjacent (Lines 12–14). We color an event e_l in gray if e_l (and all its successor events) and e are not adjacent, e.g., due to violated predicates of the query q (Line 15). Only an event e_l without black successor events may be a predecessor event of e to ensure that the graph G captures only CETs (Line 14). If an event e starts a new CET or has predecessor events, e is inserted into the graph G and all its predecessor events connect to e (Lines 5–6). The graph G is returned (Line 7).

Theorem 5.1 (Correctness of a CET Graph) *Let G be the CET graph for a query q and a stream I. Let $\mathcal{P}$ be the set of paths from a first to a last event in G. Let $\mathcal{T}$ be the set of CETs detected by q in I. Then, the set of paths $\mathcal{P}$ and the set of CETs $\mathcal{T}$ are equivalent. That is, for each path $p \in \mathcal{P}$ there is a CET $tr \in \mathcal{T}$ with the same events in the same order, and vice versa.*

Proof: ***Correctness***. For each path $p \in \mathcal{P}$, there is a CET $tr \in \mathcal{T}$ with same events in the same order, i.e., $\mathcal{P} \subseteq \mathcal{T}$. We prove this statements by induction over the number of events per window n.

Base case: $n = 1$. In this case, the graph G is empty and an event e is inserted into the graph if e is matched by the pattern P and satisfies the predicates θ (Line 5). Since we assumed that the query q has a positive flat Kleene pattern (Chapter 3), e is both a START and an END event. The event e is the only path in G and the only CET matched by q.

Induction assumption: The graph G is correct after processing n events.

Induction step: $n \rightarrow n + 1$. According to the induction assumption, for each path $p \in \mathcal{P}$ in the graph, there is a CET $tr \in \mathcal{T}$ with same events in the same order. A new event e arrives. Assume that after processing e, the graph G is not correct. The following cases are possible.

1) There is a path $p \in \mathcal{P}$ in the graph G that does not correspond to a CET. The path p corresponds to an incomplete trend t. By Definition 3.1, there must be another trend t'

such that each event in t appears in t'. Since t' is also encoded into the graph G, assume the CET t' corresponds to a path p'. Each event in p appears in p'. Since only CETs are extracted from the graph G, t is not extracted. Thus, each path corresponds to a CET.

2) A CET t is not encoded into the graph G, i.e., there is no path p that corresponds to t. By induction assumption, G is correct after processing n events. Thus, the trend t must have the from (t', e) where t' is either empty, or a CET detected from n events, or a longest compatible prefix of a CET detected from n events. However, these three cases are covered by Algorithm 4. Thus, no CET is missing in G.

Completeness. For each CET $tr \in \mathcal{T}$, there is a path $p \in \mathcal{P}$ with same events in the same order, i.e., $\mathcal{T} \subseteq \mathcal{P}$. Let $tr \in \mathcal{T}$ be a CET. We first prove that all events in tr are inserted into the graph G. Then we prove that these events are connected by directed edges such that there is a path p that visits these events in the order in which they appear in the CET tr. An event e is inserted into the graph if e is matched by the pattern P and satisfies the predicates θ. Thus, all events of the CET tr are inserted into the graph G. The first statement is proven. *All* events in the graph that are adjacent to e in a CET are connected to e. An event e_l is adjacent if all conditions in Definition 2.7 hold and, in addition, there is no successor event of e_l that is adjacent to e to ensure that only CETs are encoded into the graph. Since events are processed in order by their time stamps, edges connect previous events with more recent events. The second statement is proven. ■

Theorem 5.2 (Complexity of CET Graph Construction) *The CET graph construction algorithm has quadratic time and space complexity in the number of events.*

Proof: Let q be a CET query, n be the number of events in the window of q, and r be the number of predicates in q. The outermost for-loop is called for each event, i.e., $O(n)$ times. The event e is compared to all previous events in the graph to find predecessor events. Since all predicates have to be evaluated, the CPU costs are $O(nr)$. Altogether, the CPU costs are quadratic in n: $O(n^2r)$ where n is the dominating factor since $r < n$ for high-rate event streams and meaningful queries. The memory costs are also quadratic in n because the number of vertices is $O(n)$, while the number of edges is $O(n^2)$ if each event is adjacent to all other events in the graph. ■

Our cost model in Table 5.1 and experiments in Section 8.2 demonstrate that the CET graph considerably accelerates CET detection compared to the baseline algorithm and state-of-the-art techniques, while the graph-maintenance costs are negligible compared to the overall costs of CET detection.

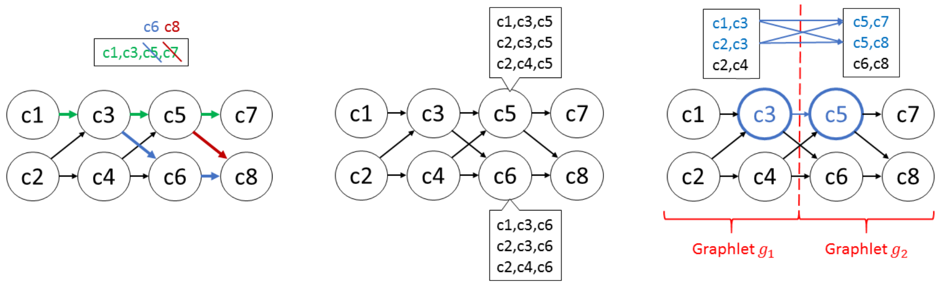
5.2 Spectrum of Graph-based CET Detection

Based on the CET graph, the traditional graph traversal algorithms such as Depth First Search (DFS) and Breadth First Search (BFS) can be applied to extract all CETs. These algorithms optimize the usage of critical resources such as memory and CPU. We distinguish between the following two algorithms at the extreme ends of the CPU versus memory trade-off spectrum:

The **M-CET algorithm** is *memory-optimized* since it stores only one CET during the DFS traversal. For example, Figure 5.2(a) shows the CET that was constructed after the colored edges had been traversed. Since no intermediate results are stored, the algorithm applies backtracking to find alternative paths through the graph. Thus, an edge is retraversed for each CET.

The **T-CET algorithm** is *CPU-time-optimized* since it traverses each edge exactly once. To this end, it stores all CETs found so far during the BFS traversal. Thus, no backtracking is necessary. For example, Figure 5.2(b) shows all CETs when events c_5 and c_6 are reached.

Complexity Analysis. In terms of memory utilization, both algorithms store the CET



(a) M-CET (b) T-CET (c) H-CET

Figure 5.2: Approaches to CET detection

	M-CET	**T-CET**	**H-CET**
Space complexity = Graph + CET number * CET length	$\Theta(\|V\|+\|E\|)+$ $O(\|V\|)$	$\Theta(\|V\|+\|E\|)+$ $O(3^{\frac{\|V\|}{3}}\|V\|)$	$\Theta(\|V\|+\|E\|)+$ $O(\sum_{i=1}^{k} 3^{\frac{\|V_i\|}{3}}\|V_i\|)+O(\|V\|)$
Time complexity = Graph construction + Graph traversal + CET update	$O(\|V\|^2)+$ $O(3^{\frac{\|V\|}{3}}\|V\|)+$ $O(3^{\frac{\|V\|}{3}}\|V\|)$	$O(\|V\|^2)+$ $\Theta(\|E\|)+$ $O(3^{\frac{\|V\|}{3}})$	$O(\|V\|^2)+$ $\Theta(\sum_{i=1}^{k}\|E_i\|)+O(\sum_{i=1}^{k} 3^{\frac{\|V_i\|}{3}})+$ $O(3^{\frac{\|V\|}{3}}k)+O(3^{\frac{\|V\|}{3}}k)$

Table 5.1: Cost model

graph itself, that is, $|V|$ vertexes and $|E|$ edges. Storing one CET requires at most $|V|$ space since in an extreme case all events build one CET. Thus, the memory cost of the M-CET algorithm is:

$$mem_{M-CET} = \Theta(|V|+|E|)+O(|V|).$$

The memory requirement for all CETs in the extreme case can be bounded by multiplying the maximal number of CETs $3^{\frac{|V|}{3}}$ by the maximal length of a CET $|V|$. Hence, the memory cost of the T-CET algorithm is:

$$mem_{T-CET} = \Theta(|V| + |E|) + O(3^{\frac{|V|}{3}}|V|).$$

Both algorithms construct the CET graph in $O(|V|^2)$ time (Theorem 5.2). The M-CET algorithm stores only one CET and thus performs at most $3^{\frac{|V|}{3}}|V|$ edge traversals and at most $3^{\frac{|V|}{3}}|V|$ CET updates. Altogether, the CPU cost of the M-CET algorithm is thus:

$$cpu_{M-CET} = O(|V|^2) + O(3^{\frac{|V|}{3}}|V|) + O(3^{\frac{|V|}{3}}|V|).$$

In contrast, the T-CET algorithm stores all CETs found so far and thus traverses an edge exactly once. In addition, it updates all CETs detected so far (Figure 5.2(b)). According to Theorem 3.1, the number of CETs is maximal if there are $\frac{|V|}{3}$ groups with 3 events at each group. Thus, the T-CET algorithm performs $O(\sum_{i=1}^{\frac{|V|}{3}} 3^i) = O(3^{\frac{|V|}{3}})$ CET updates in total. Its CPU costs are:

$$cpu_{T-CET} = O(|V|^2) + \Theta(|E|) + O(3^{\frac{|V|}{3}}).$$

In a nutshell, by storing all CETs detected so far, T-CET reduces the CPU costs for graph traversal from exponential to linear and the CPU costs of CET updates by the multiplicative factor $|V|$. On the down side, the memory requirement of T-CET degrades from linear to exponential. Thus, it risks exceeding the available memory when event queries with long windows are evaluated against high-rate event streams.

5.3 Hybrid CET Detection Algorithm

To achieve our goal of minimizing the CPU costs without running out of memory, our CET optimizer partitions the CET graph into smaller graphlets as defined in Chapters 6–7. Based on the partitioned graph, we now propose the **H-CET algorithm** (*H* for hybrid) that exploits the best of both M-CET and T-CET approaches in a dynamic programming fashion. The H-CET algorithm takes two steps: (1) The T-CET algorithm is applied to each graphlet to extract and cache CETs per graphlet. (2) The M-CET algorithm is applied to stitch these partial CETs within graphlets into final CETs across graphlets.

Complexity Analysis. Figure 5.2(c) illustrates that every edge in the cut between two graphlets requires concatenating the respective CETs within these graphlets to construct final results. This concatenation provokes additional CPU overhead. In other words, the smaller the graphlets, the fewer CETs per graphlet are computed and stored but the higher the CPU overhead of constructing the final CETs from these partial results becomes.

The memory (CPU) costs of H-CET correspond to the sum of the memory requirement to store the CET graph itself (CPU time to construct CET graph), the memory (CPU) costs of T-CET within graphlets, and the memory (CPU) cost of M-CET across graphlets. In Table 5.1, k denotes the number of graphlets. The cost of graph partitioning, ignored here, will be determined in Section 7.1.

The memory costs of H-CET are exponential in the number of events *per graphlet*, not in the number of events *per query window* as for T-CET. Thus, H-CET can achieve in several orders of magnitude memory costs reduction for high-rate event streams compared to T-CET as demonstrated by our experiments in Section 8.2.

Example 5.2 *Partitioning even the small graph in Figure 5.2(c) reduces the memory costs almost 3-fold. The memory requirement of T-CET is* $8 + 10 + 3^{\frac{8}{3}} * 8 \approx 168$*, while for H-CET it is* $8 + 10 + 3^{\frac{4}{3}} * 4 + 3^{\frac{4}{3}} * 4 + 8 \approx 61$.

The CPU costs increase by 1.5% due to partitioning. The CPU cost of T-CET is $8^4 + 10 + 3^{\frac{8}{3}} \approx 4125$*, while for H-CET it is* $8^4 + 3 + 3 + 3^{\frac{4}{3}} + 3^{\frac{4}{3}} + 3^{\frac{8}{3}} * 2 + 3^{\frac{8}{3}} * 2 \approx 4186.$

6

Foundations of CET-Graph Partitioning Problem

In this chapter, we study the properties of the graph partitioning search space to efficiently find an optimal CET-graph partitioning plan (Chapter 7).

Definition 6.1 ***(Optimal CET-Graph Partitioning Plan.)*** *Let $G = (V, E)$ be a CET graph. A partitioning plan p of G into k graphlets is a set of k sub-graphs $p = \{g_1 = (V_1, E_1) \dots, g_k = (V_k, E_k)\}$ such that $k \in \mathbb{N}, 1 \leq k \leq |V|$, $V = V_1 \uplus ... \uplus V_k$ and $E = E_1 \uplus ... \uplus E_k \uplus E_c$ where E_c is the set of cut edges that connect events in different graphlets.*

Let M be a memory limit, G be a CET graph, and P be the set of all partitioning plans of G. For a partitioning plan $p \in P$, let $cpu(p)$ and $mem(p)$ denote the CPU and memory costs of H-CET for the graph G partitioned by p, respectively. An optimal partitioning plan of G is $p_{opt} \in P$ such that $mem(p_{opt}) \leq M$ and $\nexists p \in P$ with $cpu(p) < cpu(p_{opt})$ and $mem(p) \leq M$.

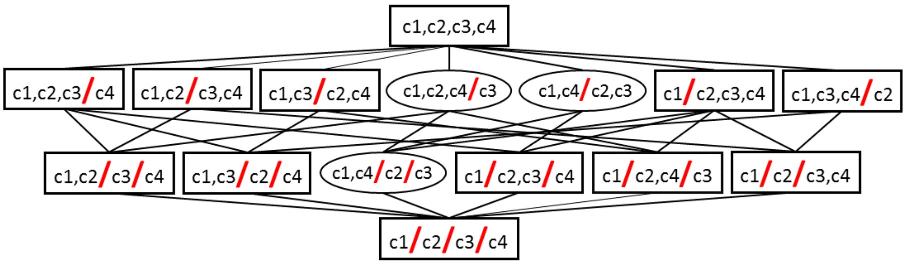


Figure 6.1: Search space of event-centric partitioning plans

6.1 CET-Graph Partitioning Search Space

Event-Centric Partitioning Plans. Figure 6.1 shows the search space of all partitioning plans for the CET graph in Figure 5.1. Each node in the search space is a partitioning plan. Events in different graphlets are separated by slashes. The number of graphlets per plan increases top to bottom in the search space. That is, the top node has only one graphlet with all events in it. The bottom node has as many graphlets as there are events, since each event belongs to a separate graphlet. Generally, the size of the search space is exponential, described by the Bell number, which represents the number of different partitions of a set of elements [40].

Time-Centric Partitioning Plans. Fortunately, we are not interested in all partitioning plans shown in Figure 6.1. We aim to partition a CET graph in such a way that we achieve the following goals: (1) ***Correct CET detection***: The H-CET algorithm can extract all CETs from the partitioned graph. (2) ***Expeditious CET detection***: Each final CET can be constructed after visiting each graphlet at most once. (3) ***Feasible memory requirement***: The memory cost of all graphlets satisfies the memory constraint.

Correct and Expeditious CET Detection. To achieve the first two goals, we consider only those partitioning plans that respect the order of events in a CET. We call such partitioning plans effective. A partitioning plan $p = \{g_1 = (V_1, E_1), \dots, g_k = (V_k, E_k)\}$ of a CET graph G is ***effective*** if for each CET tr in G that contains a pair of adjacent events v_1 and v_2 it holds that if $v_1 \in V_i, v_2 \in V_j$ then $i \leq j$. Effective partitioning plans are

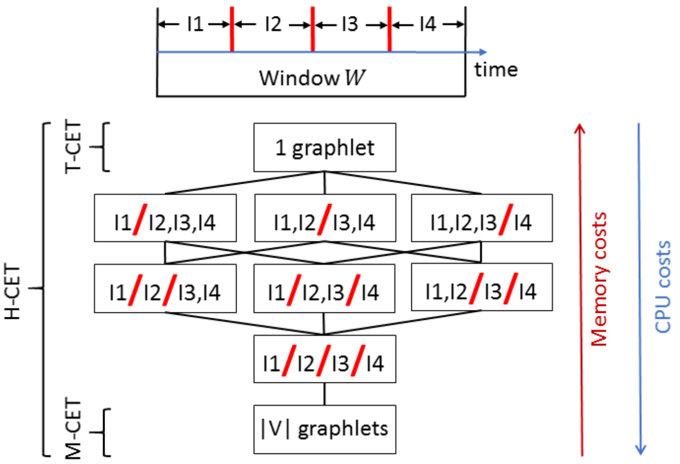


Figure 6.2: Search space of time-centric partitioning plans

indicated by rectangular frames in Figure 6.1 in contrast to ineffective plans with round frames. An ineffective partitioning plan requires visiting the same graphlet multiple times to construct one final CET. Such multiple graphlet accesses diminish the gain of sharing intermediate results and introduces CPU overhead as illustrated by Example 6.1.

Example 6.1 *Consider the partitioning into graphlets g_1 and g_2 with nodes $\{c_1, c_2\}$ and $\{c_3, c_4\}$ respectively. This plan is effective since the CET $tr_1 = (c_1, c_2)$ is extracted from g_1 and the CET $tr_2 = (c_1, c_3, c_4)$ results by concatenating c_1 from g_1 with (c_3, c_4) from g_2.*

In contrast, consider the partitioning into graphlets g'_1 and g'_2 with nodes $\{c_1, c_2, c_4\}$ and $\{c_3\}$ respectively. This plan is ineffective because the CET $tr_2 = (c_1, c_3, c_4)$ requires visiting g'_1 twice: Once to extract c_1 and once to get c_4.

We can exclude ineffective partitioning plans by traversing the CET graph to determine the order of events in all CETs. However, this process is expensive (Table 5.1). Instead, we partition the CET graph into non-overlapping consecutive time intervals. All events within a time interval are assigned to the same graphlet (similar to Chunking [56]). *Since consecutive time intervals contain consecutive sub-sequences of CETs, a time-centric partitioning plan is guaranteed to be effective.* The search space remains

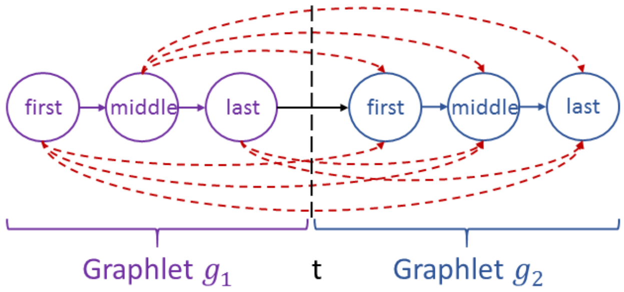


Figure 6.3: Cut of a CET graph

exponential, however, now in the number of *time intervals* (Figure 6.2), not in the number of *events* in the window (Figure 6.1). Such graph partitioning enables a substantial memory reduction for high-rate event streams in which multiple events fall into the same time interval.

Feasible Memory Requirement. To achieve the third goal, we differentiate between first, middle, and last events in a graphlet. An event e is called the ***last*** (***first***) in its graphlet g if there are no outgoing (incoming) edges from (to) e in the same graphlet g. An event that is neither first nor last is called a ***middle*** event. For example, events c_3 and c_4 are the last events in graphlet g_1 while events c_5 and c_6 are the first events in graphlet g_2 in Figure 5.2(c).

To reduce the memory consumption, we store *complete* event trends per graphlet, i.e., an event trend from a first to a last event in the graphlet. No intermediate event trends per graphlet are stored. To guarantee correctness, cut edges may connect only a first and a last event in their respective graphlets (Figure 6.3). The dashed edges are prohibited since they would require storing incomplete event trends per graphlet. To construct final results, for each edge from a last event e_l in a graphlet g_1 to a first event e_f in another graphlet g_2, we concatenate all CETs that end with e_l in g_1 with all CETs that start with e_f in g_2 (Figure 5.2(c)).

Definition 6.2 *(**Cut of a CET Graph**) A time point $t \in \mathbb{T}$ partitions a CET graph $G =$*

(V, E) into two graphlets g_1 and g_2 such that events are assigned to these graphlets by their time stamps. That is, $\forall e \in V$ if $e.time \leq t$ then $e \in g_1$. Otherwise $e \in g_2$. The time point t is called a cut of the graph G if the following conditions hold:

• If a last event in the graphlet g_1 has an outgoing edge to an event e, then e must be a first event in its graphlet g_2.

• If a first event in the graphlet g_2 has an ingoing edge from an event e, then e must be a last event in its graphlet g_1.

• If a middle event in the graphlet g_2 (g_1) has an ingoing (outgoing) edge from (to) an event e, then e must belong to the graphlet g_2 (g_1).

A graphlet is called atomic if it cannot be cut into smaller non-empty graphlets.

For example, a graphlet is atomic if all events in it have the same time stamp. Otherwise, a graphlet is not-atomic and can be partitioned into smaller graphlets.

6.2 Cost Monotonicity Across Levels

We now reveal the cost monotonicity properties across different levels and within the same level of the search space. We use these properties in Section 7.1 to design the branch-and-bound algorithm that effectively prunes the search space to find an optimal CET-graph partitioning plan.

Figure 6.2 illustrates the cost variations across levels of the search space. There are two extreme cases represented by the top and the bottom nodes. In the top node, all events are in one graphlet. Thus, CET detection takes place only in this graphlet. In other words, H-CET coincides with T-CET. In the bottom node, each event is in a separate graphlet. Thus, CET detection takes place across graphlets only. That is, H-CET coincides with M-CET. To further study the cost variations across levels of the search space, we define the parent-child relationship between nodes.

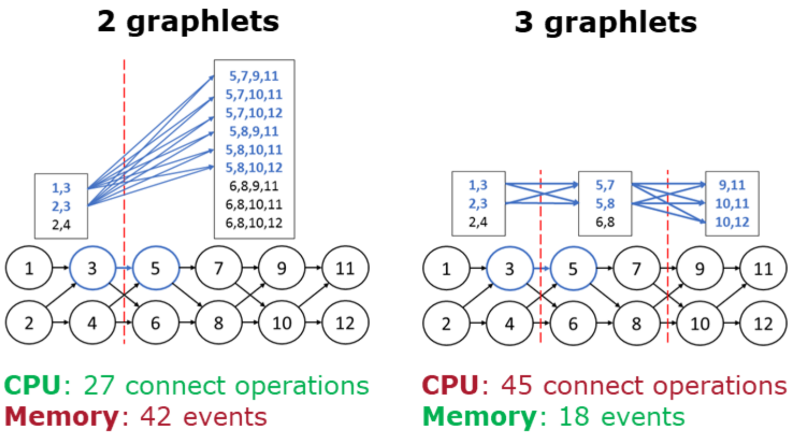


Figure 6.4: Cost monotonicity across levels

Definition 6.3 *(**Parent Partitioning Plan**) Let p be a partitioning plan of a CET graph G. Let p' be a partitioning plan of G that is the same as p except that a graphlet $g \in p$ is further partitioned into two graphlets $g_1, g_2 \in p'$. Then p is called a parent of p' and p' is called a child of p.*

Partitioning a graphlet g into two smaller graphlets g_1 and g_2 reduces the memory requirement because fewer and shorter CETs are stored in the smaller graphlets g_1 and g_2 than in the original graphlet g (Figure 6.4). In other words, memory costs monotonically decrease from parent to child in the search space (Theorem 6.1).

Theorem 6.1 (Memory Cost Monotonicity) *Let p and p' be partitioning plans of a CET graph G such that p is a parent of p'. The H-CET algorithm in G partitioned according to p' has the same or lower memory costs than in G partitioned according to p.*

Proof: Since the memory costs for storing the CET graph (Table 5.1) are independent of a partitioning plan, we ignore this cost factor in the following.

Let (v_1, v_2) be an edge that is not cut in the plan p, i.e., the vertices v_1 and v_2 belong to the same graphlet g in p. Assume this edge (v_1, v_2) is cut in the plan p', i.e., the vertices v_1 and v_2 belong to the graphlets g_1 and g_2 in p' respectively. According to Definition 6.2, v_1 is the last vertex in g_1 and v_2 is the first vertex in g_2.

Let y_1 be the number of CETs and n_1 be the number event occurrences in CETs[1] that end at the vertex v_1 in the graphlet g_1. Analogously, let y_2 be the number of CETs and n_2 be the number of event occurrences in CETs that start at v_2 in g_2. Then, the memory costs for storing CETs containing the vertices v_1 or v_2 in the graph G partitioned according to the plan p are computed as follows:

$$mem_p = y_1 * n_2 + y_2 * n_1.$$

In other words, all CETs in g_1 are replicated y_2 times and all CETs in g_2 are replicated y_1 times. In contrast, the memory costs for storing CETs containing v_1 or v_2 in G partitioned according to p' are

$$mem_{p'} = n_1 + n_2.$$

Obviously, $mem_p = mem_{p'}$ only in the special case when $y_1 = y_2 = 1$. However, $\forall n_1, y_1, n_2, y_2 \in \mathbb{N}, y_1, y_2 > 1.\ mem_p > mem_{p'}$. ■

Partitioning a graphlet g into two smaller graphlets g_1 and g_2 introduces additional CPU overhead because CETs in the smaller graphlets g_1 and g_2 have to be combined to final results (Figure 6.4). In other words, CPU cost monotonically increases from parent to child in the search space (Theorem 6.2).

Theorem 6.2 (CPU Cost Monotonicity) *Let p and p' be partitioning plans of a CET graph G such that p is a parent of p'. The H-CET algorithm in G partitioned according to p' has higher CPU costs than in G partitioned according to p.*

Proof: Since the CPU time for constructing the CET graph (Table 5.1) is independent from a partitioning plan, we ignore this cost factor in the following.

Let (v_1, v_2) be an edge that is not cut in the plan p, i.e., the vertices v_1 and v_2 belong to the same graphlet g in p. Assume this edge (v_1, v_2) is cut in the plan p', i.e., the vertices

[1] An event that occurs in k CETs it is counted k times.

v_1 and v_2 belong to the graphlets g_1 and g_2 in p' respectively. According to Definition 6.2, v_1 is the last vertex in g_1 and v_2 is the first vertex in g_2.

Let y_1 be the number of CETs that end at the vertex v_1 in the graphlet g_1 and y_2 be the number of CETs that start at v_2 in g_2. Then, the CPU time costs for connecting CETs containing the vertices v_1 or v_2 in the graph G partitioned according to the plan p' are computed as follows:

$$cpu_{p'} = y_1 * y_2.$$

In other words, each CET that ends at v_1 in g_1 is connected to each CET that starts at v_2 in g_2. In contrast, no such costs arise in G partitioned according to p. The edge (v_1, v_2) is traversed once. ■

6.3 Cost Monotonicity Within One Level

The number of partitioning plans at one level of the search space is exponential, described by the Stirling number that represents the number of ways to partition n elements into k partitions [40]. To restrict the search at one level, we differentiate between balanced, nearly balanced, and unbalanced graph partitioning plans and compare their costs.

A ***balanced partitioning plan*** divides a CET graph into graphlets such that the number of events in any two graphlets differs by at most 1. However, such a perfect partitioning plan is not always possible since an atomic graphlet cannot be divided (Definition 6.2). Thus, we define the notion of a ***nearly balanced partitioning plan*** that allows a graphlet to exceed the ideal size by less than the size of its first or last atomic graphlet. Otherwise a partitioning plan is ***unbalanced***.

Definition 6.4 ***(Balanced, Nearly Balanced and Unbalanced CET-Graph Partitioning Plans)*** *A partitioning plan p of a CET graph $G = (V, E)$ into k graphlets is called balanced if $\forall i \in \mathbb{N},\ 1 \leq i \leq k,\ |V_i| \leq \lceil \frac{|V|}{k} \rceil$ holds.*

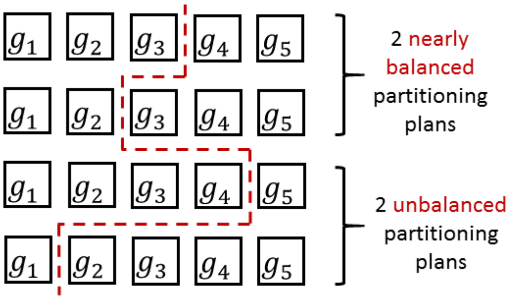

g_1 g_2 g_3 g_4 g_5
2 nearly balanced partitioning plans
2 unbalanced partitioning plans

Figure 6.5: Nearly balanced versus unbalanced partitioning plans

Let $g_i = (V_i, E_i)$ be a graphlet in a partitioning plan p such that $|V_i| > \lceil \frac{|V|}{k} \rceil$. Let $g_f = (V_f, E_f)$ be the first and $g_l = (V_l, E_l)$ be the last atomic graphlet in g_i. A partitioning plan p is called nearly balanced if two conditions hold:

- *If $i > 1$ then $|V_i| < \lceil \frac{|V|}{k} \rceil + |V_f|$ and*
- *If $i < k$ then $|V_i| < \lceil \frac{|V|}{k} \rceil + |V_l|$.*

Otherwise, a partitioning plan p is called unbalanced.

Example 6.2 *Assume a CET graph consists of atomic graphlets g_1–g_5 with equal number of events for the sake of simplicity (Figure 6.5). Assume we want to partition it into 2 graphlets. Obviously, no balanced partitioning plan exists since the atomic graphlet g_3 cannot be divided. However, 2 nearly balanced partitioning plans are possible. One of them assigns g_3 to the first graphlet and the other one assigns g_3 to the second graphlet. Note that the size of these nearly balanced graphlets exceeds the ideal size by less than the size of one atomic graphlet g_3. In contrast to that, if the graphlets are unbalanced, their size exceeds the ideal size by the size of more than one atomic graphlet.*

In the following, we will focus on homogeneous CET graphs (Definition 6.5). Cost monotonicity within one level of the search space for arbitrary graphs is subject for future work.

Definition 6.5 (Homogeneous CET Graph) *A CET graph is called homogeneous if the*

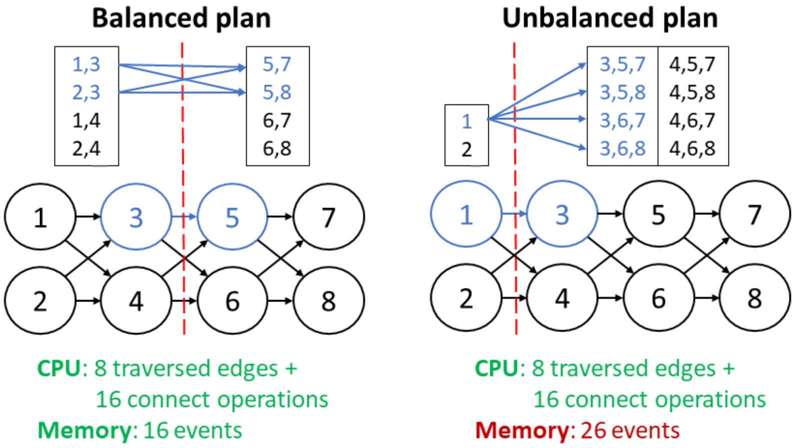


Figure 6.6: Cost monotonicity within one level

following two conditions hold: (1) All atomic graphlets in it have the same number of events. (2) The number of ingoing and outgoing edges of events is the same.

For example, the CET graph in Figure 6.6 is homogeneous. Each atomic graphlet contains two events. Each event has two ingoing and outgoing edges, except START and END events which do not have ingoing and outgoing edges respectively by definition (Section 5.1).

The closer a partitioned homogeneous CET graph is to balanced, the lower the memory costs of CET detection are. Intuitively, if a partitioned graph is unbalanced, many long CETs are stored in its large graphlets. Hence, the memory costs of large graphlets dominate the overall memory usage. In contrast, a balanced partitioning plan has the minimal memory costs among all partitioning plans with the same number of graphlets while CPU costs are the same for balanced, nearly balanced, or unbalanced partitioning plans with the same number of graphlets (Theorem 6.3).

Theorem 6.3 (Costs of Balanced Partitioning Plan) *If a balanced graph partitioning plan p_b with k graphlets exists for a homogeneous CET graph G, the memory costs of CET detection in G partitioned according to p_b are minimal compared to all other plans* *with k graphlets. Otherwise, a nearly balanced plan p_{nb} has this property. CPU costs are the same for all partitioning plans of G with the same number of graphlets k.*

Proof: Since the CPU and memory cost for CET graph construction and storage (Table 5.1) are independent from the quality of a partitioning plan, we ignore these costs below.

Let a be the number of atomic graphlets in G. Then, $\frac{a}{k}$ atomic graphlets belong to one graphlet. We assume that a is divisible by k, i.e., a balanced partitioning plan p_b of G exists. We assume that $k = 2$ below. The proof for arbitrary $k \in \mathbb{N}$ is analogous. Let y be the number of CETs in an atomic graphlet that are combined with CETs in other graphlets to form final results. Let n be the number of event occurrences in CETs per atomic graphlet. We now prove the first statement above by induction over a.

Induction begin: $a = 4$. Then, the memory costs of CET detection in G partitioned according to a balanced plan p_b is $mem_b = 2yn + 2yn = 4yn$. Namely, each of 2 graphlets stores 2 atomic graphlets. Each of atomic graphlets replicates n events in CETs y times. In contrast, the memory costs of CET detection in G partitioned according to an unbalanced plan p_{ub} is $mem_{ub} = y + 3 * 2yn = y + 6yn$. Namely, one graphlet stores one atomic graphlet and another graphlet contains 3 atomic graphlets. In the larger graphlet, n events in CETs are replicated $2y$ times. Obviously, $mem_b < mem_{ub}$.

Induction assumption: We assume that the first statement is true for any number $a \in \mathbb{N}$ of atomic graphlets.

Induction step: $a \to a + 2$. Let $x = \frac{a}{2}$ for readability. Then, the memory costs of CET detection in G partitioned according to a balanced plan p_b is $mem_b = 2(x + 1)ynx$. Namely, each of 2 graphlets contains $(x + 1)$ atomic graphlets. n events in CETs are replicated yx times in each graphlet. In contrast, the memory costs of CET detection in G partitioned according to an unbalanced plan p_{ub} is $mem_{ub} = xyn(x-1)+(x+2)yn(x+1)$. Namely, one graphlet stores x atomic graphlets and another graphlet contains $(x + 2)$

atomic graphlets.

$mem_b < mem_{ub}$

$2(x+1)ynx < xyn(x-1) + (x+2)yn(x+1)$

$2(x+1)x < x(x-1) + (x+2)(x+1)$

$2x^2 + x < x^2 - x + x^2 + x + 2x + 2$

$2x^2 + x < 2x^2 + 2x + 2$

The first statement holds for other unbalanced plans. If a is not divisible by k, then balanced plan does not exist. The proof of second statement is analogous to above.

Since the number of vertices and edges is the same for all atomic graphlets in a homogeneous graph, the number of traversed edges within graphlets and the number of connect operations across graphlets are the same for all partitioning plans with k graphlets. Thus, the last statement holds. ■

7

CET Detection Optimization

7.1 Branch-and-Bound Partitioning Algorithm

As explained in Section 6.1, the search space of partitioning plans is exponential in the number of atomic graphlets. Thus, we now utilize the cost monotonicity properties identified in Sections 6.2 and 6.3 to define an efficient branch-and-bound CET-graph partitioning algorithm (B&B). This algorithm works for homogeneous CET graphs (Definition 6.5).

The search space is depicted in Figure 6.2. Each node in it is a CET-graph partitioning plan that is connected to its parents and children (Definition 23.10). Our B&B algorithm (Algorithm 5) traverses the search space top down. Since memory costs decrease and CPU costs increase from parent to child in the search space, B&B prunes top levels with too high memory costs and branches with too high CPU costs (Figure 7.1). In addition, the algorithm disregards unbalanced partitioning plans at each level. B&B consists of the following two phases:

1. Level Search Algorithm. Since the memory requirement is high at the top levels, we skip infeasible levels that are guaranteed to contain no partitioning plan that fits into

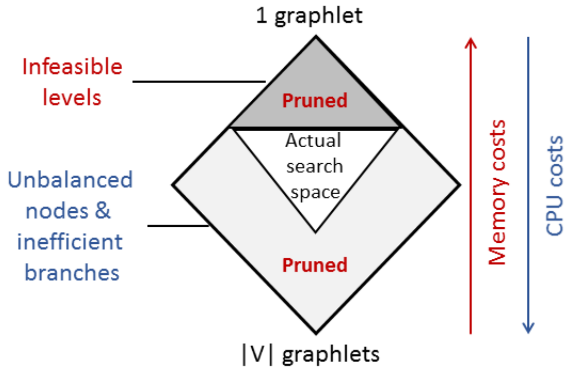


Figure 7.1: Search space for an optimal CET graph partitioning

memory. To achieve this goal, the auxiliary method *atomicGraphletNumber(G)* returns the number of automic graphlets n the graph G. For each possible number of graphlets i, the auxiliary method *balanced(G, i)* computes a hypothetical balanced partitioning plan b of the input CET graph G at level i. The algorithm then compares the estimated memory cost of this balanced plan b to the memory limit M. The memory cost of this balanced plan b with i graphlets is the lower bound of the actual minimal memory cost at level i (Theorem 6.3).

Infeasible level pruning principle: If a balanced partitioning plan with i graphlets does not fit into memory, then no other partitioning plan with i graphlets will. Thus, level i can be safely purged.

The algorithm returns the highest level that satisfies the memory constraint M with i balanced graphlets (Lines 1–9).

2. Node Search Algorithm. Knowing the minimal number i of required graphlets, this algorithm aims to find an optimal partitioning plan with at least i graphlets. To this end, the algorithm keeps track of the (nearly) balanced nodes to consider in a *heap*. It disregards all other (unbalanced) partitioning plans since according to Theorem 6.3 they are less efficient.

Unbalanced node pruning principle: If a (nearly) balanced partitioning plan with i

Algorithm 5 Branch-and-bound CET graph partitioning algorithm

```
Input: Graph $G$, Memory limit $M$, Heap $heap$, List $pruned$
Output: Node $solution$
 1: $s = 1, e = atomicGraphletNumber(G)$                          // Level search
 2: while $s \leq e$ do
 3:     $i = s + (e - s)/2$
 4:     Node $b = balanced(G, i)$
 5:     if $mem(b) \leq M$ then
 6:         $level = i$
 7:         $e = i - 1$
 8:     else
 9:         $s = i + 1$
10: $minCPU =+ \infty$                                              // Node search
11: Node[ ] $nodes = nearlyBalanced(G, level, pruned)$
12: $PushAll(heap, nodes)$
13: while $!isEmpty(heap)$ do
14:     $temp = Pop(heap)$
15:     if $temp.isPruned(pruned)$ then
16:         continue
17:     if $memory(temp) < M$ then
18:         if $cpu(temp) < minCPU$ then
19:             $solution = temp$
20:             $minCPU = cpu(temp)$
21:         $pruned.add(temp.cuts)$
22:     if $temp.level = level$ then
23:         $level$++
24:         $nodes = nearlyBalanced(G, level, pruned)$
25:         $PushAll(heap, nodes)$
26: return $solution$
```

graphlets exists at a level i, then any other (unbalanced) partitioning plans with i graphlets will have higher CPU and memory costs. Thus, all other nodes at level i can be safely purged.

Starting from the highest feasible level i, in every step the algorithm reduces the memory utilization (Theorem 6.1) at the expense of increased CPU time (Theorem 6.2). As soon as a feasible solution is found, we can disregard the descendants of this feasible node since these descendants will have higher CPU costs than their ancestor (Lines 11, 15–16, 21, 24).

Inefficient branch pruning principle: If a graph partitioning plan p that satisfies the memory constraint is found, then p's descendants will have higher CPU cost than p. Thus, they can be safely purged.

Since CPU cost increases from parent to child (not from level i to level $i-1$), it is possible that a node at level i has lower CPU than a node at level $i-1$. Thus, the algorithm considers all nearly balanced not-pruned nodes. They are computed by the auxiliary method *nearlyBalanced(G,level,pruned)*. These nodes are shown as white area in Figure 7.1. The figure conveys that the number of such nodes decreases top down. Node search terminates when it either reaches the lowest level or a level at which all nearly balanced nodes are pruned.

All descendants of a node n will have the cuts that are present in the node n (Figure 6.2). The cuts of feasible nodes are maintained in the list *pruned*. In Lines 11 and 24, we avoid generating nearly balanced nodes from pruned branches. In Lines 15–16, we disregard pruned nodes since a node may be pruned after it is added to the *heap*.

B&B keeps track of the best partitioning plan found so far (Lines 18–20) and returns it at the end of the node search (Line 26). H-CET in the graph partitioned by this plan is guaranteed to be optimal, i.e., have the minimal CPU processing time and stay within the memory limit (Definition 6.1).

Theorem 7.1 (Partitioning Plan Optimality.) *B&B returns an optimal CET-graph partitioning plan.*

Theorem 7.1 follows from Theorems 6.1–6.3. Details are omitted here to keep the discussion brief.

Complexity Analysis. Let k be the number of levels in the search space, i.e., the number of atomic graphlets in the input CET graph. Then, level search has logarithmic

CPU cost in k. It utilizes binary search and computes the memory cost of exactly one perfectly balanced partitioning plan at each level. Thus, the CPU time is $O(log\ k)$.

The node search algorithm has exponential CPU cost in k. The while-loop in Lines 13–25 is called at most for every nearly balanced node in the search space, i.e., $O(2^k)$ times. Computing all nearly balanced not-pruned partitioning plans at a level i in Lines 11 and 24 has exponential CPU cost $O(2^i)$. Checking whether a node is pruned in Line 15 is also exponential. All other auxiliary methods have constant CPU complexity. Thus, the CPU cost is $O(2^k)$.

The memory cost of level search is constant. In contrast, the memory cost of node search is exponential in k. It is determined by the size of the list *pruned* and the *heap*. Both are bounded by the maximal number of nearly balanced nodes in one level of the search space, namely, $O(2^k)$.

Despite three pruning principles, the CPU and memory costs of B&B are exponential in the number of atomic graphlets in the worst case.

7.2 Greedy Partitioning Algorithm

In this section, we thus propose a greedy search algorithm for high efficiency. It reuses the level search strategy from Algorithm 5 without change while node search is simplified. Namely, the greedy search considers only one greedily constructed, nearly balanced partitioning plan at level i. If it satisfies the memory constraint, the algorithm returns it as a result. Otherwise, a nearly balanced partitioning plan at level $i - 1$ is considered. Thus, the result of the greedy search is guaranteed to be a nearly balanced partitioning plan. However, this result might not be optimal since it might have more graphlets than necessary to satisfy the memory constraint. In other words, the greedy search algorithm does not utilize the entire available memory to speed up CET detection.

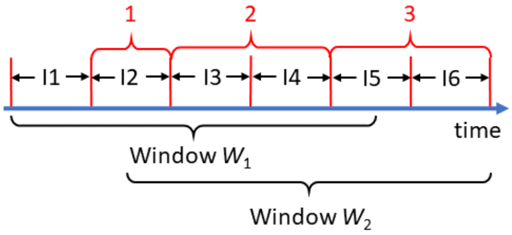


Figure 7.2: Graphlet sharing technique

Complexity Analysis. Since the greedy search algorithm considers only one (nearly) balanced partitioning plan at a level, its memory cost is constant and its CPU complexity is linear $O(k)$ in the number of atomic graphlets k.

7.3 Graphlet Sharing Across Windows

Our CET executor effectively shares graphlets between overlapping sliding windows. When the window slides, graphlets are categorized into the following three groups (Figure 7.2): (1) Partially expired graphlet, (2) Shared graphlets among previous and new window, and (3) (Partially) new graphlets. Since graphlets store consecutive stream portions, there can be at most one partially expired graphlet ($I2$ in Figure 7.2) and at most one partially new graphlet ($I5$). There can be however any number of (completely) expired, shared, or new graphlets. The shared graphlets can be reused across overlapping windows such that repeated CET graph partitioning, construction, traversal, and CET detection in the shared graphlets can be avoided. The construction of partially expired graphlets can be shared between several windows by ignoring expired events. A partially new graphlet must be updated by new events.

Analogously to sharing graphlets between overlapping windows of the *same CET query*, the execution of a workload of *multiple CET queries* can be optimized by sharing the processing of common graphlets across sub-queries. However, sharing graphlets may not always be beneficial due to different window parameters or predicates associated with each CET query. Hence, multi-CET-query optimization is left as subject for future research.

8

Performance Evaluation

8.1 Experimental Setup and Methodology

Experimental Infrastructure. We have implemented our CET detection approach in Java with JRE 1.7.0_25. We run our experiments on a cluster machine with Ubuntu 14.04, 16-core 3.4GHz QEMU Virtual CPU, and 128GB of RAM. We execute each experiment three times and report their average results here.

Data Sets. We evaluate our CET approach using the following data sets.

• ***PA: Physical Activity Real Data Set***. The physical activity monitoring real data set [42] (1.6GB) contains physical activity reports for 14 people during 1 hour 15 minutes. There are 20 physical activities which can be classified into active (e.g., running, soccer playing) and passive (e.g., watching TV, working on computer). Other attributes are time stamp, heart rate, temperature and person identifier.

• ***ST: Stock Real Data Set***. The NYSE data set [43] contains 225k transaction records of 10 companies in 1 sector during 12 hours. Each event carries company, sector, and transaction identifiers, volume, price, time stamp, and type (sell or buy). We replicate this data set 10 times with adjusted company, sector, and transaction identifiers such that the resulting data set contains transactions for 110 companies in 11 sectors. No other attributes except identifiers were changed in the replicas compared to the original.

- ***FT: Financial Transaction Data Set***. We developed an event stream generator that creates check deposit events for 3 days. Each event carries time stamp in seconds, source bank, destination bank, and status (covered or not). The generator allows us to vary event rate per second R, event compatibility C, and predicate selectivity S. Given these 3 parameters, each of $R * S$ events with time stamp t is compatible with C events with earlier time stamps than t. Unless stated otherwise, the default event rate is 3.5k events per second and the default event compatibility is 3.

CET Queries. We evaluate the workload of 10 appropriate CET queries against each of these event streams. These queries are variations of queries Q_1, Q_2, and Q_3 in [5]. They differ by event patterns and predicates.

Methodology. To show the efficiency of our CET approach, we compare it to the baseline algorithm (BL, Chapter 4) and SASE [3] because both approaches support Kleene closure computation over event streams under the skip-till-any-match semantics. To achieve fair comparison, we have implemented SASE [3] including its optimization techniques [7, 8] on top of our platform. In a nutshell, SASE stores each matched event e in a stack and computes pointers to e's previous events in a CET. At the end of each window, DFS-based CET extraction is applied to these stacks. Chapter 9 contains a more detailed discussion.

We also compare our approach to the open-source streaming system Apache Flink [30] that supports event pattern matching under skip-till-any-match. We express our CET queries using Flink operators. Flink guarantees that events are processed in parallel but in-order by their time stamps. Each event is processed exactly once. However, like most industrial streaming systems [31, 41, 57], the Kleene operator is not explicitly supported by Flink. It is currently being developed for the next release [45]. To express our CET queries in Flink, each CET query q with a Kleene operator is flattened into a set of event sequence queries that cover all possible lengths of event trends matched by the query q. We vary the event rate, event compatibility, window length, and window overlap size in these experiments.

To demonstrate the effectiveness of our branch-and-bound graph partitioning (B&B, Section 7.1), we compare it to the exhaustive (Exh) and greedy search (Greedy, Section 7.2) by varying the event rate, window length, memory limit, and event compatibility.

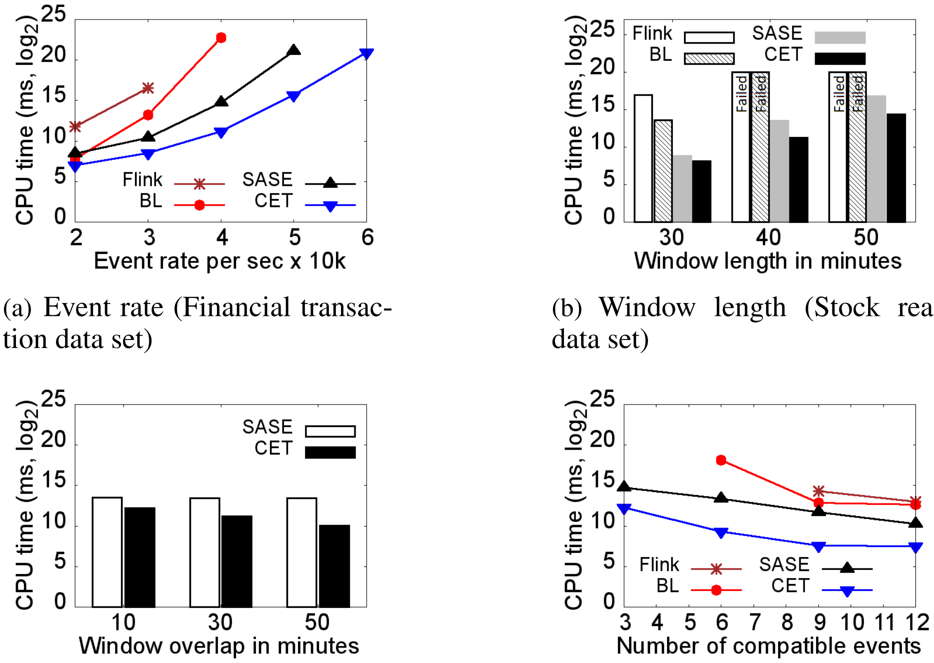
Under certain parameter settings, *Flink, BL, SASE*, and *Exh* are unable to produce results within several hours. Thus, the results are either discontinued in the line charts or highlighted by bars with maximal height in the bar charts.

Since event rate and event compatibility cannot be varied for the real data sets, we ran these experiments on the FT data set. We vary window length and window overlap size on the PA and ST real data sets.

Metrics. We measure two metrics common for streaming systems, namely, average *CPU time* and *memory requirement* per window [3, 4, 7, 38, 58]. Average CPU time is measured in milliseconds as the sum of total elapsed time in all windows divided by the number of windows. Average memory requirement is measured as the sum of peak memory costs in all windows divided by the number of windows. For CET (or SASE), memory costs include all stored CET, all vertices and edges in a CET graph (or all events and pointers between them in a SASE stack [3]). For Baseline and Flink, all stored CETs contribute to memory costs. For CET graph partitioning algorithms, all nodes in the search space that are stored at a time contribute to memory costs.

8.2 CET Detection Algorithms

In Figures 8.1–8.2, we compare our CET approach to other CET detection solutions.



(a) Event rate (Financial transaction data set)

(b) Window length (Stock real data set)

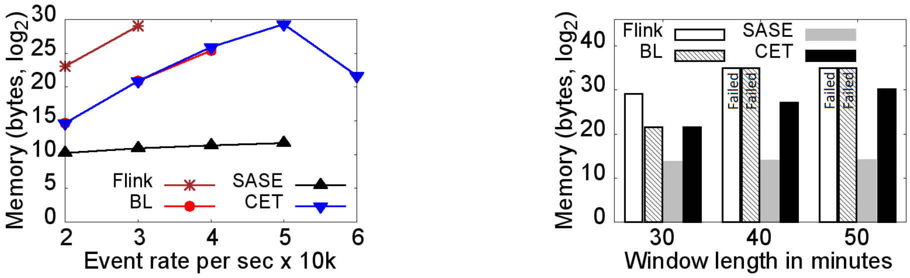
(c) Window overlap (Stock real data set)

(d) Event compatibility (Financial transaction data set)

Figure 8.1: CPU costs of CET detection algorithms

Varying Number of Events per Window. CPU costs of all approaches grow exponentially with an increasing stream rate and window length (Figures 8.1(a)–8.1(b)). For large numbers of events per window, Flink, BL, and SASE do not terminate within several hours. These approaches do not scale because they construct all CETs from scratch – provoking repeated computations. In contrast, our CET approach stores partial CETs and reuses them while constructing final CETs. Thus, CET is 42–fold faster than SASE at a stream rate of 50k events per second and two orders of magnitude faster than Flink at a stream rate of 3k events per second. Flink is even slower than BL in all experiments because we express each Kleene query as a set of event sequence queries in Flink. Thus, the workload of Flink is considerably higher than other approaches tailored for Kleene computation.

Figure 8.2 demonstrates that the CET approach adapts to the available memory. That



(a) Event rate (Financial transaction data set)

(b) Window length (Stock real data set)

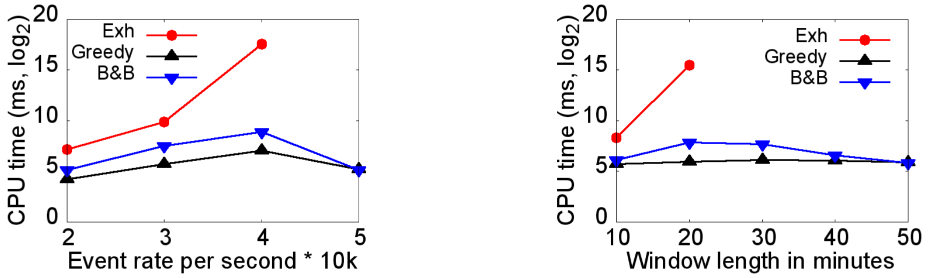
Figure 8.2: Memory costs of CET detection algorithms

is, it partitions the graph in such a way that CET detection in the partitioned graph is guaranteed to stay within the given memory limit. If the stream rate is below 50k events per second or the window is shorter than 50 minutes, H-CET coincides with T-CET (Section 5.3). Otherwise, T-CET would run out of memory since it stores CETs for the *whole graph*. Instead, our CET approach partitions the graph into smaller graphlets and stores CETs *per each graphlet*.

CET and BL have almost the same memory cost (Figure 8.2). Thus, the amount of memory required for the CET graph is negligible compared to the amount of memory required for CET storage. We thus conclude that CET graph is worth maintaining to compactly capture all CETs.

SASE requires 5 orders of magnitude less memory than the CET approach if event rate is 50k events per second since SASE stores events in stacks and pointers to their previous events. When the window ends, these pointers are traversed using DFS to extract all CETs. Hence, SASE is conceptually close to the M-CET algorithm, namely, it is lightweight but slow (Figures 8.1 and 8.2).

Flink requires two orders of magnitude more memory than CET at a stream rate of 3k events per second, because Flink stores all trends of all possible lengths, since it expresses Kleene closure by a set of sequence queries.



(a) Event rate (Financial transaction data set)

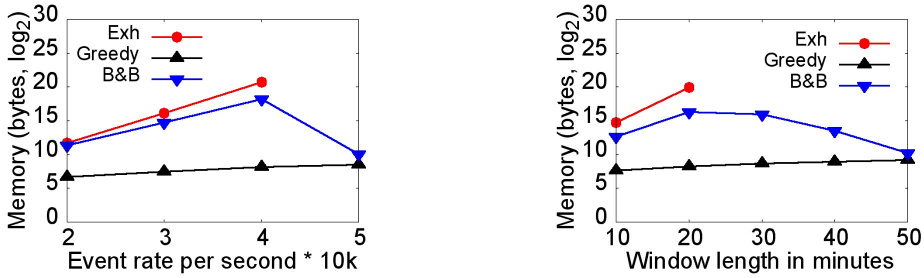
(b) Window length (Physical activity monitoring real data set)

Figure 8.3: CPU costs of CET graph partitioning algorithms

We also ran the experiment on the PA data set varying window lengths. These charts have similar trends as in Figures 8.1(b) and 8.2(b). Thus, they are omitted here.

Varying Window Overlap. Figure 8.1(c) measures the effect of the graphlet sharing technique. As the window overlap increases, the CPU time of the CET approach decreases exponentially. This performance is explained by the fact that CET graph construction, partitioning and CET detection within graphlets is shared between overlapping windows. The larger the overlap, the more the gain of sharing. In contrast, the average CPU time per window of SASE remains fairly constant, since no intermediate results are shared among overlapping windows. Our CET approach outperforms SASE 13–fold if the window overlap is 50 minutes. Since Flink and BL do not keep up with such long windows, they are omitted in Figure 8.1(c).

Varying Event Compatibility. Figure 8.1(d) experimentally confirms Theorem 3.1. Namely, the number of CETs (and thus the cost of CET detection) is maximal if event compatibility is 3. When event compatibility increases, the CPU time of all algorithms decreases exponentially. When event compatibility is 3, our CET approach is up to 5.4–fold faster than SASE since it avoids repeated computations by caching and reusing intermediate results. Neither Flink nor BL can produce any results in this worst case scenario.



(a) Event rate (Financial transaction data set)

(b) Window length (Physical activity monitoring real data set)

Figure 8.4: Memory costs of CET graph partitioning algorithms

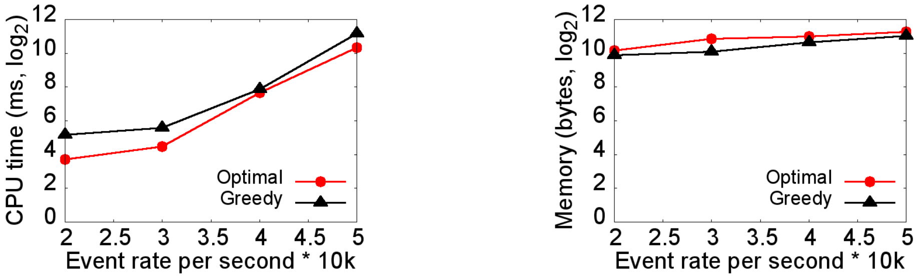
8.3 CET-Graph Partitioning Algorithms

Varying the Number of Events per Window. In Figures 8.3–8.4, we compare the performance of the CET graph partitioning algorithms used by our optimizer.

The search space for an optimal CET graph partitioning plan has a lattice shape (Figure 6.2). Thus, the B&B optimizer performs best if it searches the top or the bottom of the search space where the number of nodes is relatively small. The top or the bottom of the search space is traversed if the memory constraint is loose or tight for the given number of events (first and last cases on the X-axis). Otherwise, B&B searches the middle of the search space where the number of nodes is large. This search causes higher CPU and memory costs (middle cases on the X-axis).

The CPU and memory costs of the exhaustive algorithm grow exponentially with an increasing event rate or window length. Due to three effective pruning principles, our B&B is up to two orders of magnitude faster at a stream rate of 40k events per second and requires up to 12–fold less memory when window length is 20 minutes than the exhaustive optimizer. Yet B&B returns an optimal CET graph partitioning plan.

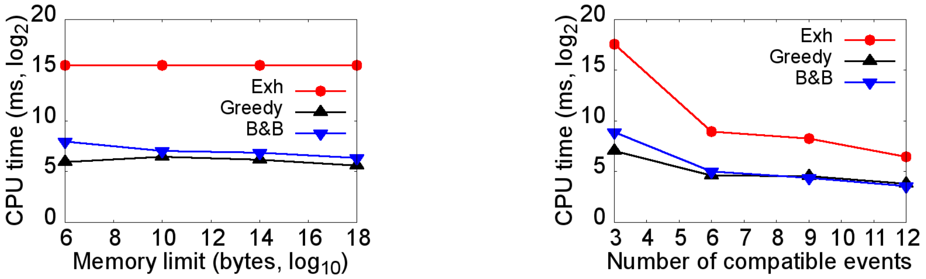
The greedy optimizer has fairly constant CPU and memory costs. It is up to three times faster and requires up to three orders of magnitude less memory than B&B if the stream rate is 40k events per second.



(a) CPU time

(b) Memory costs

Figure 8.5: Partitioned graph quality (Financial transaction data set)



(a) Memory limit (Physical activity monitoring real data set)

(b) Event compatibility (Financial transaction data set)

Figure 8.6: CPU costs of CET graph partitioning algorithms

Partitioned Graph Quality. The greedy optimizer tends to return a sub-optimal CET graph partitioning plan because it considers only one nearly balanced partitioning plan per level until it finds a plan that satisfies the memory constraint. Thus, greedy search tends to partition a CET graph more than necessary. In other words, it does not utilize the entire memory resources to speed up CET detection. Figure 8.5 compares the CPU and memory costs of CET detection in an optimally-partitioned versus a greedily-partitioned CET graph. A greedy partitioning plan is 2.8–fold slower than an optimal partitioning solution at an event rate of 20k events per second.

Varying Memory Limit. As Figure 8.6(a) illustrates, the exhaustive optimizer is indifferent to the memory limit. It traverses the entire search space to find an optimal partitioning plan. In contrast, both B&B and greedy algorithms utilize the memory constraint to skip infeasible levels. Thus, B&B is two orders of magnitude faster and requires 12–fold less memory than the exhaustive optimizer for a tight memory limit of 3MB.

When the memory constraint is loose, a few cuts are enough to satisfy the memory constraint. Thus, the B&B and greedy algorithms perform almost equally well. However, when the memory constraint is tight, a CET graph has to be partitioned more to avoid an out-of-memory error. While our B&B optimizer considers all nearly balanced partitioning plans at a level to find an optimal partitioning plan, the greedy optimizer considers only one per level. Thus, the greedy optimizer is up to four times faster than B&B for a tight memory limit of 3MB, at the expense of finding a sub-optimal partitioning plan.

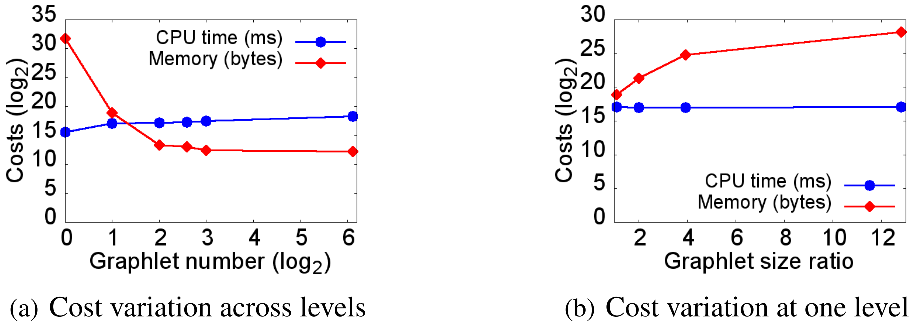
Varying Event Compatibility. With the increasing number of compatible events, the size of graphlets increases and their number decreases (Figure 8.6(b)). Thus, the size of the search space also decreases. Consequently, the CPU time of the algorithms rapidly decreases as well until it becomes almost constant if more than 10 events are compatible. If three events are compatible, our B&B optimizer is two orders of magnitude faster than the exhaustive optimizer thanks to our effective pruning principles.

8.4 Properties of Partitioning Search Space

In Figure 8.7, we experimentally confirm the cost monotonicity properties of the CET graph partitioning search space (Sections 6.2 and 6.3) while varying the number of graphlets and their size ratio.

Varying Graphlet Number. The goal of this experiment is to confirm the properties across levels of the search space (Theorems 6.1 and 6.2) which support the infeasible level and inefficient branch pruning by our B&B optimizer.

In Figure 8.7(a), we observe monotonic CPU and memory cost variation while varying the number of graphlets. The first case on the X-axis (one graphlet) corresponds to T-CET



(a) Cost variation across levels

(b) Cost variation at one level

Figure 8.7: Search space properties (Physical activity monitoring real data set)

since all events belong to one graphlet. The last case on the X-axis (69 graphlets) corresponds to M-CET, since each relevant event is a separate graphlet. In all other cases, H-CET is applied to an optimally-partitioned CET graph into the given number of graphlets.

As the number of graphlets increases, the memory requirement decreases exponentially (5 orders of magnitude comparing 1 and 69 graphlets) and CPU time increases exponentially (7–fold comparing 1 and 69 graphlets). Such cost variation is expected since the larger the number of graphlets, the smaller their size, the fewer CETs are stored per graphlet and the more CPU overhead is provoked while computing CETs across graphlets.

Varying Graphlet Size Ratio. The goal of this experiment is to confirm the properties at one level of the search space (Theorem 6.3) which lead to the unbalanced node pruning by our B&B optimizer.

In Figure 8.7(b), we observe the monotonic cost variation while varying the graphlet size ratio. The first case on the X-axis (the ratio is 1) corresponds to CET detection in a CET graph that is partitioned in a balanced way. The last case on the X-axis (the ratio is 13) shows the costs of CET detection in a CET graph that is partitioned in an unbalanced way, namely, one graphlet has 13 times more events than another.

As the graphlet size ratio increases, the memory requirement increases exponentially (two orders of magnitude comparing the balanced and the unbalanced partitioning plans). Such memory increase is expected since the larger the size of graphlets the more CETs are stored in them. In contrast, CPU time stays stable since the same number of graphlets causes the same CPU costs for CET construction within and across graphlets.

9

Related Work

Complex Event Processing. Most existing approaches either compute event sequences of a fixed length [4, 38, 58] or assume a known upper bound on the length of event sequences [59]. They do not support event queries with Kleene patterns. Thus, their expressive power is limited.

Several approaches [3, 7, 8, 28, 29] support Kleene closure computation over event streams. However, ZStream [29] and Cayuga [28] do not support Kleene closure computation under the skip-till-any-match semantics. In contrast, SASE [7, 8] optimizes Kleene queries under various semantics. SASE adapts a Finite State Automaton (FSA) based execution paradigm. Each event query is translated into an FSA. Each state of an FSA is associated with a stack with single events stored in each stack. To speed up stack traversal, each event is augmented with a pointer to its previous event in a match.

SASE++ [3] further optimizes query evaluation by breaking it into pattern matching and result construction phases. Pattern matching computes the main runs of an FSA (equivalent to CETs) with certain predicates postponed. Result construction derives all matches of the Kleene pattern by applying the postponed predicates to remove non-viable runs. The pattern matching phase in SASE++ simply reuses the FSA-based approach from SASE [7, 8] without any optimization. Computing matches for the main runs incurs repeated computations since the same common event sub-sequence is re-traversed for each match that contains it. In contrast, we introduce a compact CET graph to capture all CETs and partition the graph to trade off between CPU and memory costs of CET detection.

Recursive Queries. Kleene closure computation has been studied in recursive query processing as well [20, 21, 22, 23, 24, 25, 26, 27]. However, most existing solutions either focus on achieving high expressive power for recursive queries [26] or ensuring correctness of such queries [20, 23]. These approaches incur additional execution costs for supporting more expressive queries than CEP queries. The optimization techniques proposed in [21, 22, 24, 25] neither support the skip-till-any-match semantics nor take memory constraints into consideration. Therefore, none of the existing solutions fully tackles the challenges of event queries with Kleene closure we target.

Static Databases. Traditional SQL queries [15, 16, 17] do not support streaming operators such as event sequence, Kleene closure, and windows that treat the order of events by their time stamps as first-class citizen. While static sequence data bases extend traditional SQL queries by order-aware join operators [18, 19], they still do not support Kleene closure computation under the skip-till-any-match semantics. Worse yet, these approaches assume that the data is statically stored and indexed prior to processing. Hence, they do not tackle challenges that arise due to dynamically streaming data such as event expiration and real-time execution.

Streaming Graph Partitioning approaches [56, 60, 61] consider dynamic graphs in which vertexes or edges or both change over time. These approaches aim at a balanced one-pass dynamic graph partitioning algorithm. Since the problem is NP-hard [62], several approximation algorithms [62, 63] and heuristics [64, 65, 66] have been developed.

While these strategies also aim to find a balanced graph partitioning, they focus on optimization criteria that are distinct from our problem. Namely, they minimize the total number [56, 61, 65, 66] or weight [62] of cut edges. If we were to apply their method and minimize the number of cut edges, it would not necessarily reduce the cost of CET detection since one cut edge may need to be traversed multiple times. Potentially, we could address this problem by defining an edge weight as the number of edge traversals. However, CETs within and across graphlets would have to be computed to determine the number of edge traversals. This risks making our CET-graph partitioning algorithm more expensive than the CET detection algorithm itself.

Furthermore, existing graph partitioning approaches do not take the order of events in a CET into consideration – which is crucial to CET detection. Indeed, if a graph partitioning algorithm is oblivious to the event order, then correct CET detection that visits each graphlet at most once for each CET would not be possible (Section 6.1). Also, graphlets could not be shared between overlapping sliding windows (Section 7.3). Lastly, the properties of the graph partitioning search space we uncover are specific to the cost model for CET detection, and thus lead us to unique pruning principles of our B&B algorithm.

Part II

Event Trend Aggregation

10

GRETA Approach Overview

In Chapters 10–13, we focus on the aggregation of event trends matched by nested Kleene patterns (Definition 2.1) with expressive predicates under the most flexible skip-till-any-match semantics (Section 2.4). Other semantics are considered in Chapters 14–19.

Our ***Event Trend Aggregation Problem*** to compute event trend aggregation results of a query q against an event stream I with *minimal latency*. Figure 10.1 provides an overview of our GRETA framework. The **GRETA *Query Analyzer*** statically encodes the query into a GRETA configuration. In particular, the pattern is split into positive and negative sub-patterns (Section 2.3). Each sub-pattern is translated into a GRETA template (Section 2.2). Predicates are classified into vertex and edge predicates (Section 11.3).

Guided by the GRETA configuration, the **GRETA *Runtime*** first filters and partitions the stream based on the vertex predicates and grouping attributes of the query. Then, the GRETA runtime encodes matched event trends into a GRETA graph. During the graph construction, aggregates are propagated along the edges of the graph in a dynamic programming fashion. The final aggregate is updated incrementally, and thus is returned immediately at the end of each window (Sections 11.1–11.3).

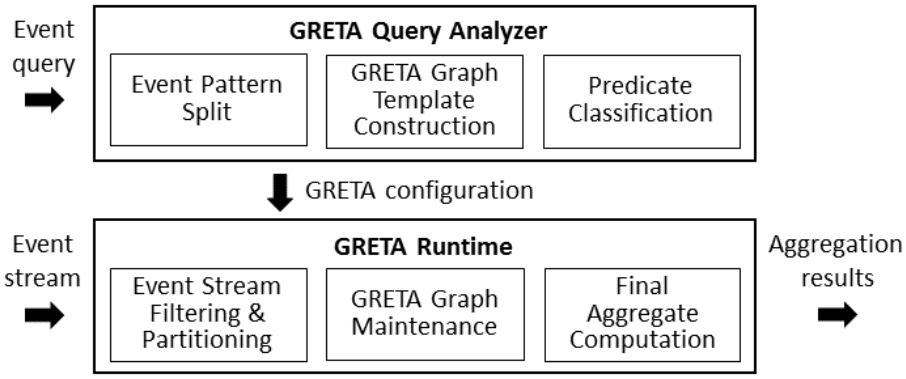


Figure 10.1: GRETA framework

11

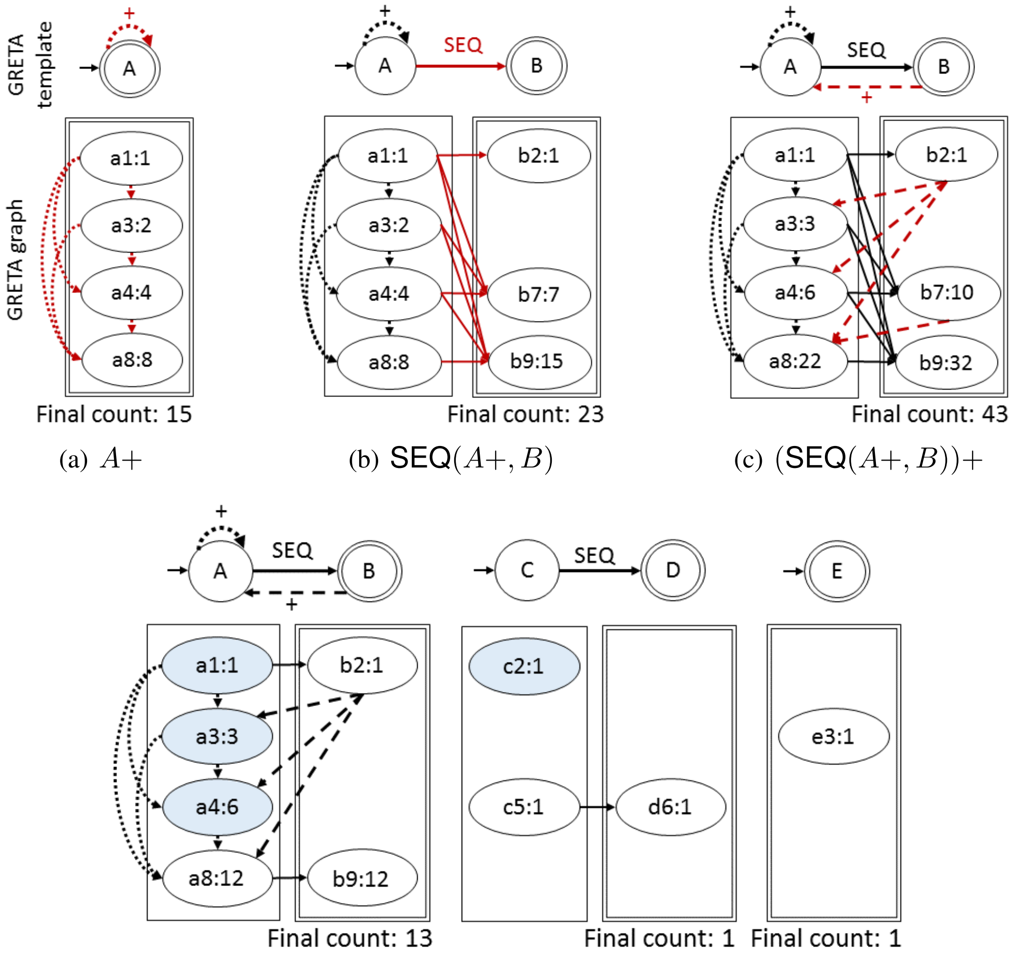
Graph-Based Event Trend Aggregation

In Sections 11.1 and 11.2, we assume that the stream I is finite, the query has neither predicates, nor grouping. We will describe how GRETA works with windows, predicates, and grouping in Section 11.3. In the examples below, we compute event trend count COUNT(*) (Definition 2.5). The same principles apply to other aggregation functions (Chapter 12).

11.1 Positive Nested Patterns

At compile time, we translate a positive pattern (Definition 2.1) into a template (Section 2.2) As events arrive at runtime, the GRETA graph is maintained according to this template.

The GRETA graph is a runtime instantiation of the template. The graph is constructed on-the-fly as events arrive. The graph compactly captures all matched trends and enables their incremental aggregation. We now informally describe the graph construction and aggregate propagation using our running example. Afterwards, Algorithm 6 defines graph-based trend aggregation. We also prove its correctness and analyze the complexity.



(a) $A+$ (b) $\text{SEQ}(A+, B)$ (c) $(\text{SEQ}(A+, B))+$

(d) $(\text{SEQ}(A+, \text{NOT SEQ}(C, \text{NOT } E, D), B))+$

Figure 11.1: Count of trends matched by the pattern P in the stream *I* = {*a1, b2, c2, a3, e3, a4, c5, d6, b7, a8, b9*}

Compact Event Trend Encoding. The graph encodes all trends and thus avoids their construction.

Vertices represent events in the stream I matched by the pattern P. Each state with label E in the template is associated with the sub-graph of events of type E in the graph. We highlight each sub-graph by a rectangle frame. If E is an end state, the frame is depicted as a double rectangle. Otherwise, the frame is a single rectangle. An event is labeled by its event type, time stamp, and intermediate aggregate (see below). Each event is stored once. Figure 11.1(c) illustrates the template and the graph for the stream I.

Edges connect adjacent events in a trend matched by the pattern P in a stream I (Definition 2.2). While transitions in the template express predecessor relationships between event types in the pattern (Section 2.2), edges in the graph capture predecessor relationships between events in a trend (Definition 2.2). In Figure 11.1(c), we depict a transition in the template and its respective edges in the graph in the same way. A path from a START to an END event in the graph corresponds to a trend. The length of these trends ranges from the shortest $(a1, b2)$ to the longest $(a1, b2, a3, a4, b7, a8, b9)$.

In summary, the GRETA graph in Figure 11.1(c) compactly captures all 43 event trends matched by the pattern P in the stream I. In contrast to the two-step approach, the graph avoids repeated computations and replicated storage of common sub-trends such as $(a1, b2)$.

Dynamic Aggregation Propagation. Intermediate aggregates are propagated through the graph from previous events to new events in dynamic programming fashion. Final aggregate is incrementally computed based on intermediate aggregates. In the examples below, we compute event trend count COUNT(*) (Definition 2.5). Same principles apply to other aggregation functions (Chapter 12).

Intermediate Count $e.count$ of an event e corresponds to the number of (sub-)trends in G that begin with a START event in G and end at e. When e is inserted into the graph, all predecessor events of e connect to e. That is, e extends all trends that ended at a predecessor event of e. To accumulate the number of trends extended by e, $e.count$ is set to the sum of counts of the predecessor events of e. In addition, if e is a START event, it starts a new trend. Thus, $e.count$ is incremented by 1. In Figure 11.1(c), the count of the START event $a4$ is set to 1 plus the sum of the counts of its predecessor events $a1, b2$, and $a3$.

$$a4.count = 1 + (a1.count + b2.count + a3.count) = 6$$

$a4.count$ is computed once, stored, and reused to compute the counts of $b7$, $a8$, and

$b9$ that $a4$ connects to. For example, the count of $b7$ is set to the sum of the counts of all predecessor events of $b7$.

$$b7.count = a1.count + a3.count + a4.count = 10$$

Final Count corresponds to the sum of the counts of all END events in the graph.

$$\mathit{final\_count} = b2.count + b7.count + b9.count = 43$$

In summary, the count of a new event is computed based on the counts of previous events in the graph following the dynamic programming principle. Each intermediate count is computed once. The final count is incrementally updated by each END event and instantaneously returned at the end of each window.

Definition 11.1 *(*GRETA ***Graph****.) The* GRETA *graph $G = (V, E, final\_count)$ for a query q and a stream I is a directed acyclic graph with a set of vertices V, a set of edges E, and a final_count. Each vertex $v \in V$ corresponds to an event $e \in I$ matched by q. A vertex v has the label $(e.type\ e.time : e.count)$ (Theorem 11.2). For two vertices $v_i, v_j \in V$, there is an edge $(v_i, v_j) \in E$ if their respective events e_i and e_j are adjacent in a trend matched by q. Event v_i is called a predecessor event of v_j.*

GRETA graphs have different shapes depending on the pattern and the stream. Figure 11.1(a) shows the graph for the pattern $A+$. Events of type B are not relevant for it. Events of type A are both START and END events. Figure 11.1(b) depicts the GRETA graph for the pattern SEQ$(A+, B)$. There are no dashed edges since b's may not precede a's.

Theorems 11.1 and 11.2 prove the correctness of the event trend count computation based on the GRETA graph.

Theorem 11.1 (Correctness of a GRETA Graph) *Let G be the* GRETA *graph for a query q and a stream I. Let $\mathcal{P}$ be the set of paths from a* START *to an* END *event in G. Let $\mathcal{T}$ be the set of trends detected by q in I. Then, the set of paths $\mathcal{P}$ and the set of trends $\mathcal{T}$ are equivalent. That is, for each path $p \in \mathcal{P}$ there is a trend $tr \in \mathcal{T}$ with the same events in the same order, and vice versa.*

Proof: ***Correctness***. For each path $p \in \mathcal{P}$, there is a trend $tr \in \mathcal{T}$ with same events in the same order, i.e., $\mathcal{P} \subseteq \mathcal{T}$. Let $p \in \mathcal{P}$ be a path. By definition, p has one START event, one END event, and any number of MID events. Edges between these events are determined by the query q such that a pair of *adjacent events in a trend* is connected by an edge. Thus, the path p corresponds to a trend $tr \in \mathcal{T}$ matched by the query q in the stream I.

Completeness. For each trend $tr \in \mathcal{T}$, there is a path $p \in \mathcal{P}$ with same events in the same order, i.e., $\mathcal{T} \subseteq \mathcal{P}$. Let $tr \in \mathcal{T}$ be a trend. We first prove that all events in tr are inserted into the graph G. Then we prove that these events are connected by directed edges such that there is a path p that visits these events in the order in which they appear in the trend tr. A START event is always inserted, while a MID or an END event is inserted if it has predecessor events since otherwise there is no trend to extend. Thus, all events of the trend tr are inserted into the graph G. The first statement is proven. *All* previous events that satisfy the query q connect to a new event. Since events are processed in order by time stamps, edges connect previous events with more recent events. The second statement is proven. ■

Theorem 11.2 (Event Trend Count Computation) *Let G be the* GRETA *graph for a query q and a stream I, $e \in I$ be an event with predecessor events Pr in G, and End be the* END *events in G. Then the following statements hold:*

(1) The intermediate count $e.count$ is the number of (sub) trends in G that start at a

START event and end at e.

$$e.count = \sum_{p \in Pr} p.count.$$

If e *is a START event,* $e.count$ *is incremented by one.*

(2) The final count is the number of trends captured by G.

$$final\_count = \sum_{end \in End} end.count.$$

Proof: (1) We prove the first statement by induction on the number of events in G.

Induction Basis: $n = 1$. If there is only one event e in the graph G, e is the only (sub-)trend captured by G. Since e is the only event in G, e has no predecessor events. The event e can be inserted into the graph only if e is a START event. Thus, $e.count = 1$.

Induction Assumption: The statement is true for n events in the graph G.

Induction Step: $n \rightarrow n + 1$. Assume a new event e is inserted into the graph G with n events and the predecessor events $Pred$ of e are connected to e. According to the induction assumption, each of the predecessor events $p \in Pred$ has a count that corresponds to the number of sub-trends in G that end at the event p. The new event e continues *all* these trends. Thus, the number of these trends is the sum of counts of all $p \in Pred$. In addition, each START event initiates a new trend. Thus, 1 is added to the count of e if e is a START event. The first statement is proven.

(2) By definition only END events may finish trends. We are interested in the number of finished trends only. Since the count of an END event end corresponds to the number of trends that finish at the event end, the total number of trends captured by the graph G is the sum of counts of all END events in G. The second statement is proven. ■

Event Trend Count Algorithm for Positive Patterns (Algorithm 6) computes the number of trends matched by the pattern P in the stream I. The set of vertices V in the

Algorithm 6 Event trend count algorithm for positive patterns

Input: Positive pattern P, stream I

Output: Count of trends matched by P in I

```
1: $process\_pos\_pattern(P, I)$ {
2: $V \leftarrow \emptyset$, $final\_count \leftarrow 0$
3: for all $e \in I$ of type $E$ do
4:     $Pr \leftarrow V.predEvents(e)$
5:     if $E = start(P)$ or $Pr \neq \emptyset$ then
6:         $V \leftarrow V \cup e,\ e.count \leftarrow (E = start(P))\ ?\ 1\ :\ 0$
7:         for all $p \in Pr$ do $e.count \mathrel{+}= p.count$
8:         if $E = end(P)$ then $final\_count \mathrel{+}= e.count$
9: return $final\_count$ }
```

GRETA graph is initially empty (Line 2). Since each edge is traversed exactly once, edges are not stored. When an event e of type E arrives, the auxiliary method *V.predEvents(e)* returns the predecessor events of e in the graph (Line 4). A START event is always inserted into the graph since it always starts a new trend, while a MID or an END event is inserted only if it has predecessor events (Lines 5–6). The count of e is increased by the counts of its predecessor events (Line 7). If e is a START event, its count is incremented by 1 (Line 6). If e is an END event, the final count is increased by the count of e (Line 8). This final count is returned (Line 9).

Theorem 11.3 (Correctness of the Event Trend Count Algorithm) *Given a positive pattern P and a stream I, Algorithm 6 constructs the* GRETA *graph for P and I (Definition 23.5) and computes the intermediate and final counts (Theorem 11.2).*

Proof: ***Graph Construction***. Each event $e \in I$ is processed (Line 3). A START event is always inserted, while a MID or an END event is inserted if it has predecessor events (Lines 4–6). Thus, the set of vertices V of the graph corresponds to events in the stream I that are matched by the pattern P. Each predecessor event p of a new event e is connected to e (Lines 8–9). Therefore, the edges E of the graph capture adjacency relationships between events in trends matched by the pattern P.

Count Computation. Initially, the intermediate count $e.count$ of an event e is either 1 if e is a START event or 0 otherwise (Line 7). $e.count$ is then incremented by $p.count$ of each predecessor event p of e (Lines 8, 10). Thus, $e.count$ is correct. Initially, the final count is 0 (Line 2). Then, it is incremented by $e.count$ of each END event e in the graph. Since $e.count$ is correct, the final count is correct too. ■

We analyze complexity of Algorithm 6 in Section 11.5.

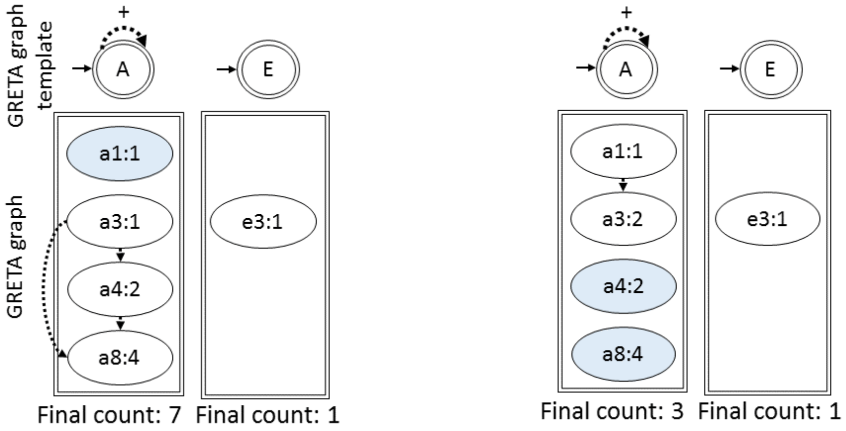
11.2 Patterns with Nested Negation

To handle nested patterns with negation, we split the pattern into positive and negative sub-patterns at compile time (Section 2.3). At runtime, we then maintain the GRETA graph for each of these sub-patterns.

Definition 11.2 *(**Dependent** GRETA **Graph**.) Let G_N and G_P be* GRETA *graphs that are constructed according to templates $\mathfrak{T}_N$ and $\mathfrak{T}_P$ respectively. The* GRETA *graph G_P is dependent on the graph G_N if there is a previous or following link from $\mathfrak{T}_N$ to an event type in $\mathfrak{T}_P$.*

Definition 11.3 *(**Invalid Event**.) Let G_P and G_N be* GRETA *graphs such that G_P is dependent on G_N. Let $tr = (e_1, \ldots, e_n)$ be a finished trend captured by G_N, i.e., e_n is an END event. The trend tr marks all events of the previous event type that arrived before $e_1.time$ as invalid to connect to any event of the following event type that will arrive after $e_n.time$.*

Example 11.1 *Figure 11.1(d) depicts the graphs for the sub-patterns from Example 2.2. The match $e3$ of the negative sub-pattern E marks $c2$ as invalid to connect to any future d. Invalid events are highlighted by a darker background. Analogously, the match $(c5, d6)$ of the negative sub-pattern SEQ(C, D) marks all a's before $c5$ (in this example they are*



(a) SEQ$(A+,$ NOT $E)$ (b) SEQ(NOT $E, A+)$

Figure 11.2: Count of trends matched by the pattern P in the stream *I* = {*a1, b2, c2, a3, e3, a4, c5, d6, b7, a8, b9*}

$a1, a3, a4$*) as invalid to connect to any* b *after* $d6$*. The event* $b7$ *has no valid predecessor events and thus cannot be inserted. The event* $a8$ *is inserted and all previous* a*'s are connected to it. The marked* a*'s are valid to connect to new* a*'s. The event* $b9$ *is inserted and its valid predecessor event* $a8$ *is connected to it. The marked* a*'s may not connect to* $b9$*.*

Figures 11.2(a) and 11.2(b) depict the graphs for the patterns from Example 2.3. The trend $e3$ *of the negative sub-pattern* E *marks all previous* a*'s as invalid in Figure 11.2(a). In contrast, in Figure 11.2(b),* $e3$ *invalidates all following* a*'s.*

Event Pruning. Negation allows us to purge events from the graph to speed-up insertion of new events and aggregation propagation. The following events can be deleted:

• ***Finished Trend Pruning***. A finished trend that is matched by a negative sub-pattern can be deleted once it has invalidated all respective events.

• ***Invalid Event Pruning***. An invalid event of type $end(P_i)$ will never connect to any new event if events of type $end(P_i)$ may precede only events of type $start(P_j)$. The aggregates of such invalid events will not be propagated. Thus, such events may be safely purged from the graph.

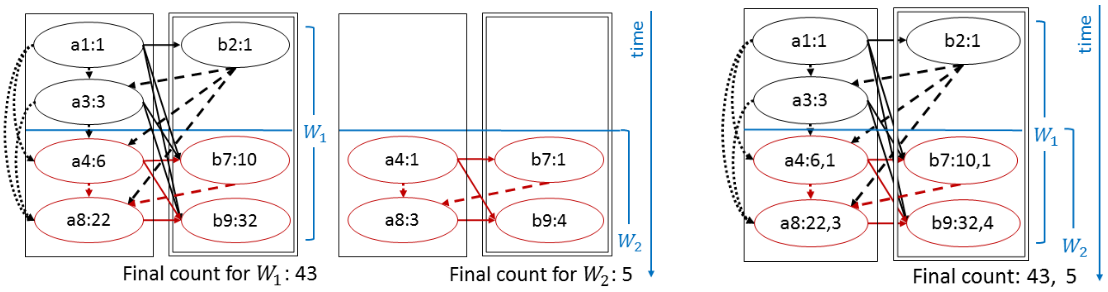
Example 11.2 *Continuing Example 11.1 in Figure 11.1(d), the invalid $c2$ will not connect to any new event since c's may connect only to d's. Thus, $c2$ is purged. $e3$ is also deleted. Once a's before $c5$ are marked, $c5$ and $d6$ are purged. In contrast, the marked events $a1, a3$, and $a4$ may not be removed since they are valid to connect to future a's. In Figures 11.2(a) and 11.2(b), $e3$ and all marked a's are deleted.*

Theorem 11.4 ***(Correctness of Event Pruning.)*** *Let G_P and G_N be* GRETA *graphs such that G_P is dependent on G_N. Let G'_P be the same as G_P but without invalid events of type $end(P_i)$ if $end(P_i)$ is the only predecessor type of $start(P_j)$ in P, i.e., $P.predTypes(start(P_j)) = \{end(P_i)\}$. Let G'_N be the same as G_N but without finished event trends. Then, G'_P returns the same aggregation results as G_P.*

Proof: We first prove that all invalid events are marked in G_P despite finished trend pruning in G'_N. We then prove that G'_P returns the same aggregation result as G_P despite invalid event pruning.

All invalid events are marked in G_P. Let $tr = (e_1, \ldots, e_n)$ be a finished trend in G_N. Let Inv be the set of events that are invalidated by tr in G_P. By Definition 11.3, all events in Inv arrive before $e_1.time$. According to Chapter 2, events arrive in-order by time stamps. Thus, no event with time stamp less than $e_1.time$ will arrive after e_1. Hence, even if an event $e_i \in \{e_1, \ldots, e_{n-1}\}$ connects to future events in G_N, no event $e \notin Inv$ in G_P can be marked as invalid.

G_P ***and*** G'_P ***return the same aggregate***. Let e be an event of type $end(P_i)$ that is marked as invalid to connect to events of type $start(P_j)$ that arrive after $e_n.time$. Before $e_n.time$, e is valid and its count is correct by Theorem 11.2. Since events arrive in-order by time stamps, no event with time stamp less than $e_n.time$ will arrive after e_n. After $e_n.time$, e will not connect to any event and the count of e will not be propagated if $P.predTypes(start(P_j)) = \{end(P_i)\}$. Hence, deletion of e does not affect the final aggregate of G_P. ■



(a) GRETA sub-graph replication (b) GRETA sub-graph sharing

Figure 11.3: Sliding window WITHIN 10 *seconds* SLIDE 3 *seconds*

GRETA Algorithm for Patterns with Negation. Algorithm 6 is called on each event sub-pattern with the following modifications. First, only valid predecessor events are returned in Line 4. Second, if the algorithm is called on a negative sub-pattern N and a match is found in Line 12, then all previous events of the previous event type of N are either deleted or marked as incompatible with any future event of the following event type of N. Afterwards, the match of N is purged from the graph. GRETA concurrency control is described in Section 11.4.

11.3 Other Query Clauses

We now expand our GRETA approach to handle sliding windows, predicates, and grouping. Thus, we can support all clauses of an event query (Definition 2.5) except the event matching semantics that will be handled in Chapters 14–19. We will describe the possible extensions of the language in Chapter 12.

Sliding Windows. Due to the continuous nature of streaming, an event may contribute to the aggregation results of several overlapping windows. Furthermore, events may expire in some windows but remain valid in other windows.

- **GRETA *Sub-Graph Replication***. A naive solution would build a GRETA graph for each window independently from other windows. Thus, it would replicate an event e

across all windows that e falls into. Worse yet, this solution introduces repeated computations, since an event p may be predecessor event of e in multiple windows.

Example 11.3 *In Figure 11.3(a), we count the number of trends matched by the pattern* $(\mathsf{SEQ}(A+, B))+$ *within a 10-seconds-long window that slides every 3 seconds. The events* $a1$–$b9$ *fall into window* W_1*, while the events* $a4$–$b9$ *fall into window* W_2*. If a* GRETA *graph is constructed for each window, the events* $a4$–$b9$ *are replicated in both windows and their predecessor events are recomputed for each window.*

- **GRETA *Sub-Graph Sharing***. To avoid these drawbacks, we share a sub-graph G across all windows to which G belongs. Let e be an event that falls into k windows. The event e is stored once and its predecessor events are computed once across all k windows. The event e maintains a count for each window. To differentiate between k counts maintained by e, each window is assigned an identifier wid [67]. The count with identifier wid of e ($e.count_{wid}$) is computed based on the counts with identifier wid of e's predecessor events (Line 10 in Algorithm 6). The final count for a window wid ($final\_count_{wid}$) is computed based on the counts with identifier wid of the END events in the graph (Line 12). In Example 11.3, the events $a4$–$b9$ fall into two windows and thus maintain two counts in Figure 11.3(b). The first count is for W_1, the second one for W_2.

Predicates on vertices and edges of the GRETA graph are handled differently by the GRETA runtime.

- ***Vertex Predicates*** restrict the vertices in the GRETA graph. They are evaluated on single events (Section 2.5). *Local predicates* purge irrelevant events early. We associate each local predicate with its respective state in the GRETA template. *Equivalence predicates* partition the stream by event attribute values. Thereafter, GRETA queries are evaluated against each sub-stream in a divide and conquer fashion.

- ***Edge Predicates*** restrict the edges in the graph (Line 4 of Algorithm 6). They are

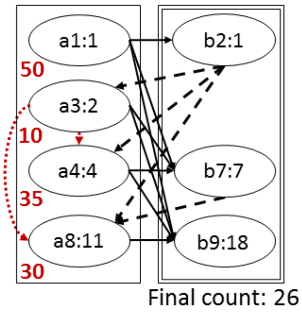


Figure 11.4: Edge predicate $A.attr < \mathsf{NEXT}(A).attr$

evaluated on a pair of adjacent events in a trend (Section 2.5). We associate each edge predicate with its respective transition in the GRETA template.

Example 11.4 *The edge predicate $A.attr$ < NEXT$(A).attr$ in Figure 19.3 requires the value of attribute attr of events of type A to increase from one event to the next in a trend. The attribute value is shown in the bottom left corner of a vertex. Only two dotted edges satisfy this predicate.*

Event Trend Grouping. As illustrated by our motivating examples in Section 1.1, event trend aggregation often requires event trend grouping. Analogously to A-Seq [2], our GRETA runtime first partitions the event stream into sub-streams by the values of grouping attributes. A GRETA graph is then maintained separately for each sub-stream. Final aggregates are output per sub-stream.

11.4 Implementation

Putting Sections 11.1–11.3 together, we now describe the GRETA runtime data structures and parallel processing.

Data Structure for a Single GRETA Graph. Edges logically capture the paths for aggregation propagation in the graph. Each edge is traversed *exactly once* to compute the aggregate of the event to which the edge connects (Lines 8–10 in Algorithm 6). Hence, edges are not stored.

Vertices must be stored in such a way that the predecessor events of a new event can be efficiently determined (Line 4). To this end, we leverage the following data structures. To quickly locate *previous* events, we divide the stream into non-overlapping consecutive time intervals, called ***Time Panes*** [36]. Each pane contains all vertices that fall into it based on their time stamps. These panes are stored in a time-stamped array in increasing order by time (Figure 11.5). The size of a pane depends on the window specifications and stream rate such that each query window is composed of several panes – allowing panes to be shared between overlapping windows [33, 36]. To efficiently find vertices of *predecessor event types*, each pane contains an ***Event Type Hash Table*** that maps event types to vertices of this type.

To support *edge predicates*, we utilize a tree index that enables efficient range queries. The overhead of maintaining ***Vertex Trees*** is reduced by event sorting and a pane purge mechanism. An event is inserted into the Vertex Tree for its respective pane and event type. This sorting by time and event type reduces the number of events in each tree. Furthermore, instead of removing single expired events from the Vertex Trees, a whole pane with its associated data structures is deleted after the pane has contributed to all windows to which it belongs. To support *sliding windows*, each vertex e maintains a ***Window Hash Table*** storing an aggregate per window that e falls into. Similarly, we store final aggregates per window in the ***Results Hash Table***.

Data Structure for GRETA Graph Dependencies. To support negative sub-patterns, we maintain a ***Graph Dependencies Hash Table*** that maps a graph identifier G to the identifiers of graphs upon which G depends. Below we describe how such interdependent graphs are processed concurrently.

Parallel Processing. The grouping clause partitions the stream into sub-streams that

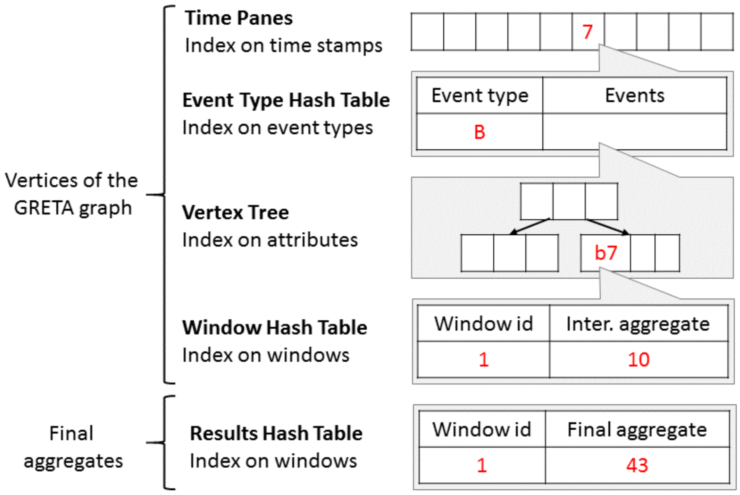


Figure 11.5: Data structure for a single GRETA graph

are processed in parallel *independently* from each other. Such stream partitioning enables a highly scalable execution as demonstrated in Section 19.4.

In contrast, negative sub-patterns require concurrent maintenance of *interdependent* GRETA graphs. To avoid race conditions, we deploy the time-based transaction model [68]. A *stream transaction* is a sequence of operations triggered by all events with the same time stamp on the same GRETA graph. The application time stamp of a transaction (and all its operations) coincides with the application time stamp of the triggering events. For each time stamp t and each GRETA graph G, our time-driven scheduler waits till the processing of all transactions with time stamps smaller than t on the graph G and other graphs that G depends upon is completed. Then, the scheduler extracts all events with the time stamp t, wraps their processing into transactions, and submits them for execution.

11.5 Complexity Analysis

We now analyze the complexity of GRETA. Since a negative sub-pattern is processed analogously to a positive sub-pattern (Section 11.2), we focus on positive patterns below.

Theorem 11.5 (Complexity) *Let q be a query with edge predicates, I be a stream, G be the* GRETA *graph for q and I, n be the number of events per window, and k be the number of windows into which an event falls. The time complexity of* GRETA *is $O(n^2k)$, while its space complexity is $O(nk)$.*

Proof: **Time Complexity**. Let e be an event of type E. The following steps are taken to process e. Since events arrive in-order by time stamps (Chapter 2), the Time Pane to which e belongs is always the latest one. It is accessed in constant time. The Vertex Tree in which e will be inserted is found in the Event Type Hash Table mapping the event type E to the tree in constant time. Depending on the attribute values of e, e is inserted into its Vertex Tree in logarithmic time $O(log_bm)$ where b is the order of the tree and m is the number of elements in the tree, $m \leq n$.

The event e has n predecessor events in the worst case, since each vertex connects to each following vertex under the skip-till-any-match semantics. Let x be the number of Vertex Trees storing previous vertices that are of predecessor event types of E and fall into a sliding window $wid \in windows(e)$, $x \leq n$. Then, the predecessor events of e are found in $O(log_bm + m)$ time by a range query in one Vertex Tree with m elements. The time complexity of range queries in x Vertex Trees is computed as follows:

$$\sum_{i=1}^{x} O(log_bm_i + m_i) = \sum_{i=1}^{x} O(m_i) = O(n).$$

If e falls into k windows, a predecessor event of e updates at most k aggregates of e. If e is an END event, it also updates k final aggregates. Since these aggregates are maintained in hash tables, updating one aggregate takes constant time. GRETA concurrency control ensures that all graphs this graph G depends upon finishing processing all events with time stamps less than t before G may process events with time stamp t. Therefore, all invalid events are marked or purged before aggregates are updated in G at time t. Consequently, an aggregate is updated at most once by the same event. Putting it all together, the time complexity is:

$$O(n(log_b m + nk)) = O(n^2 k).$$

Space Complexity. The space complexity is determined by x Vertex Trees and k counts maintained by each vertex.

$$\sum_{i=1}^{x} O(m_i k) = O(nk).$$

■

12

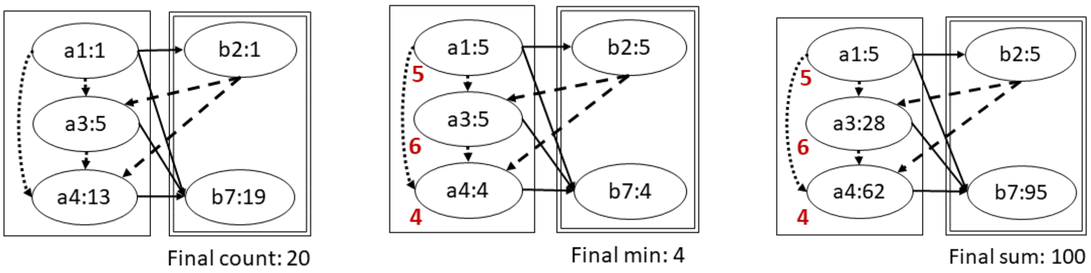
Extensions of the GRETA Approach

So far, we have described how the query clauses (Definition 2.5) are handled by GRETA. In this chapter, we sketch how our GRETA approach can be extended to support additional language features.

Other Aggregation Functions. While Theorem 11.2 defines event trend count computation, i.e., COUNT(*), we now sketch how the principles of incremental event trend aggregation proposed by our GRETA approach apply to other aggregation functions (Definition 2.5).

Theorem 12.1 (Event Trend Aggregation Computation) *Let G be the* GRETA *graph for a query q and a stream I, $e, e' \in I$ be events in G such that $e.type = E$, $e'.type \neq E$, $attr$ is an attribute of e, Pr and Pr' be the predecessor events of e and e' respectively in G, and End be the* END *events in G.*

$$
\begin{aligned}
&e.count_E = e.count + \textstyle\sum_{p \in Pr} p.count_E \\
&e'.count_E = \textstyle\sum_{p' \in Pr'} p'.count_E \\
&\textsf{\textit{COUNT}}(E) = \textstyle\sum_{end \in End} end.count_E
\end{aligned}
$$



(a) COUNT(A) (b) MIN$(A.attr)$ (c) SUM$(A.attr)$

Figure 12.1: Aggregation of trends matched by the pattern *P=(SEQ(A+,B))+* in the stream *I* = $\{$*a1, b2, a3, a4, b7*$\}$ where *a1.attr=5, a3.attr=6,* and *a4.attr=4*

$$e.min = min_{p \in Pr}(e.attr, p.min)$$

$$e'.min = min_{p' \in Pr'}(p'.min)$$

$$\textit{MIN}(E.attr) = min_{end \in End}(end.min)$$

$$e.max = max_{p \in Pr}(e.attr, p.max)$$

$$e'.max = max_{p' \in Pr'}(p'.max)$$

$$\textit{MAX}(E.attr) = max_{end \in End}(end.max)$$

$$e.sum = e.attr * e.count + \sum_{p \in Pr} p.sum$$

$$e'.sum = \sum_{p' \in Pr'} p'.sum$$

$$\textit{SUM}(E.attr) = \sum_{end \in End} end.sum$$

$$\textit{AVG}(E.attr) = \textit{SUM}(E.attr) / \textit{COUNT}(E)$$

Analogously to Theorem 11.2, Theorem 12.1 can be proven by induction on the number of events in the graph G.

Example 12.1 *In Figure 12.1, we compute COUNT(A), MIN$(A.attr)$, and SUM$(A.attr)$ based on the* GRETA *graph for the pattern P and the stream I. Compare the aggregation results with Example 2.8. MAX$(A.attr) = 6$ is computed analogously to MIN$(A.attr)$. AVG$(A.attr)$ is computed based on SUM$(A.attr)$ and COUNT(A).*

Disjunction and **Conjunction** can be supported by our GRETA approach without changing its complexity because the count for a disjunctive or a conjunctive pattern P can be computed based on the counts for the sub-patterns of P as defined below. Let P_i and P_j be patterns (Definition 2.1). Let P_{ij} be the pattern that detects trends matched by both P_i and P_j. P_{ij} can be obtained from its DFA representation that corresponds to the intersection of the DFAs for P_i and P_j [69]. Let $\textsf{COUNT}(P)$ denote the number of trends matched by a pattern P. Let $C_{ij} = \textsf{COUNT}(P_{ij})$, $C_i = \textsf{COUNT}(P_i) - C_{ij}$, and $C_j = \textsf{COUNT}(P_j) - C_{ij}$.

In contrast to event sequence and Kleene plus (Definition 2.1), disjunctive and conjunctive patterns do not impose a time order constraint upon trends matched by their sub-patterns.

Disjunction $(P_i \vee P_j)$ matches a trend that is a match of P_i or P_j. $\textsf{COUNT}(P_i \vee P_j) = C_i + C_j - C_{ij}$. C_{ij} is subtracted to avoid counting trends matched by P_{ij} twice.

Conjunction $(P_i \wedge P_j)$ matches a pair of trends tr_i and tr_j where tr_i is a match of P_i and tr_j is a match of P_j. $\textsf{COUNT}(P_i \wedge P_j) = C_i * C_j + C_i * C_{ij} + C_j * C_{ij} + \binom{C_{ij}}{2}$ since each trend detected only by P_i (not by P_j) is combined with each trend detected only by P_j (not by P_i). In addition, each trend detected by P_{ij} is combined with each other trend detected only by P_i, only by P_j, or by P_{ij}.

Kleene Star and **Optional Sub-patterns** can also be supported without changing the complexity because they are syntactic sugar operators. Indeed, $\textsf{SEQ}(P_i*, P_j) = \textsf{SEQ}(P_i+, P_j) \vee P_j$ and $\textsf{SEQ}(P_i?, P_j) = \textsf{SEQ}(P_i, P_j) \vee P_j$.

Constraints on Minimal Trend Length. While our language does not have an explicit constraint on the minimal length of a trend, one way to model this constraint in GRETA is to unroll a pattern to its minimal length. For example, assume we want to detect trends matched by the pattern $A+$ and with minimal length 3. Then, we unroll the pattern $A+$ to length 3 as follows: $\textsf{SEQ}(A, A, A+)$.

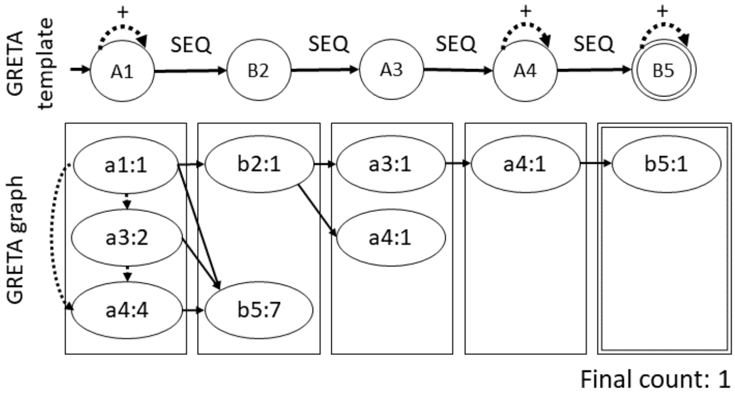


Figure 12.2: Count of trends matched by the pattern P in the stream *I* = $\{$*a1 ,b2, a3, a4, b5*$\}$

Any correct trend processing strategy must keep all current trends, including those which did not reach the minimal length yet. Thus, these constraints do not change the complexity of trend detection. They could be added to our language as syntactic sugar.

Multiple Event Type Occurrences in a Pattern. While in Chapters 10–11 we assumed for simplicity that an event type may occur in a pattern at most once, we now sketch a few modifications of our GRETA approach allowing to drop this assumption. First, we assign a unique identifier to each event type. For example, SEQ$(A+, B, A, A+, B+)$ is translated into P = SEQ$(A1+, B2, A3, A4+, B5+)$. Then, each state in a GRETA template has a unique label (Figure 12.2). Our GRETA approach still applies with the following modifications. (1) Events in the first sub-graph are START events, while events in the last sub-graph are END events. (2) An event e may not be its own predecessor event since an event may occur at most once in a trend. (3) An event e may be inserted into several sub-graphs. Namely, e is inserted into a sub-graph for $e.type$ if e is a START event or e has predecessor events. For example, a4 is inserted into the sub-graphs for A1, A3, and A4 in Figure 12.2. a4 is a START event in the sub-graph for A1. b5 is inserted into the sub-graphs for B2 and B5. b5 is an END event in the sub-graph for B5.

Since an event is compared to each previous event in the graph in the worst case, our GRETA approach still has quadratic time complexity $O(n^2k)$ where n is the number of events per window and k is the number of windows into which an event falls (Theorem 11.5). Let t be the number of occurrences of an event type in a pattern. Then, each event is inserted into t sub-graphs in the worst case. Thus, the space complexity increases by the multiplicative factor t, i.e., $O(tnk)$, where n remains the dominating cost factor for high-rate streams and meaningful patterns (Theorem 11.5).

13

Performance Evaluation

13.1 Experimental Setup

Infrastructure. We have implemented our GRETA approach in Java with JRE 1.7.0_25 running on Ubuntu 14.04 with 16-core 3.4GHz CPU and 128GB of RAM. We execute each experiment three times and report their average.

Data Sets. We evaluate the performance of our GRETA approach using the following data sets.

• ***Stock Real Data Set***. We use the real NYSE data set [43] with 225k transaction records of 10 companies. Each event carries volume, price, time stamp in seconds, type (sell or buy), company, sector, and transaction identifiers. We replicate this data set 10 times.

• ***Linear Road Benchmark Data Set***. We use the traffic simulator of the Linear Road benchmark [39] for streaming systems to generate a stream of position reports from vehicles for 3 hours. Each position report carries a time stamp in seconds, a vehicle identifier, its current position, and speed. Event rate gradually increases during 3 hours until it reaches 4k events per second.

Attribute	Distribution	min–max
Mapper id, job id	Uniform	0–10
CPU, memory	Uniform	0–1k
Load	Poisson with $\lambda = 100$	0–10k

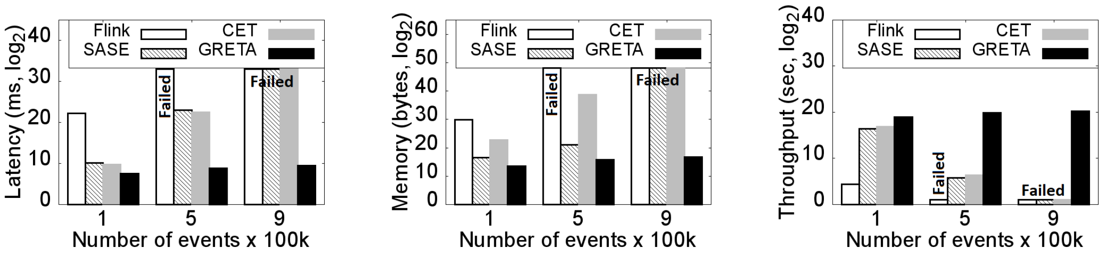
Table 13.1: Attribute values

- ***Cluster Monitoring Data Set***. Our stream generator creates cluster performance measurements for 3 hours. Each event carries a time stamp in seconds, mapper and job identifiers, CPU, memory, and load measurements. The distribution of attribute values is summarized in Table 13.1. The stream rate is 3k events per second.

Event Queries. Unless stated otherwise, we evaluate query Q_1 [6] and its nine variations against the stock data set. These query variations differ by the predicate $S.price * X < \textsf{NEXT}(S).price$ that requires the price to increase (or decrease with $>$) by $X \in \{1, 1.05, 1.1, 1.15, 1.2\}$ percent from one event to the next in a trend. Similarly, we evaluate queries Q_2 and Q_3 [6] and their nine variations against the cluster and the Linear Road data sets respectively. We have chosen these queries because they contain all clauses (Definition 2.5) and allow us to measure the effect of each clause on the number of matched trends. The number of matched trends ranges from few hundreds to trillions. In particular, we vary the number of events per window, presence of negative sub-patterns, predicate selectivity, and number of event trend groups.

Methodology. We compare GRETA to CET [5], SASE [3], and Flink [30]. To achieve a fair comparison, we have implemented CET and SASE on top of our platform. We execute Flink on the same hardware as our platform. While Chapter 20 is devoted to a detailed discussion of these approaches, we briefly sketch their main ideas below.

- ***CET*** [5] is the state-of-the-art approach to event trend detection. It stores and reuses partial event trends while constructing the final event trends. Thus, it avoids the recomputation of common sub-trends. While CET does not explicitly support aggregation,



(a) Latency
(b) Memory
(c) Throughput

Figure 13.1: Positive patterns (Stock real data set)

we extended this approach to aggregate event trends upon their construction.

• ***SASE*** [3] supports aggregation, nested Kleene patterns, predicates, and windows. It implements the two-step approach as follows. (1) Each event e is stored in a stack and pointers to e's previous events in a trend are stored. For each window, a DFS-based algorithm traverses these pointers to construct all trends. (2) These trends are aggregated.

• ***Flink*** [30] is an open-source streaming platform that supports event pattern matching. We express our queries using Flink operators. Like other industrial systems [31, 41, 70], Flink does not explicitly support Kleene closure. Thus, we flatten our queries, i.e., for each Kleene query q we determine the length l of the longest match of q. We specify a set of fixed-length event sequence queries that cover all possible lengths from 1 to l. Flink is a two-step approach.

Metrics. We measure common metrics for streaming systems, namely, *latency, throughput*, and *memory*. *Latency* measured in milliseconds corresponds to the peak time difference between the time of the aggregation result output and the arrival time of the latest event that contributes to the respective result. *Throughput* corresponds to the average number of events processed by all queries per second. *Memory* consumption measured in bytes is the peak memory for storing the GRETA graph for GRETA, the CET graph and trends for CET, events in stacks, pointers between them, and trends for SASE, and trends for Flink.

13.2 Number of Events per Window

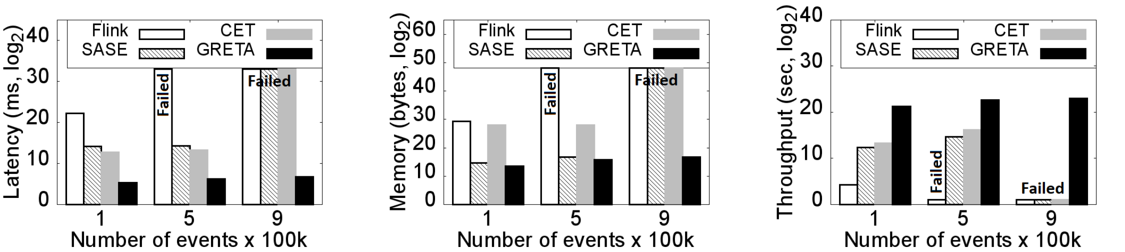
Positive Patterns. In Figure 13.1, we evaluate positive patterns against the stock real data set while varying the number of events per window.

Flink does not terminate within several hours if the number of events exceeds 100k because Flink is a two-step approach that evaluates a set of event sequence queries for each Kleene query. Both the unnecessary event sequence construction and the increased query workload degrade the performance of Flink. For 100k events per window, Flink requires 82 minutes to terminate, while its memory requirement for storing all event sequences is close to 1GB. Thus, Flink is neither real time nor lightweight.

SASE. The latency of SASE grows exponentially in the number of events until it fails to terminate for more than 500k events. Its throughput degrades exponentially. Delayed responsiveness of SASE is explained by the DFS-based stack traversal which re-computes each sub-trend tr for each longer trend containing tr. The memory requirement of SASE exceeds the memory consumption of GRETA 50–fold because DFS stores the trend that is currently being constructed. Since the length of a trend is unbounded, the peak memory consumption of SASE is significant.

CET. Similarly to SASE, the latency of CET grows exponentially in the number of events until it fails to terminate for more than 700k events. Its throughput degrades exponentially until it becomes negligible for over 500k events. In contrast to SASE, CET utilizes the available memory to store and reuse common sub-trends instead of recomputing them. To achieve almost double speed-up compared to SASE, CET requires 3 orders of magnitude more memory than SASE for 500k events.

GRETA consistently outperforms all two-step approaches above regarding all three metrics because it does not waste computational resources to construct and store exponentially many event trends. Instead, GRETA incrementally computes event trend ag-

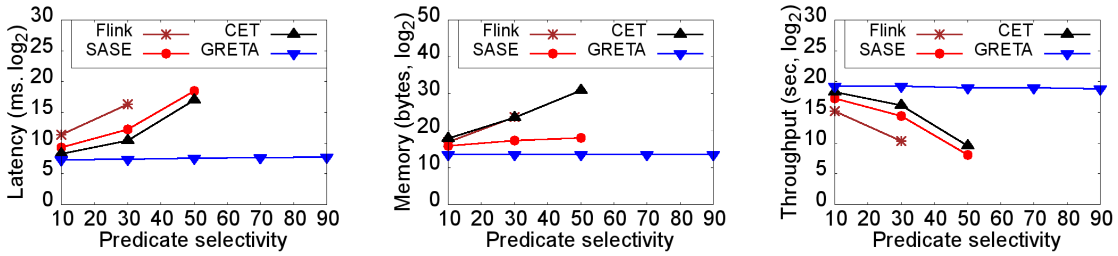


(a) Latency (b) Memory (c) Throughput

Figure 13.2: Patterns with negative sub-patterns (Stock real data set)

gregation. Thus, it achieves 4 orders of magnitude speed-up compared to all approaches above. GRETA also requires 4 orders of magnitude less memory than Flink and CET since these approaches store event trends. The memory requirement of GRETA is comparable to SASE because SASE stores only one trend at a time. Nevertheless, GRETA requires 50–fold less memory than SASE for 500k events.

Patterns with Negative Sub-Patterns. In Figure 13.2, we evaluate the same patterns as in Figure 13.1 but with negative sub-patterns against the stock real data set while varying the number of events. Compared to Figure 13.1, the latency and memory consumption of all approaches except Flink significantly decreased, while their throughput increased. Negative sub-patterns have no significant effect on the performance of Flink because Flink evaluates multiple event sequence queries instead of one Kleene query and constructs all matched event sequences. In contrast, negation reduces the GRETA graph, the CET graph, and the SASE stacks *before* event trends are constructed and aggregated based on these data structures. Thus, both CPU and memory costs reduce. Despite this reduction, SASE and CET fail to terminate for over 700k events.



(a) Latency

(b) Memory

(c) Throughput

Figure 13.3: Selectivity of edge predicates (Linear Road benchmark data set)

13.3 Selectivity of Edge Predicates

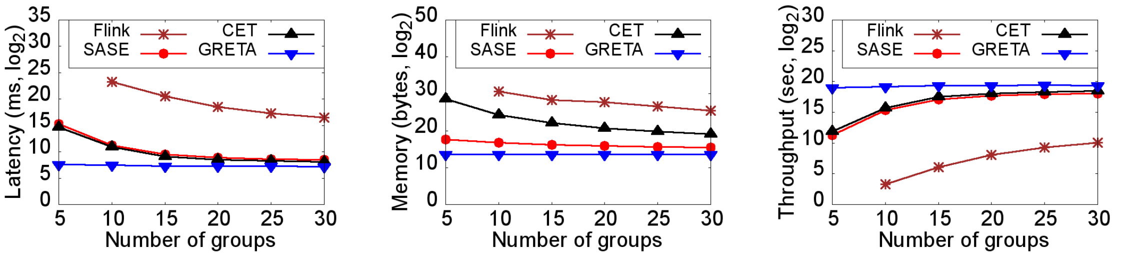
In Figure 13.3, we evaluate positive patterns against the Linear Road benchmark data set while varying the selectivity of edge predicates. We focus on the selectivity of edge predicates because vertex predicates determine the number of trend groups (Section 11.3) that is varied in Section 19.4. To ensure that the two-step approaches terminate in most cases, we set the number of events per window to 100k.

The latency of Flink, SASE, and CET grows exponentially with the increasing predicate selectivity until they fail to terminate when the predicate selectivity exceeds 50%. In contrast, the performance of GRETA remains fairly stable regardless of the predicate selectivity. GRETA achieves 2 orders of magnitude speed-up and throughput improvement compared to CET for 50% predicate selectivity.

The memory requirement of Flink and CET grows exponentially (these lines coincide in Figure 13.3(b)). The memory requirement of SASE remains fairly stable but almost 22–fold higher than for GRETA for 50% predicate selectivity.

13.4 Number of Event Trend Groups

In Figure 13.4, we evaluate positive patterns against the cluster monitoring data set while varying the number of trend groups. The number of events per window is 100k.



(a) Latency (b) Memory (c) Throughput

Figure 13.4: Number of event trend groups (Cluster monitoring data set)

The latency and memory consumption of Flink, SASE, and CET decrease exponentially with the increasing number of event trend groups, while their throughput increases exponentially. Since trends are constructed per group, their number and length decrease with the growing number of groups. Thus, both CPU and memory costs reduce. In contrast, GRETA performs equally well independently from the number of groups since event trends are never constructed. Thus, GRETA achieves 4 orders of magnitude speed-up compared to Flink for 10 groups and 2 orders of magnitude speed-up compared to CET and SASE for 5 groups.

14

Cogra Approach Overview

In this chapter, we propose our COGRA approach that extends GRETA to support rich event matching semantics (Section 2.4) and further optimizes event trend aggregation. Similarly to GRETA, COGRA pushes aggregation inside Kleene closure computation. This way, it avoids trend construction with exponential costs. In contrast to GRETA, COGRA maintains aggregates at the coarsest possible granularity level. Thus, it minimizes the number of aggregates and reduces both time and space complexity.

To support time-critical streaming applications (Section 1.1), we solve the following ***Event Trend Aggregation Problem***. Given an event trend aggregation query q (Definition 2.5) evaluated under any of the event matching semantics (Definitions 2.2, 2.3, and 2.4) over an event stream I, our goal is to compute the aggregation results of q with *minimal latency*.

Figure 14.1 illustrates our COGRA framework. The ***Static Query Analyzer*** selects the granularity level at which the aggregates are maintained for a query q. This choice is determined by event types, predicates, and event matching semantics of q. Thus, we analyze the pattern (Section 2.2), classify the predicates (Section 2.5), and select the granularity level as described below. The results of this query analysis are encoded into

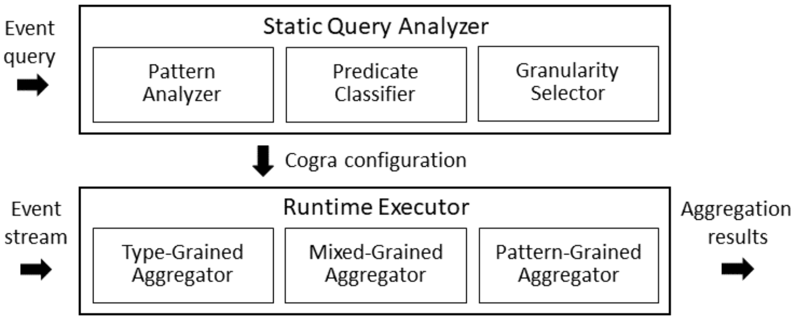


Figure 14.1: COGRA framework

Semantics	Without predicates on adjacent events	With predicates on adjacent events
ANY	Type	Mixed
NEXT, CONT	Pattern	

Table 14.1: Granularity selection

the COGRA configuration to guide our ***Runtime Executor*** (Chapters 15 and 16).

The granularity selector determines the coarsest granularity level at which aggregates can be maintained for a query q (Table 14.1).

Type-grained aggregator. If the query q is evaluated under the skip-till-any-match semantics and has no predicates on adjacent events, our executor maintains an aggregate per each event type in the pattern (Section 15.1).

Mixed-grained aggregator. If the query q is evaluated under the skip-till-any-match semantics and has predicates on adjacent events θ, our executor maintains the aggregates at mixed granularity levels, namely, either per event e if e is required to evaluate the predicates θ or per event type *e.type* otherwise (Section 15.2).

Pattern-grained aggregator. If the query q is evaluated under the skip-till-next-match or contiguous semantics, our executor adapts the pattern-grained aggregation strategy. Namely, only the final aggregate of q and the intermediate aggregate of the last event matched by q are maintained (Section 16.2).

15

Skip-Till-Any-Match Semantics

In this chapter, we define coarse-grained event trend aggregation under the skip-till-any-match semantics. In this case, aggregates are maintained either at type granularity level (Section 15.1) or at mixed granularity levels (Section 15.2).

15.1 Type-Grained Aggregator

We now propose maintaining aggregates at the *type granularity level* and discard all events once they have updated the aggregates. Thus, we minimize the number of aggregates and further reduce the costs. Type-grained event trend aggregation applies if *no predicates on adjacent events* exist in the query. Indeed, without such predicates, the adjacency relation between events is determined by event types, event time stamps, and predicates on single events (Definition 2.2).

Let e be an event of type E and $\mathcal{T} = P.predTypes(E)$ be predecessor types of E in the pattern P (Section 2.2). When e arrives, *all* previously matched events of types $E' \in \mathcal{T}$ are adjacent to e (Figure 15.1(a)). Thus, a count can be assigned to each type in the FSA representation of P (Figure 15.1(b)). An event e updates the count of its type E and is

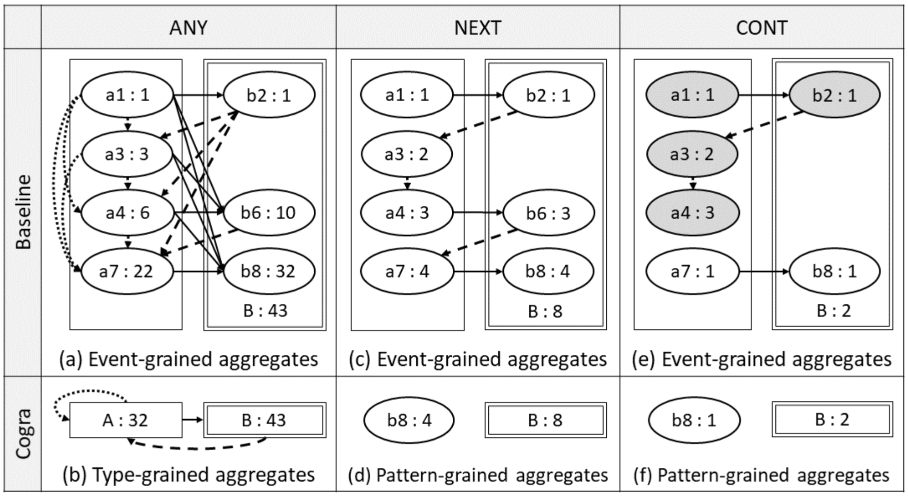


Figure 15.1: Baseline approach vs. our COGRA approach for the pattern *P=(SEQ(A+,B))+* and the stream *I=*$\{$*a1,b2,a3,a4,c5,b6,a7,b8*$\}$

discarded thereafter. The final count corresponds to the count of the end type of P.

Example 15.1 *According to our predecessor relationship analysis in Section 2.2, all previously matched a's and b's are adjacent to $a7$ in Figure 15.1(a). Thus, $a7.count$ is set to the sum of the counts of all preciously matched a's and b's. We further increment $a7.count$ by one since $a7$ is a START event. The count of type A is increased by $a7.count$ since $a7$ is of type A. The count of B is the final count.*

$$\begin{aligned}
a7.count &= 1 + A.count + B.count = 1 + 10 + 11 = 22.\\
A.count &= A.count + a7.count = 10 + 22 = 32.\\
b8.count &= A.count = 32.\\
B.count &= B.count + b8.count = 11 + 32 = 43.
\end{aligned}$$

Theorem 15.1 ***(Type-Grained Trend Count)****. Let query q be evaluated under the skip-till-any-match semantics and have no predicates on adjacent events. Let P be its pattern. Let $e \in I$ be an event of type E and $\mathcal{T} = P.predTypes(E)$. Then, the type-grained event*

Algorithm 7 Type-grained trend count algorithm for the skip-till-any-match semantics

```
Input: Query $q$ with pattern $P$, event stream $I$
Output: Count of event trends matched by $q$ in $I$
$H \leftarrow$ empty hash table
for all event type $E$ in $P$ do $H.put(E, 0)$
for all $e \in I$ of type $E$ do
    $e.count \leftarrow (E = start(P))\ ?\ 1\ :\ 0$
    for all $E' \in P.predTypes(E)$ do $e.count \mathrel{+}= H.get(E')$
    $E.count \leftarrow H.get(E) + e.count;\ H.put(E, E.count)$
return $H.get(end(P))$
```

trend count is computed as follows:

$$\begin{aligned} e.count &= \textstyle\sum_{E' \in \mathfrak{T}} E'.count. \\ E.count &= \textstyle\sum_{e.type=E} e.count. \\ final\_count &= end(P).count. \end{aligned}$$

If e is a START event, $e.count$ is incremented by one.

Proof: An event-grained trend count $e.count$ computed under the skip-till-any-match semantics is correct (Theorem 11.2). Without predicates on adjacent events, $e.count$ is set to the sum of event-grained trend counts of *all* previously matched events e' of a predecessor type E' of E under the skip-till-any-match semantics (Section 11.1), i.e., the sum of type-grained trend counts of all predecessor types E' of E. The first equation is proven.

A type-grained trend count $E.count$ for type E is computed based on event-grained trend counts of all matched events e of type E. Thus, $E.count$ is correct under the skip-till-any-match semantics. The second equation is proven.

In particular, a type-grained trend count for the end type of P is correct. The third equation is proven. ■

The Type-Grained Trend Count Algorithm consumes a query q and a stream I and returns the number of trends matched by q in I. Algorithm 7 maintains a hash table H that maps each type E in the pattern P to the count for E. Initially, all counts are set to 0 (Lines 1–2). For each event e of type E, $e.count = 1$ if E is a start type of P. Otherwise, $e.count = 0$ (Lines 3–4). For each predecessor type E' of E, $e.count$ is incremented by $E'.count$ (Line 5). $E.count$ is incremented by $e.count$ in the table H (Line 6). The count of the end type of P is returned (Line 7).

Theorem 15.2 *(**Complexity**). Let q be a query with pattern P of length l. Let n by the number of events per window of q. Algorithm 7 has linear time $O(nl)$ and space $\Theta(l)$ complexity.*

Proof: For each matched event of type E, the type-grained trend counts of all predecessor types of E are accessed. In the worst case, the counts of all types in P are accessed. Since $n > l$ for high-rate streams and meaningful patterns, the time complexity is linear in n: $\Theta(n) * O(l) = O(nl)$. The space complexity is determined by the number of counts. Since one count is stored per type, the space costs are linear in l: $\Theta(l)$. ■

15.2 Mixed-Grained Aggregator

In this section, we extend our coarse-grained trend aggregation techniques to a more general class of queries with *predicates on adjacent events* θ. To this end, we now propose to maintain aggregates at *mixed granularity levels*. Namely, we divide the event types $\mathcal{T}_P$ in the pattern P into two disjoint sets $\mathcal{T}_e$ and $\mathcal{T}_t$. Events of types $\mathcal{T}_e$ must be stored to evaluate the predicates θ as new events arrive. Thus, an *event-grained aggregate* is computed for each event of type in $\mathcal{T}_e$. In contrast, events of types $\mathcal{T}_t$ do not have to be kept. Thus, *type-grained aggregates* are maintained for each type in $\mathcal{T}_t$.

Example 15.2 *There are two sub-graphs for types A and B in Figure 15.1(a). If predicates θ restrict the adjacency relations between b's and a's, event-grained aggregates*

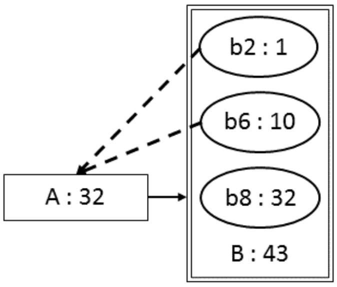


Figure 15.2: The type-granular count for type A and the event-granular counts for events of type B

must be maintained for b's. Indeed, when an a arrives, we have to compare it to each previously matched b to select those b's that satisfy the predicates θ, denoted $(b, a)\theta = \top$. In contrast, a type-grained count is maintained for type A. Assuming that $a7$ is adjacent to $b2$ and $b6$, $a7.count$ is computed based on the mixed-grained counts as follows:

$$\begin{aligned} a7.count &= A.count + \textstyle\sum_{(b,a7)\theta=\top} b.count \\ &= A.count + b2.count + b6.count = 22. \end{aligned}$$

Analogously, if predicates θ restrict the adjacency relation between a's and a's (or between a's and b's), a type-grained count is maintained for B, while event-grained counts are computed for a's.

Theorem 15.3 ***(Mixed-Grained Trend Count)****. Let query q be evaluated under the skip-till-any-match semantics, θ be its predicates on adjacent events, P be its pattern, attr and attr' be attributes of types E and E' in P respectively, and $\circ \in \{>, \geq, <, \leq, =, \neq\}$ be a comparison operator. A type-grained trend count is maintained for a type E if either there is no predicate of the form $(E.attr \circ E'.attr')$ in θ or $E \notin P.predTypes(E')$. Otherwise, an event-grained trend count is computed for each matched event of type E.*

Let $e \in I$ be an event of type E with $P.predTypes(E) = \mathcal{T}_t \uplus \mathcal{T}_e$ where $\mathcal{T}_t$ ($\mathcal{T}_e$) is the set of event types for which type-grained (event-grained) trend counts are maintained.

Algorithm 8 Mixed-grained trend count algorithm for the skip-till-any-match semantics

```
Input: Query $q$ with pattern $P$ and predicates $\theta$, stream $I$
Output: Count of event trends matched by $q$ in $I$
 1: $H \leftarrow$ empty hash table; $V \leftarrow \emptyset$; $final\_count \leftarrow 0$
 2: for all event type $E$ in $P$ do $H.put(E, 0)$
 3: for all $(E.attr\ Op\ E'.attr) \in \theta$ do
 4:     if $E \in P.predTypes(E')$ then $H.remove(E)$
 5: for all $e \in I$ of type $E$ do
 6:     $e.count \leftarrow (E = start(P))\ ?\ 1\ :\ 0$
 7:     for all $E' \in P.predTypes(E)$ do
 8:         if $E' \in H$ then $e.count$ += $H.get(E')$
 9:         else for each $e' \in V.predEvents(e)$ of type $E'$ do
10:             if $e'$ and $e$ satisfy $\theta$ then
11:                 $V \leftarrow V \cup e$; $e.count$ += $e'.count$
12:     if $E \in H$ then
13:         $E.count \leftarrow H.get(E) + e.count$; $H.put(E, E.count)$
14:     else if $E = end(P)$ then $final\_count$ += $e.count$
15: if $end(P) \in H$ then return $H.get(end(P))$
16: else return $final\_count$
```

Then, a mixed-grained trend count is computed as follows:

$$\begin{aligned} e.count \;=\; & \textstyle\sum_{E' \in \mathcal{T}_t} E'.count + \\ & \textstyle\sum_{e".type \in \mathcal{T}_e,\ (e",e)\theta=\top} e".count. \end{aligned}$$

If e *is a* START *event,* $e.count$ *is incremented by one.* $E.count$ *and* $final\_count$ *are computed as defined in Theorem 15.1.*

Proof: According to Theorems 15.1 and 11.3, both a type-grained trend count $E'.count$ and an event-grained trend count $e".count$ and are correct under the skip-till-any-match semantics. Since a mixed-grained trend count $e.count$ is computed based on these counts, $e.count$ is also correct under the skip-till-any-match semantics. ∎

The Mixed-Grained Trend Count Algorithm consists of two phases, namely, the static analysis and the runtime execution.

During the ***static analysis*** (Lines 1–4), for each type E in P, Algorithm 8 decides whether to maintain a single type-grained count $E.count$ or an event-grained count for each matched event of type E. Type-grained counts are maintained in a hash table H. Initially, counts for each type in P is set to 0 (Line 2). For each predicate that restricts the adjacency relation between events of type E and events of type E', if E is a predecessor type of E, then E is removed from the table H (Lines 3–4).

During the ***runtime execution*** (Lines 5–16), for each event e of type E, $e.count = 1$ if E is a start type of P. Otherwise, $e.count = 0$ (Lines 5–6). For each predecessor type E' of E, if a type-grained count $E'.count$ is maintained, $e.count$ is incremented by $E'.count$ (Lines 7–8). If event-grained counts for events of type E' are computed, $e.count$ is incremented by the count of each predecessor event e' of type E' that satisfies the predicates θ (Lines 9–11). If a type-grained count $E.count$ is maintained, $E.count$ is incremented by $e.count$ in the table H (Lines 12–13). If event-grained counts for events of type E are computed and E is an end type of P, then the final count is incremented by $e.count$ (Line 14). Lastly, if a type-grained count is maintained for the end type of P, it is returned as a result. Otherwise, $final\_count$ is returned (Lines 15–16).

Theorem 15.4 ***(Complexity)****. Let q be a query with pattern P of length l and n be the number of events per window of q. Let $\mathcal{T}_t$ ($\mathcal{T}_e$) be the set of event types for which type-grained (event-grained) trend counts are maintained. Let $t \leq l$ be the number of types in $\mathcal{T}_t$ and $n_e \leq n$ be the number of events of a type in $\mathcal{T}_e$ per window of q. Algorithm 8 has quadratic time $O(n(t + n_e))$ and linear space $\Theta(t + n_e)$ complexity.*

Proof: The time complexity of the static analysis is linear in l and the number of predicates θ. These values are negligible compared to the number of events n for high-rate streams and meaningful queries. During runtime execution, for each matched event $O(t)$ type-grained and $O(n_e)$ event-grained trend counts are accessed. Thus, the time complexity is quadratic in n: $O(n(t + n_e))$. The space complexity corresponds to the number of type-grained and event-grained trend counts: $\Theta(t + n_e)$. ■

16

Skip-Till-Next-Match and Contiguous Semantics

In this chapter, we define coarse-grained event trend aggregation under the skip-till-next-match and contiguous semantics. We first define baseline event-grained trend aggregation (Section 16.1). We then propose our COGRA approach that maintains aggregates at the pattern granularity level (Section 16.2).

16.1 Baseline Event-Grained Aggregator

Skip-Till-Next-Match Semantics does not skip relevant events in a trend (Section 2.4). Thus, only the last matched event e_l may connect to a new event e. Indeed, connecting an event earlier than e_l to e would mean that e_l would be skipped in a trend. Such execution strategy would produce wrong results by Definition 2.3.

Further, a new event trend can only start after the previous trend has ended. Otherwise, an event considered relevant in one trend would be skipped in the other trend. However, skipping relevant events is prohibited by Definition 2.3. Thus, a START event is matched only if the last event is not yet set or is an END event.

Example 16.1 *If the pattern P is evaluated under the skip-till-next-match semantics, the graph in Figure 15.1(c) is constructed. Now, only the last a or b connects to a new a. Analogously, the last a connects to a new b. Since less edges are drawn than in Figure 15.1(a), only eight trends are detected (Figure 2.3).*

Contiguous Semantics skips no event between matched events in a trend. Thus, an event e that cannot be matched discards all events matched so far from the graph. The final aggregate is not reset, however since it corresponds to the number of contiguous trends that had been matched before the event e arrived.

Example 16.2 *If the pattern P is evaluated under the contiguous semantics, the graph in Figure 15.1(e) is constructed. Since $c5$ cannot be matched, it disqualifies all previously matched events ($a1$–$a4$) from contributing to new contiguous trends to be formed in the future. These events are deleted. They are highlighted by a dark background in Figure 15.1(e). Since there is no trend that $b6$ can extend, $b6$ is not matched. Only two trends are detected (Figure 2.3).*

16.2 Pattern-Grained Aggregator

We now propose to compute trend aggregation at the coarsest possible *pattern granularity level*. Only two aggregates have to be maintained, namely, the intermediate count of the last matched event e_l and the final count (Figures 15.1(d) and 15.1(f)). Under these event matching semantics, predicates on adjacent events to not require fine-grained trend aggregation (Section 11.1 and Section 15.2). Indeed, since only e_l may connect to a new event e, predicates on adjacent events are evaluated based on e_l and e. Each matched END event e increments the final count by $e.count$.

Example 16.3 *In Figure 15.1(d), $a7$ propagates the count of the last event $b6$, adds one to it since $a7$ is a START event, and becomes the new last event. $b6$ is discarded. Since $b6$ is an END event, it increased the final count by $b6.count = 3$.*

Theorem 16.1 *(**Pattern-Grained Trend Count**). Let query q be evaluated under the skip-till-next-match or under the contiguous semantics and P be its pattern. The pattern-grained counts $e_l.count$ and $final\_count$ are maintained as follows.*

If a new event $e \in I$ a START event, $e.count = 1$. Otherwise, $e.count = 0$. If the last matched event e_l and e are adjacent in a trend (Definitions 2.3 and 2.4), then $e_l.count$ += $e.count$. The event e becomes the new last event.

The final count accumulates the counts of all matched END events, i.e., final_count $=$ $\sum_{e_l.type=end(P)} e_l.count$.

Proof: We prove the first equation by induction on the number of matched events n.

Induction Basis: $n = 1$. If only one event e is matched, e is a START event. Thus, $e.count = 1$. The event e becomes the last matched event e_l.

Induction Assumption: These equations hold for n matched events.

Induction Step: $n \rightarrow n + 1$. Assume a new event e is matched. According to the induction assumption, $e_l.count$ is correct, i.e., it corresponds to the number of sub-trends that end at the event e_l. If e_l and e are adjacent, e continues all these trends. If e is a START event, it begins a new trend and one is added to $e_l.count$. The event e becomes the last matched event e_l. The first equation is proven.

We count the number of finished trends only. By definition only END events may finish trends. Since the count of an END event end corresponds to the number of trends that finish at the event end, the total number of trends is the sum of counts of all matched END events. The second equation is proven. ■

The Pattern-Grained Trend Count Algorithm. Initially, the last matched event e_l is *null*, while the final count is set to 0 (Line 1). For a new matched event $e \in I$ of

Algorithm 9 Pattern-grained trend count algorithm for the skip-till-next-match and contiguous semantics

```
Input: Query $q$ with pattern $P$, event stream $I$
Output: Count of event trends matched by $q$ in $I$
 1: $e_l \leftarrow null$; $final\_count \leftarrow 0$
 2: for all $e \in I$ of type $E$ do
 3:     if $isMatched(e_l, e)$ then
 4:         $e.count \leftarrow (E = start(P))\ ?\ 1\ :\ 0$
 5:         if $isAdjacent(e_l, e)$ then $e_l.count$ += $e.count$
 6:         else $e_l.count = e.count$
 7:         $e_l \leftarrow e$
 8:         if $E = end(P)$ then $final\_count$ += $e_l.count$
 9:     else if $q.semantics = contiguous$ then $e_l \leftarrow null$
10: return $final\_count$
```

type E, $e.count = 1$ if e is a START event. Otherwise, $e.count = 0$ (Lines 2–4). An event e is matched if one of the following conditions holds: (1) The events e_l and e are adjacent (Definitions 2.3 and 2.4). In this case, the count of e_l is increased by the count of e (Line 5). (2) The events e_l and e are not adjacent, e is a START event, and e_l is either null or an END event. Then, the count of e_l is set to the count of e (Line 6). The event e becomes the new last event (Line 7). If e is an END event, the final count is increased by the count of e_l (Line 8). If e cannot be matched and the query q detects contiguous trends, then e_l is set to null (Line 9). The final count is returned (Line 10).

Theorem 16.2 *(**Complexity**). Let q be a query evaluated under the skip-till-next-match or contiguous semantics and n be the number of events per window of q. Algorithm 9 has linear time $\Theta(n)$ and constant space $O(1)$ complexity.*

Proof: The time complexity is determined by the number of matched events, i.e., $\Theta(n)$. The space complexity is constant since two aggregates are stored per pattern. ■

17

Implementation

Cogra Runtime Data Structures must support event trend aggregation at different granularity levels.

Event-grained aggregator. If aggregates are maintained per event, we utilize the GRETA the data structures described in Section 11.4.

Type-grained aggregator. If aggregates are maintained per event type (Section 15.1), each type in the ***Type Hash Table*** is mapped directly to the ***Results Hash Table*** (Figure 17.1(b)) since the aggregate of the end type per window corresponds to the final result for a window.

Mixed-grained aggregator. If mixed-grained aggregates are computed (Section 15.2), each type E in the ***Type Hash Table*** is either mapped to its respective ***Event Tree*** if an aggregate is maintained per each event of type E (Figure 17.1(a)) or directly to the ***Results Hash Table*** if a single aggregate is computed for the type E (Figure 17.1(b)).

Pattern-grained aggregator. If aggregates are computed at pattern level (Section 16.2), each pane in the ***Time Pane Array*** is mapped directly to the ***Results Hash Table*** (Figure 17.1(c)) that stores the last matched event and the final aggregate per window.

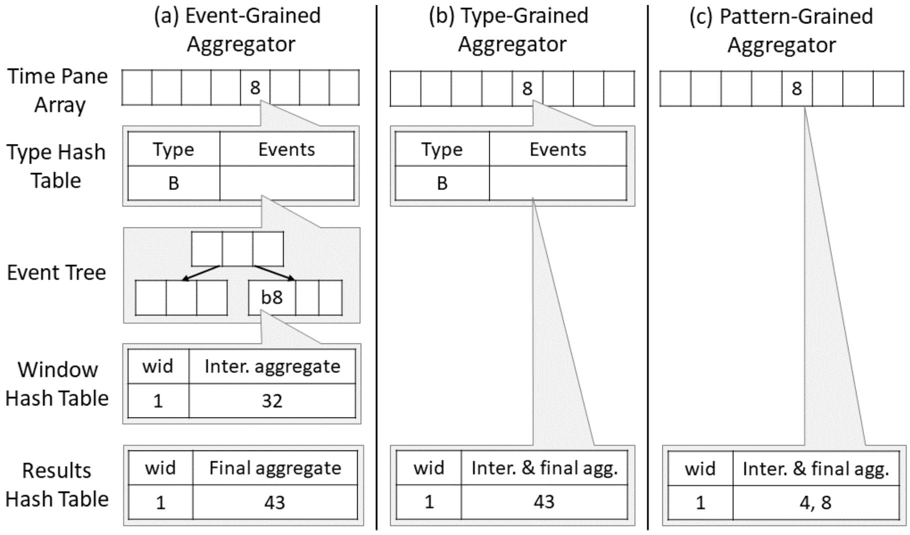


Figure 17.1: COGRA runtime data structures

18

Other Aggregation Functions

While so far we have focused on event trend count computation, i.e., COUNT(*), we now sketch how the principles of coarse-grained online trend aggregation apply to other aggregation functions (Definition 2.5). Table 18.1 defines COUNT(E), MIN$(E.attr)$, and SUM$(E.attr)$ at different granularity levels. In contrast to COUNT$(*)$, only matched events e of type E update the aggregates. All other matched events x of type $X \neq E$ propagate the aggregates from previous to more recent events in a trend matched by a pattern P (or event types in P). MAX$(E.attr)$ is maintained analogously to MIN$(E.attr)$, while AVG$(E.attr)$ is computed based on SUM$(E.attr)$ and COUNT(E) (Definition 2.5).

Type-grained aggregates	**Mixed-grained aggregates**	**Pattern-grained aggregates**
e.count$_E$ $=$ $e.count+$ $\sum_{E'\in\mathcal{T}} E'.count_E$ ***x.count***$_E$ $=$ $\sum_{X'\in\mathcal{Y}} X'.count_E$	***e.count***$_E$ $=$ $e.count+$ $\sum_{E'\in\mathcal{T}_t} E'.count_E+$ $\sum_{e”.type\in\mathcal{T}_e,\ (e”,e)\theta=\top} e”.count_E$ ***x.count***$_E$ $=$ $\sum_{X'\in\mathcal{Y}_t} X'.count_E+$ $\sum_{x”.type\in\mathcal{Y}_e,\ (x”,x)\theta=\top} x”.count_E$	***e***$_l$***.count***$_E$ $\mathrel{+}=$ $e.count$
T.count$_E$ $= \sum_{t.type=T} t.count_E$ ***COUNT(E)*** $= end(P).count_E$		***COUNT(E)*** $=$ $\sum_{e_l.type=end(P)} e_l.count_E$
e.min $=$ $\min_{E'\in\mathcal{T}}$ $(e.attr, E'.min)$ ***x.min*** $=$ $\min_{X'\in\mathcal{Y}}(X'.min)$	***e.min*** $=$ $\min_{E'\in\mathcal{T}_t,\ e”.type\in\mathcal{T}_e,\ (e”,e)\theta=\top}$ $(e.attr, E'.min, e”.min)$ ***x.min*** $=$ $\min_{X'\in\mathcal{Y}_t,\ x”.type\in\mathcal{Y}_e,\ (x”,x)\theta=\top}$ $(X'.min, x”.min)$	***e***$_l$***.min*** $=$ $\min(e.attr, e_l.min)$
T.min $= \min_{t.type=T}(t.min)$ ***MIN(E.attr)*** $= end(P).min$		***MIN(E.attr)*** $=$ $\min_{e_l.type=end(P)}(e_l.min)$
e.sum $=$ $e.attr * e.count+$ $\sum_{E'\in\mathcal{T}} E'.sum$ ***x.sum*** $=$ $\sum_{X'\in\mathcal{Y}} X'.sum$	***e.sum*** $=$ $e.attr * e.count+$ $\sum_{E'\in\mathcal{T}_t} E'.sum+$ $\sum_{e”.type\in\mathcal{T}_e,\ (e”,e)\theta=\top} e”.sum$ ***x.sum*** $=$ $\sum_{X'\in\mathcal{Y}_t} X'.sum+$ $\sum_{x”.type\in\mathcal{Y}_e,\ (x”,x)\theta=\top} x”.sum$	***e***$_l$***.sum*** $\mathrel{+}=$ $e.attr * e.count$
T.sum $= \sum_{t.type=T} t.sum$ ***SUM(E.attr)*** $= end(P).sum$		***SUM(E.attr)*** $=$ $\sum_{e_l.type=end(P)} e_l.sum$

Table 18.1: Coarse-grained event trend aggregation ($e, x, e”, x”, t \in I$ are matched events, $e.type = E$, $x.type = X \neq E$, T is any event type in P, $P.predTypes(E) = \mathcal{T} = \mathcal{T}_t \dot{\cup} \mathcal{T}_e$ (as defined in Theorem 15.3), and $P.predTypes(X) = \mathcal{Y} = \mathcal{Y}_t \dot{\cup} \mathcal{Y}_e$)

19

Performance Evaluation

19.1 Experimental Setup

Infrastructure. We have implemented our approach in Java with JRE 1.7.0_25 running on Ubuntu 14.04 with 16-core 3.4GHz CPU and 128GB of RAM. We execute each experiment three times and report their average results here.

Methodology. We demonstrate the effectiveness of COGRA by comparing it to Flink [30], SASE [3], A-Seq [2], and GRETA [6] since they cover the spectrum of state-of-the-art event aggregation approaches (Table 1.1). We run Flink on the same hardware as our platform. To achieve a fair comparison, we implemented SASE, A-Seq, and GRETA on top of our platform. While Chapter 20 is devoted to a detailed discussion of these approaches, we summarize them in Table 19.1.

• ***Flink*** [30] is a popular open-source streaming platform that supports event pattern matching. We express our queries using Flink operators such as event sequence, window, grouping. Similarly to other industrial systems [31, 32], Flink does not explicitly support Kleene closure. Thus, we flatten our queries as follows. For each Kleene pattern P, we determine the length l of the longest match of P. We specify a set of fixed-length event

Approach		Kleene closure	Event matching semantics			Predicates on		Event trend grouping
			ANY	NEXT	CONT	single events	adjacent events	
Two-step	Flink	−	+	−	+	+	+	+
	SASE	+	+	+	+	+	+	+
On-line	GRETA	+	+	−	−	+	+	+
	A-Seq	−	+	−	−	+	−	+
	COGRA	+	+	+	+	+	+	+

Table 19.1: Expressive power of the event aggregation approaches

sequence queries that cover all possible lengths up to l. Flink supports the skip-till-any-match and contiguous semantics. It implements a two-step approach that constructs all event sequences prior to their aggregation.

• ***SASE*** [3] supports Kleene closure and all event matching semantics. It implements the two-step approach. Namely, it first stores each event e in a stack and computes the pointers to e's previous events in a trend. For each window, a DFS-based algorithm traverses these pointers to construct all trends. Then, these trends are aggregated.

• **GRETA** [6] captures all matched events and the trend relationships among them as a graph. Based on the graph, it computes event trend aggregation online, that is, it avoids event trend construction (Chapters 10–13). It supports only the skip-till-any-match semantics.

• ***A-Seq*** [2] avoids event sequence construction by dynamically maintaining a count for each prefix of a pattern. Since A-Seq does not support Kleene closure, we flatten our queries as described above. A-Seq supports only the skip-till-any-match semantics. It does not support arbitrary predicates on adjacent events beyond equivalence predicates, such as $[patient]$ in query q' in Section 1.1.

Data Sets. We compare our COGRA approach to the state-of-the-art techniques using the following data sets.

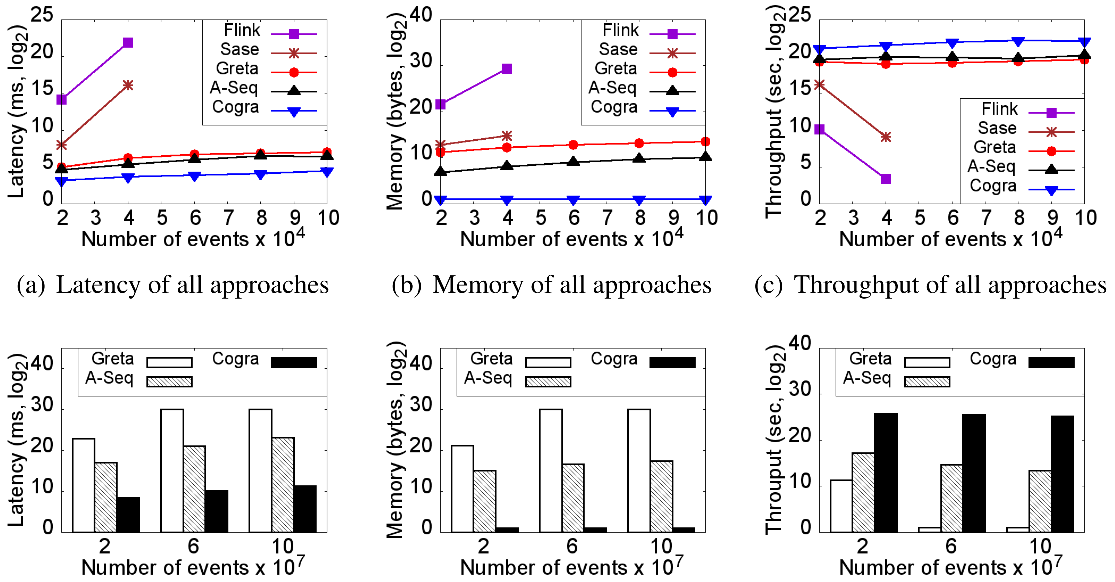
• ***Physical Activity Monitoring Real Data Set*** [42] contains physical activity reports for 14 people during 1 hour 15 minutes. 18 activities (e.g., walking, driving) are considered. A report carries time stamp in seconds, person identifier, activity identifier, heart rate, etc. The size of the data set is 1.6GB.

• ***Stock Real Data Set*** [43] contains 225k transaction records of 19 companies in 10 sectors. Each event carries time stamp in seconds, company identifier, sector identifier, transaction identifier, transaction type (sell or buy), volume, price, etc. We replicate this data set 10 times.

• ***Public Transportation Synthetic Data Set***. Our stream generator creates trips for 30 passengers using public transportation services in a metropolitan area with 100 stations. Each event carries a time stamp in seconds, passenger identifier, station identifier, and waiting time in seconds. Waiting durations are generated uniformly at random.

Event Queries. We evaluate queries Q_1–Q_3 [6] against the physical activity monitoring, public transportation, and stock data sets respectively. We vary event matching semantics, number of events per window, predicate selectivity, and number of trend groups. Unless stated otherwise, we evaluate our queries under the skip-till-any-match semantics since all state-of-the-art approaches support this semantics (Table 19.1). To ensure that the two-step approaches terminate at least in some cases, the default number of events per window is 50k. Since A-Seq does not support arbitrary predicates on adjacent events, we evaluate our queries without such predicates by default. Unless stated otherwise, the number of trend groups is 14 for the physical activity data set, 19 for the stock data set, and 30 for the public transportation data set.

Metrics. We measure the common metrics for streaming systems, namely, latency, throughput, and peak memory. ***Latency*** is measured in milliseconds as the average time difference between the time of the aggregation result output and the arrival time of the latest event that contributes to the respective result. ***Throughput*** corresponds to the average number of events processed by all queries per second. ***Peak memory*** includes the mem-



(a) Latency of all approaches (b) Memory of all approaches (c) Throughput of all approaches

(d) Latency of online approaches (e) Memory of online approaches (f) Throughput of online appr.

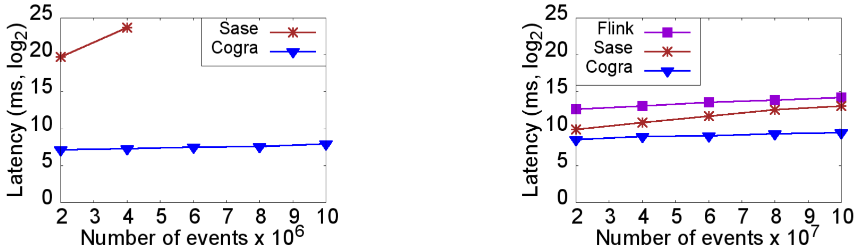
Figure 19.1: Skip-till-any-match semantics (Stock real data set)

ory for storing the aggregates and sub-graphs for COGRA, the GRETA graph for GRETA, prefix counters for A-Seq, events in stacks, pointers between them, and trends for SASE, and trends for Flink.

19.2 Event Matching Semantics

In Figures 19.1–19.2(b), we compare the performance of COGRA to the state-of-the-art approaches under diverse event matching semantics, while varying the number of events per window. If an approach does not support a semantics (Table 19.1), the approach is not shown in the chart for this semantics.

Two-Step Approaches perform well for high-rate streams only under the most restrictive contiguous semantics (Figure 19.2(b)) because the number and length of contiguous trends are relatively small. Nevertheless, COGRA achieves 27–fold speed up compared to Flink and 12–fold speed up compared to SASE when the number of events per window



(a) Skip-till-next-match semantics (Public transport data set)

(b) Contiguous semantics (Physical activity real data set)

Figure 19.2: Skip-till-next-match and contiguous semantics

reaches 100 million.

Flink is not optimized for trend aggregation for two reasons. One, Flink must evaluate a workload of event sequence queries for each Kleene query. Two, Flink first constructs all event sequences and then aggregates them. Under the skip-till-any-match semantics, the latency and memory of Flink grow exponentially in the number of events, while its throughput decreases exponentially (Figures 19.1(a)–19.1(c)). Flink does not terminate when the number of events exceeds 40k. For 40k events, COGRA achieves 4 orders of magnitude speed-up and uses 8 orders of magnitude less memory than Flink.

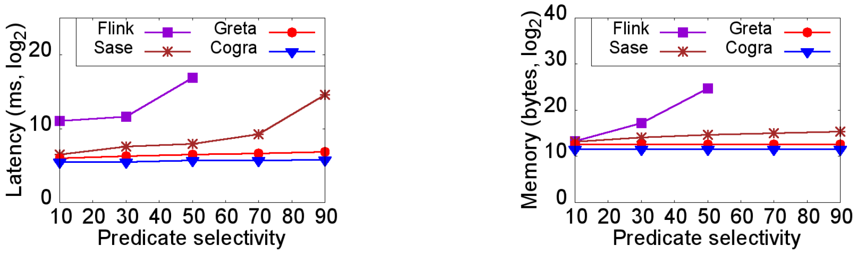
SASE supports Kleene patterns but also implements a two-step approach. Under skip-till-any-match, the latency and memory usage of SASE grow exponentially in the number of events, while its throughput degrades exponentially (Figures 19.1(a)–19.1(c)). SASE fails to terminate when the number of events exceeds 40k. For 40k events, COGRA achieves 3 orders of magnitude speed-up and 4 orders of magnitude memory reduction compared to SASE. Even under skip-till-next-match, SASE does not terminate if a window contains over 4 million events (Figure 19.2(a)). For 4 million events, SASE produces aggregation results with an over 3 hours long delay – which is unacceptable for time-critical applications. COGRA achieves 4 orders of magnitude speed-up and 5 orders of magnitude memory reduction compared to SASE in this case.

Online Approaches perform similarly for low-rate streams (Figures 19.1(a)–19.1(c)), while high-rate streams reveal the difference between them (Figures 19.1(d)–19.1(f)).

GRETA captures all matched events and their trend relationships as a graph. While the overhead of graph construction is negligible for low-rate streams, it becomes a bottleneck when the stream rate increases. Under skip-till-any-match, GRETA does not terminate if the stream rate is higher than 20 million events per window (Figure 19.1(d)). For 20 million events, GRETA suffers from over an hour long delay. Its latency is 4 orders of magnitude higher compared to COGRA in this case.

A-Seq performs best among the state-of-the-art approaches. However, its expressive power is limited (Table 19.1). Also, A-Seq must evaluate a workload of event sequence queries for each Kleene pattern. The number of queries grows linearly in the number of events. Due to this large workload, the latency of A-Seq is 3 orders of magnitude higher compared to COGRA when the number of events reaches 100 million (Figure 19.1(d)). Each event sequence query maintains a fixed number of aggregates [2]. Thus, the memory usage of A-Seq grows linearly with the number of queries (i.e., with the number of events). A-Seq requires 4 orders of magnitude more memory than COGRA for 100 million events (Figure 19.1(e)).

COGRA performs well for all semantics and stream rates because COGRA maintains a fixed number of aggregates per Kleene pattern. Its memory usage is constant (Figures 19.1(b) and 19.1(e)). The latency of COGRA grows linearly in the number of events. For 100 million events per window, the latency of COGRA stays within 3 seconds (Figure 19.1(d)), while its throughput is over 39 million events per second (Figure 19.1(f)). COGRA enables real-time and in-memory trend aggregation.



(a) Latency (b) Memory

Figure 19.3: Predicate selectivity (Stock real data set)

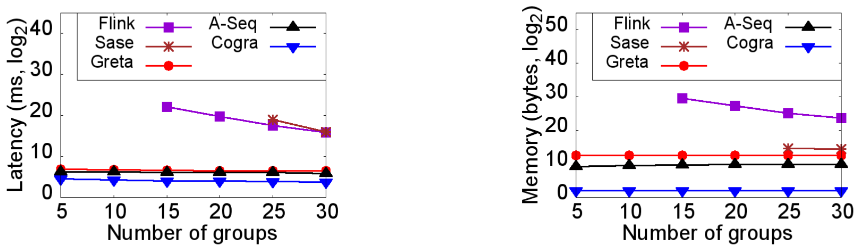
19.3 Predicate Selectivity

In Figure 19.3, we focus on the selectivity of predicates on adjacent events, while predicates on single events are evaluated when varying the number of trend groups (Section 19.4) [2, 6]. To ensure that the two-step approaches terminate in most cases, we run this experiment against a low-rate stream of 50k events per window. Since A-Seq does not support expressive predicates on adjacent events, it is not evaluated here.

Two-Step Approaches. When the predicate selectivity increases, more and longer trends are constructed and stored by ***Flink***. In fact, its latency and memory usage grow exponentially (Table 2.1, Figures 19.3(a)–19.3(b)). Flink fails to terminate once the predicate selectivity exceeds 50%. When the predicate selectivity is 50%, COGRA achieves 3 orders of magnitude speed-up and 3 orders of magnitude memory reduction compared to Flink.

SASE constructs all trends. Thus, its latency grows exponentially. Only the current trend is stored however. The number of stored pointers between events increases when the predicate selectivity grows. Thus, the memory costs grow linearly. The latency of COGRA is 2 orders of magnitude lower, while it uses 13–fold less memory than SASE for 90% predicate selectivity.

Online Approaches perform well for such low-rate stream of 50k events per window. When the predicate selectivity increases, **GRETA** stores the same events in the GRETA



(a) Latency

(b) Memory

Figure 19.4: Number of trend groups (Public transport data set)

graph but the number of edges between them increases. Thus, latency increases linearly in the number of edges. Since edges are not stored, the memory consumption remains stable.

Similarly, the number of aggregates maintained by **COGRA** stays the same with the growing predicate selectivity. Since COGRA maintains aggregates per event type (not per event), it achieves double speed-up and memory reduction compared to GRETA when the predicate selectivity reaches 90%.

19.4 Number of Event Trend Groups

In Figure 19.4, we vary the number of trend groups. To ensure that the two-step approaches terminate in most cases, we run this experiment against a low-rate stream of 50k events per window. When the number of groups increases, less events fall into each group. Thus, the latency of all approaches reduces with the increasing number of trend groups.

Two-Step Approaches. Since trends are constructed per group, the number and length of trends decrease with the growing number of groups. Thus, the latency and memory costs of ***Flink*** decrease exponentially when the number of groups increases. Flink fails to terminate when the number of groups is fewer than 15. For 15 groups, COGRA wins 5 orders of magnitude with respect to latency and 8 orders of magnitude regarding memory usage compared to Flink.

SASE constructs all trends without storing them. Thus, its latency reduces exponentially, while its memory consumption decreases linearly with the growing number of groups. SASE does not terminate for fewer than 25 groups. For 25 groups, COGRA achieves 4 orders of magnitude speed-up and 3 orders of magnitude memory reduction compared to SASE.

Online Approaches performs well independently from the number of groups since trends are not constructed. The latency of ***A-Seq*** reduces linearly when the number of groups increases. Since A-Seq evaluates a workload of event sequence queries for each Kleene query, its latency is 5–fold higher than the latency of COGRA for 5 groups. A-Seq maintains aggregates per group. Thus, its memory costs increase linearly in the number of groups. COGRA wins 2 orders of magnitude regarding memory compared to A-Seq for 30 groups.

The latency of **GRETA** decreases linearly with the increasing number of groups. Due to the GRETA graph construction overhead, the latency of GRETA is 7–fold higher than the latency of COGRA for 5 groups. The memory usage of GRETA remains stable when varying the number of groups because the same number of events is stored in the GRETA graphs. The memory costs of GRETA are 3 orders of magnitude higher compared to COGRA in all cases.

20

Related Work

Complex Event Processing approaches such as SASE [3, 7], Cayuga [28], ZStream [29], and E-Cube [4] support aggregation computation over streams. SASE and Cayuga deploy a Finite State Automaton-based query execution paradigm. ZStream translates an event query into an operator tree optimized using rewrite rules. E-Cube employs hierarchical event stacks to share events across different event queries. However, the expressive power of all these approaches is limited. Cayuga, ZStream, and E-Cube do not support rich event matching semantics. Cayuga and ZStream do not support GROUP-BY clause, while E-Cube does not support Kleene closure in their event query languages. Since these approaches do not design any optimization techniques for trend aggregation, they require trend construction prior to trend aggregation. Due to the exponential time complexity of trend construction (Table 2.1), these two-step approaches fail to respond within a few seconds (Chapter 19).

In contrast to these approaches, A-Seq [2] proposes *online* aggregation of *fixed-length* event sequences under the skip-till-any-match semantics. However, it supports neither Kleene closure, nor arbitrarily nested event patterns, nor other event matching semantics, nor expressive predicates on adjacent events. Thus, A-Seq does not tackle the exponential complexity of event trends.

Traditional Data Streaming approaches [33, 34, 35, 36, 37, 67, 71, 72] support aggregation over data streams. Some incrementally aggregate *raw input events for single-stream queries* [36, 67]. Others share aggregation results among overlapping sliding windows [33, 36] or multiple queries [35, 71, 72]. However, these approaches are restricted to Select-Project-Join queries with window semantics. That is, their execution paradigm is set-based. They support neither event matching semantics nor CEP-specific operators such as event sequence and Kleene closure that treat the order of events as first-class citizens. These approaches require the *construction of join results* prior to their aggregation. Thus, they define incremental aggregation of *single raw events* but implement a two-step approach for join results.

Industrial streaming systems such as Flink [30], Esper [31], and Oracle Stream Analytics [32] support fixed-length event sequences. They support neither Kleene closure nor various event matching semantics. While Kleene closure computation can be simulated by a set of event sequence queries covering all possible lengths of a trend, this approach is possible only if the maximal length of a trend is known apriori. This is rarely the case in practice. Furthermore, this approach is highly inefficient for two reasons. One, it must execute a set of event sequence queries for each Kleene query. This increased workload degrades the system performance. Two, since this approach requires trend construction prior to their aggregation, it has exponential time complexity.

Static Sequence Databases extend traditional SQL queries by order-aware join operations and support aggregation of their results [18, 19]. However, they do not support Kleene closure. Instead, *single data items* are aggregated [18, 73, 74, 75]. They also do not support event matching semantics. Lastly, these approaches assume that the data is statically stored and indexed prior to processing. Hence, they do not tackle challenges that arise due to dynamically streaming data such as event expiration and real-time execution.

Part III

Shared Event Sequence Aggregation

21

Sharon Approach Overview

In this chapter, we share online aggregation of event queries with event sequence patterns (Definition 2.1) evaluated under the most flexible skip-till-any-match semantics (Section 2.4). These queries do not support predicates on adjacent events (Section 2.5). To simplify our discussion, we assume that: (1) All queries in the workload have the same predicates, grouping, and windows. (2) A sub-pattern p is shared among all queries containing p. In Chapter 24, we sketch straightforward extensions of our approach to relax these assumptions.

We target *time-critical* applications that require event sequence aggregation results within a few seconds (Section 1.1). *Latency* corresponds to the average time difference between the time point of the aggregation result output by a query in the workload and the arrival time of the latest event that contributed to this result. Given a query workload Q and an event stream I, our ***Multi-query Event Sequence Aggregation (MESA) Problem*** is to determine which queries share the aggregation of which patterns (i.e., a sharing plan $\mathcal{P}$) such that the *latency of evaluating the workload Q according to the plan $\mathcal{P}$ against the stream I is minimal.*

To solve this problem, our SHARON framework deploys the following components

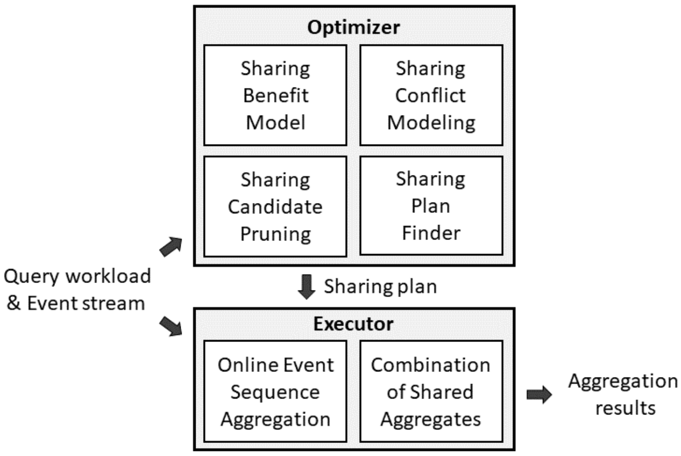

Optimizer
Sharing Benefit Model
Sharing Conflict Modeling
Sharing Candidate Pruning
Sharing Plan Finder
Query workload & Event stream
Sharing plan
Executor
Online Event Sequence Aggregation
Combination of Shared Aggregates
Aggregation results

Figure 21.1: SHARON framework

(Figure 21.1). For a given workload, our **SHARON Optimizer** identifies sharing candidates of the form (p, Q_p) where p is a pattern that could potentially be shared by a set of queries Q_p. It then estimates the benefit of each candidate (p, Q_p) (Chapter 22), determines sharing conflicts among these candidates, and compactly encodes all candidates, their benefits and conflicts into a SHARON graph (Section 23.1). Based on the graph, the optimizer prunes large portions of the search space (Section 23.2) and returns an optimal sharing plan (Section 23.3). Based on this plan, our **SHARON Executor** first computes the aggregation results for each shared pattern and then combines these shared aggregations to obtain the final results for each query in the workload (Chapter 22).

22

Sharing Benefit Model

Our optimizer first identifies sharing candidates in a workload (Section 22.1). Our executor leverages A-Seq [2] to compute the aggregation results of each query using either the *Not-Shared method* (Section 22.2) or the *Shared method* (Section 22.3). Lastly, the optimizer estimates the benefit of sharing each candidate (Section 22.4).

22.1 Sharing Candidate

First, our optimizer identifies those patterns that could potentially be shared by queries in a given workload.

Definition 22.1 ***(Sharable Pattern, Sharing Candidate**.) Let Q be a workload and p be a pattern that appears in queries $Q_p \subseteq Q$. The pattern p is sharable in Q if $p.length > 1$ and $|Q_p| > 1$. A sharable pattern p and queries Q_p constitute a sharing candidate, denoted as (p, Q_p).*

Existing pattern mining approaches can detect sharable patterns [76].

The pattern of a query $q_i \in Q_p$ consists of three sub-patterns, namely, prefix^i, p, and suffix^i (Figure 22.1).

$$\mathrm{P}^{\mathrm{i}} = \underbrace{\mathrm{E}_1^{\mathrm{i}} \dots \mathrm{E}_{\mathrm{m}-1}^{\mathrm{i}}}_{\mathrm{prefix}^{\mathrm{i}}} \underbrace{\mathbf{E_m} \dots \mathbf{E_{m+l}}}_{\mathbf{p}} \underbrace{\mathrm{E}_{\mathrm{m+l+1}}^{\mathrm{i}} \dots \mathrm{E}_{\mathrm{n}}^{\mathrm{i}}}_{\mathrm{suffix}^{\mathrm{i}}}$$

Figure 22.1: Sub-patterns of q_i

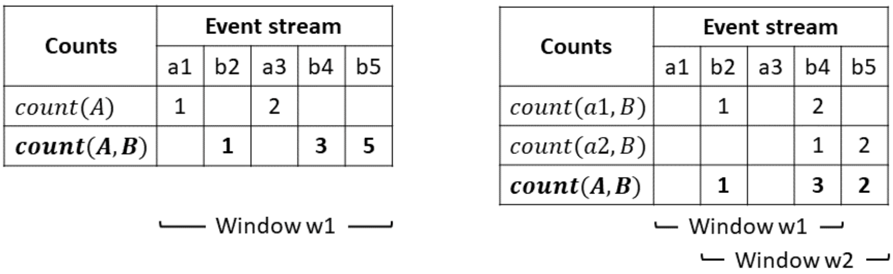
22.2 Not-Shared Method

While A-Seq considers grouping, predicates, negation, and various aggregation functions, below we sketch only its key ideas, namely, online event sequence aggregation and event sequence expiration. We use event sequence count as an example, i.e., COUNT(*) (Definition 2.5).

Online Event Sequence Aggregation A-Seq computes the count of event sequences online, i.e., without constructing and storing these sequences. To this end, it maintains a count for each prefix of a pattern. The count of a prefix of length j is incrementally computed based on the count of the prefix of length $j - 1$ and the previous value of the count of the prefix of length j.

Example 22.1 *Let an event be described by its type and time stamp in seconds, e.g., a_1 is an event of type A with time stamp 1. In Figure 22.2(a), we count event sequences matched by the pattern (A, B), denoted $count(A, B)$. A count is maintained for each prefix of the pattern, i.e., for (A) and (A, B). The value of $count(A, B)$ is updated every time a b arrives by summing the previous values of $count(A)$ and $count(A, B)$. For example, when b_4 arrives, it is appended to each previously matched a to form new sequences (a_1, b_4) and (a_2, b_4) (Definition 2.1). The number of new sequences corresponds to $count(A) = 2$. In addition, the previously formed sequence (a_1, b_2) is kept. The number of previous sequences corresponds to $count(A, B) = 1$. In sum, the value of $count(A, B)$ is updated to 3.*

Event Sequence Expiration. Due to the sliding window semantics of our queries



Counts	Event stream				
	a1	b2	a3	b4	b5
$count(A)$	1		2		
$\boldsymbol{count(A,B)}$		**1**		**3**	**5**

Window w1

(a) Online sequence aggregation

Counts	Event stream				
	a1	b2	a3	b4	b5
$count(a1,B)$		1		2	
$count(a2,B)$				1	2
$\boldsymbol{count(A,B)}$		**1**		**3**	**2**

Window w1
Window w2

(b) Event sequence expiration

Figure 22.2: Not-Shared method

(Definition 2.5), event sequences expire over time. To avoid the re-computation of all affected aggregates, we observe that a START event of a sequence (Definition 2.2) expires sooner than any other event in it. Thus, we maintain the aggregates per each matched START event. When a new event arrives, only the counts of not-expired START events are updated. When an END event e arrives, it updates the final counts for all windows that e falls into.

Example 22.2 *In Figure 22.2(b), assume a window of length four seconds slides every second. A count is now maintained per each matched* a *as described above. When* b_5 *arrives,* a_1 *is expired. Thus,* b_5 *disregards* $count(a_1, B)$*. The event* b_5 *updates* $count(a_2, B)$ *and* $count(A, B)$ *for window* w_2*.*

Time Complexity. The query q_i processes each event that it matches. The rate of matched events is computed as the sum of rates of all event types in the pattern P^i of q_i (Figure 22.1):

$$Rate(P^i) = \sum_{j=1}^{n} Rate(E^i_j) \tag{22.1}$$

Since counts are maintained per START event and an event type appears at most once in a pattern, each matched event updates one count per each not-expired START event.

There are $Rate(E_1^i)$ START events. In summary, the time complexity of processing the query q_i by the Not-Shared method is:

$$NotShared(p, q_i) = Rate(E_1^i) \times Rate(P^i) \tag{22.2}$$

For the set of queries Q_p, the time complexity corresponds to the summation of the time complexity for each query q_i.

$$NotShared(p, Q_p) = \sum_{q_i \in Q_p} NotShared(p, q_i) \tag{22.3}$$

The pattern p is computed once by each query $q_i \in Q_p$. The time complexity of the re-computation caused by each query q_i is determined as follows.

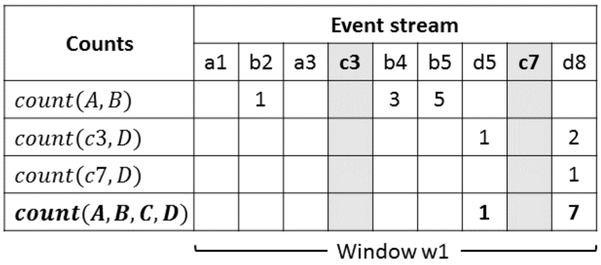
$$Recomp(p, q_i) = Rate(E_1^i) \times Rate(p) \tag{22.4}$$

The overall re-computation overhead corresponds to the summation of overhead caused by each query $q_i \in Q_p$.

$$Recomp(p, Q_p) = \sum_{q_i \in Q_p} Recomp(p, q_i) \tag{22.5}$$

22.3 Shared Method

Let p_1 and p_2 be patterns whose aggregates are shared. These aggregates are combined to obtain the aggregate for the pattern (p_1, p_2). Due to event sequence semantics, the executor must guarantee that the sequences matched by p_1 appear before the sequences matched by p_2 in the event stream (Definition 2.1). To this end, the executor performs two steps:



Counts	Event stream								
	a1	b2	a3	**c3**	b4	b5	d5	**c7**	d8
$count(A,B)$		1			3	5			
$count(c3,D)$							1		2
$count(c7,D)$									1
$\boldsymbol{count(A,B,C,D)}$							**1**		**7**

Figure 22.3: Shared method

1) ***Count computation***. We compute the counts of p_1 and p_2 as defined in Section 22.2.

2) ***Count combination***. We multiply $count(p_1)$ with the count for each START event of p_2. The resulting counts are summed to obtain $count(p_1, p_2)$ that now can be combined with counts of other patterns.

Example 22.3 *In Figure 22.3, we compute the count of (A, B, C, D) based on the counts of (A, B) and (C, D). Assuming that events a_1–d_8 belong to the same window, the count for (A, B) is computed as shown in Figure 22.2(a). In addition, a count for each c (i.e., c_3 and c_7) is maintained. When c_3 arrives, $count(A, B) = 1$. We multiply it with $count(c_3, D) = 2$ to obtain $count(A, B, c_3, D) = 2$. Analogously, when c_7 arrives, $count(A, B) = 5$. It is multiplied with $count(c_7, D) = 1$ to get $count(A, B, c_7, D) = 5$. Lastly, we sum these counts to obtain $count(A, B, C, D) = 7$ and store it for further reference.*

Time Complexity. 1) ***Count computation***. Counts are maintained per each START event of $prefix^i$, p, and $suffix^i$. Since the shared sub-pattern p is processed once for all queries in Q_p, the time complexity of computing the count for each sub-pattern of q_i by the Shared method corresponds to the sum of the time complexity of processing $prefix^i$ and $suffix^i$.

$$Comp(p, q_i) = Rate(E^i_1) \times Rate(prefix^i) + Rate(E^i_{m+l+1}) \times Rate(suffix^i) \qquad (22.6)$$

2) ***Count combination***. The time complexity of count multiplication is the product of the number of counts.

$$Comb(p, q_i) = Rate(E_1^i) \times Rate(E_m) \times Rate(E_{m+l+1}^i) \tag{22.7}$$

The time complexity of processing q_i by the Shared method is the sum of the time complexity of these two steps.

$$Shared(p, q_i) = Comp(p, q_i) + Comb(p, q_i) \tag{22.8}$$

For the set of queries Q_p, the time complexity corresponds to the summation of time complexity for each query q_i. In contrast to the Not-Shared method (Equation 22.3), the pattern p is computed once by the Shared method.

$$Shared(p, Q_p) = Rate(E_m) \times Rate(p) + \sum_{q_i \in Q_p} Shared(p, q_i) \tag{22.9}$$

On the down side, the CPU overhead caused by the count combination is considerable if the number of counts is large.

$$Comb(p, Q_p) = \sum_{q_i \in Q_p} Comb(p, q_i) \tag{22.10}$$

22.4 Benefit of a Sharing Candidate

Definition 22.2 *(**Benefit of a Sharing Candidate**.) The benefit of sharing a pattern p by the set of queries Q_p is computed as the difference between the time complexity of the*

Not-Shared and Shared methods (Equations 22.3 and 22.9):

$$BValue(p, Q_p) = NotShared(p, Q_p) - Shared(p, Q_p) \tag{22.11}$$

A sharing candidate (p, Q_p) *is called beneficial if* $BValue(p,\ Q_p) > 0$*. It is called non-beneficial otherwise.*

Non-Beneficial Candidate Pruning. All non-beneficial candidates are pruned from further analysis.

Based on this cost model, we conclude that the following three factors determine the benefit of sharing: *the number of queries, the length of their patterns*, and *the stream rate*. We experimentally study the effect of these factors in Chapter 25.

23

Sharon Optimizer

23.1 Sharing Conflict Modeling

A decision to share a pattern p by a query $q \in Q_p$ may prevent sharing another pattern p' by the query q if the patterns p and p' overlap in the query q. Such sharing candidates are said to be in a *sharing conflict*. In this chapter, we encode sharing candidates, their benefit, and conflicts among them into the SHARON graph. Based on the graph, we then reduce the search space of our sharing plan finder (Sections 23.2–23.3).

Example 23.1 *In Table 23.1, queries q_3 and q_4 contain the overlapping patterns $p_2 = (ParkAve, OakSt)$ and $p_1 = (OakSt, MainSt)$. Since the executor computes and stores the aggregates for a pattern as a whole (Chapter 22), queries q_3 and q_4 can either share p_1 or p_2, but not both. Thus, the sharing candidates $(p_1, \{q_1, q_2, q_3, q_4\})$ and $(p_2, \{q_3, q_4\})$ give "contradictory instructions" for queries q_3 and q_4. These candidates are said to be in a sharing conflict. However, if p_1 were to be shared only by queries q_1 and q_2, the sharing conflict between these candidates would be resolved. We sketch the sharing conflict resolution techniques in Chapter 24.*

Definition 23.1 ***(Sharing Conflict.)*** *Let $p_A = (A_0 \dots A_n)$ and $p_B = (B_0 \dots B_m)$ be*

Pattern p	**Queries** $Q_p \subseteq Q$ containing p
$p_1 = (\mathit{OakSt}, \mathit{MainSt})$	q_1, q_2, q_3, q_4
$p_2 = (\mathit{ParkAve}, \mathit{OakSt})$	q_3, q_4
$p_3 = (\mathit{ParkAve}, \mathit{OakSt}, \mathit{MainSt})$	q_3, q_4
$p_4 = (\mathit{MainSt}, \mathit{WestSt})$	q_2, q_4
$p_5 = (\mathit{OakSt}, \mathit{MainSt}, \mathit{WestSt})$	q_2, q_4
$p_6 = (\mathit{MainSt}, \mathit{StateSt})$	q_1, q_5
$p_7 = (\mathit{ElmSt}, \mathit{ParkAve})$	q_6, q_7

Table 23.1: Sharing candidates of the form (p, Q_p) in the workload Q

patterns, $n, m \in \mathbb{N}$, and Q_A and Q_B be query sets. The sharing candidates (p_A, Q_A) and (p_B, Q_B) are in sharing conflict if p_A overlaps with p_B in at least one query $q \in Q_A \cap Q_B$, i.e., $\exists k \in \mathbb{N}, 0 \leq k \leq n, m$ $A_{n-k} \ldots A_n = B_0 \ldots B_k$ in q. The query q causes the conflict between (p_A, Q_A) and (p_B, Q_B).

Definition 23.2 ***(Valid Sharing Plan.)*** *A sharing plan $\mathcal{P}$ is a set of sharing candidates. $\mathcal{P}$ is called valid if it contains no candidates that are in a sharing conflict with each other. $\mathcal{P}$ is called invalid otherwise.*

Definition 23.3 ***(Score of a Sharing Plan.)*** *The score of a sharing plan $\mathcal{P} = \{(p_1, Q_{p1}), \ldots, (p_s, Q_{ps})\}$ is:*

$$\mathit{Score}(\mathcal{P}) = \sum_{j=1}^{s} \mathit{BValue}(p_j, Q_{pj}) \tag{23.1}$$

Definition 23.4 ***(Optimal Sharing Plan.)*** *Let $\mathbb{P}_{val}$ be the set of all valid sharing plans. $\mathcal{P}_{opt} \in \mathbb{P}_{val}$ is an optimal sharing plan if $\nexists \mathcal{P} \in \mathbb{P}_{val}$ with $\mathit{Score}(\mathcal{P}) > \mathit{Score}(\mathcal{P}_{opt})$.*

Example 23.2 *Given the workload in Figure 1.3, the sharing plan $\mathcal{P} = \{(p_2, \{q_3, q_4\}); (p_4, \{q_2, q_4\})\}$ is valid. Its sharing candidates are not in conflict since the patterns $p_2 = (\mathit{ParkAve}, \mathit{OakSt})$ and $p_4 = (\mathit{MainSt}, \mathit{WestSt})$ do not overlap. However, $\mathcal{P}$ is not an*

Algorithm 10 SHARON graph construction algorithm

Input: A hash table H mapping each sharable pattern p to a set of queries Q that contain p

Output: SHARON graph $G = (V, E)$

```
1: $V \leftarrow \emptyset;\ E \leftarrow \emptyset;\ G \leftarrow (V, E)$
2: **for all** $p$ in $H$ **do** $Q_p \leftarrow H.get(p)$
3:     **if** $BValue(p, Q_p) > 0$ **and** $Q_p.size > 1$ **then**
4:         $v \leftarrow (p, Q_p);\ v.weight \leftarrow BValue(p, Q_p))$
5:         $V \leftarrow V \cup v$
6:         **for all** $u$ in $V$ **do**
7:             **if** $v$ and $u$ are in sharing conflict **then**
8:                 $E \leftarrow E \cup (v, u)$
9: **return** $G$
```

optimal plan because $Score(\mathcal{P}) = 24$ is not maximal among all valid plans. Indeed, another valid plan $\{(p_1, \{q_1, q_2, q_3, q_4\})\}$ has higher score 25.

Definition 23.5 *(**SHARON Graph.**) Let S be the set of sharable patterns in a workload Q. The* SHARON *graph $G = (V, E)$ has a set of weighted vertices V and a set of undirected edges E. Each vertex $v \in V$ represents a sharing candidate (p, Q_p) where $p \in S$ is a pattern and $Q_p \subseteq Q$ is the set of queries containing p. Each vertex is assigned a weight $BValue(p, Q_p) > 0$ that corresponds to the benefit value of (p, Q_p) (Equation 22.11). Each edge $(v, u) \in E$ represents a sharing conflict between the candidates $v, u \in V$.*

Example 23.3 *Figure 5.1 shows the* SHARON *graph for the traffic monitoring workload in Figure 1.3 and Table 23.1.*

SHARON Graph Construction Algorithm consumes a hash table H that maps each sharable pattern p to the set of queries Q that contain p. If a pattern p is beneficial to be shared by at least two queries, the vertex $v = (p, Q_p)$ with weight $BValue(p, Q_p)$ is inserted into the graph (Lines 3–5 in Algorithm 10). Non-beneficial candidates are pruned. The edges representing the sharing conflicts between v and other vertices in the graph are inserted (Lines 6–8). The graph G is returned (Line 9).

Complexity Analysis. The time complexity is determined by the nested for-loops since all other operations take constant time, i.e., $\Theta(|H||V|)$. The space complexity corresponds to the size of the graph G, i.e., $\Theta(|V| + |E|)$.

23.2 Sharing Candidate Pruning

Since the search space for an optimal plan is exponential in the number of candidates (Section 23.3), we prune two classes of candidates from a SHARON graph. One, *conflict-ridden candidates* are guaranteed not to be in the optimal plan because their benefit values are too low to counterbalance the loss of benefit from the sharing opportunities they exclude. Two, *conflict-free candidates* are guaranteed to be in the optimal plan since they do not prevent any other sharing opportunities.

Conflict-Ridden Candidates. We now map our MESA problem to the problem of finding a Maximum Weight Independent Set (MWIS) in a graph which is known to be NP-hard [77]. The greedy algorithm GWMIN [44] for MWIS[1] does not always return a high-quality sharing plan as confirmed by our experiments in Section 25.3. However, its guaranteed minimal weight can be used to prune conflict-ridden candidates from a SHARON graph.

Definition 23.6 ***(Maximum Weight Independent Set (MWIS).)*** *Let $G = (V, E)$ be a graph with a set of weighted vertices V and a set of edges E. For a set of vertices $V' \subseteq V$, we denote the sum of the weights of the vertices in V' as $Weight(V')$. $IS \subseteq V$ is said to be an independent set of G if for any vertices $v_i, v_j \in IS$, $(v_i, v_j) \notin E$ holds. Let S_{IS} be the set of all independent sets of G. $IS \in S_{IS}$ is a maximum weight independent set of G if $\nexists IS' \in S_{IS}$ with $Weight(IS') > Weight(IS)$.*

[1] The GWMIN algorithm is described in [76].

Lemma 23.1 *Let Q be a query workload, G be the* SHARON *graph for Q, and $\mathcal{P}_{opt}$ be an optimal sharing plan for Q. Then, $\mathcal{P}_{opt}$ is an MWIS of G.*

Proof: By Definitions 23.2 and 23.4, $\mathcal{P}_{opt}$ is valid. That is, it contains no conflicting sharing candidates. By Definition 23.5, no vertices in $\mathcal{P}_{opt}$ are connected by an edge in G. By Definition 23.6, $\mathcal{P}_{opt}$ is an independent set of G. By Definition 23.4, $\mathcal{P}_{opt}$ has the maximum score among all valid plans. By Definition 23.5, $\mathcal{P}_{opt}$ has the maximum weight among all independent sets of G. By Definition 23.6, $\mathcal{P}_{opt}$ is an MWIS of G. ■

The GWMIN algorithm is proven in [44] to find an independent set IS with weight:

$$Weight(IS) \geq \sum_{u \in V} \frac{weight(u)}{degree(u) + 1} \tag{23.2}$$

To safely prune a conflict-ridden candidate v, we define the maximal score of a plan containing v, denoted $Score_{max}(v)$. In the best case, a plan containing v includes all other candidates that are not in conflict with v. Thus, $Score_{max}(v)$ corresponds to the summation of benefit values of all sharing candidates that are not in conflict with v.

Definition 23.7 ***(Maximal Score of a Plan Containing a Sharing Candidate.)*** *Let $v \in V$ be a sharing candidate in a* SHARON *graph $G = (V, E)$ and $\mathcal{N}(v) \subseteq V$ be the neighborhood of v, i.e., the set of candidates that are in conflict with v. Then, we define the maximal score of a sharing plan containing v as follows:*

$$Score_{max}(v) = \sum_{u \in V \setminus \mathcal{N}(v)} BValue(u) \tag{23.3}$$

Lemma 23.2 *For a valid sharing plan $\mathcal{P}$ and a sharing candidate $v \in \mathcal{P}$, $Score(\mathcal{P}) \leq Score_{max}(v)$ holds.*

Proof: Let $G = (V, E)$ be the SHARON graph such that $\mathcal{P} \subseteq V$ is an independent set of G and $v \in \mathcal{P}$. By Definition 23.2, $\mathcal{P}$ contains no conflicting sharing candidates. By Definition 23.5, all neighbors of v, denoted $\mathcal{N}(v)$, are in conflict with v and thus are not in $\mathcal{P}$. Since $\mathcal{P}$ may need to remove additional vertices to avoid other conflicts, $\mathcal{P} \subseteq V \setminus \mathcal{N}(v)$. By Definition 23.7, $Score_{max}(v)$ corresponds to the sum of BValues of all sharing candidates in $V \setminus \mathcal{N}(v)$. Since all BValues of vertices in V are positive (Section 22.4), $\mathcal{P} \subseteq V \setminus \mathcal{N}(v)$ implies $Score(\mathcal{P}) \leq Score_{max}(v)$. ■

Definition 23.8 ***(Conflict-Ridden Sharing Candidate.)*** *Let $G = (V, E)$ be a* SHARON *graph. A sharing candidate $v \in V$ is conflict-ridden if the maximal score of a sharing plan containing v is lower than the guaranteed weight of GWMIN.*

$$Score_{max}(v) < \sum_{u \in V} \frac{BValue(u)}{degree(u) + 1} \tag{23.4}$$

Conflict-Ridden Candidate Pruning. All conflict-ridden candidates are pruned from the SHARON graph without sacrificing the optimality of the resulting sharing plan.

Example 23.4 *The guaranteed weight on the graph in Figure 5.1 is $\frac{25}{6} + \frac{9}{4} + \frac{12}{5} + \frac{15}{4} + \frac{20}{5} + \frac{8}{2} + \frac{18}{1} \approx 38.57$. Since $Score_{max}(p_3, \{q_3, q_4\}) = BValue(p_3, \{q_3, q_4\}) + BValue(p_6, \{q_1, q_5\}) + BValue(p_7, \{q_6, q_7\}) = 38 < 38.57$, an optimal sharing plan cannot contain $(p_3, \{q_3, q_4\})$. Thus, this candidate and its conflicts can be pruned.*

Conflict-Free Candidates do not exclude any other sharing opportunities and increment the score of a plan by their benefit values. Such candidates can be directly added to an optimal plan and removed from further analysis.

Definition 23.9 ***(Conflict-Free Sharing Candidate.)*** *A sharing candidate $v \in V$ in a* SHARON *graph $G = (V, E)$ is conflict-free if $\nexists u \in V$ with $(v, u) \in E$.*

Algorithm 11 SHARON graph reduction algorithm

Input: SHARON graph G, guaranteed weight min of GWMIN

Output: Reduced graph G, conflict-free candidates F

```
1: $F \leftarrow \emptyset$
2: while $G$ can be reduced do
3:     for all $v$ in $V$ do
4:         if degree($v$) = 0 then
5:             $F \leftarrow F \cup v$; $G$.remove($v$)
6:         else if $Score_{max}(v) < min$ then
7:             $G$.remove($v$)
8: return $G, F$
```

Example 23.5 *The conflict-free candidate $(p_7, \{q_6, q_7\})$ in Figure 5.1 increments the score of a plan by its benefit 18.*

SHARON Graph Reduction Algorithm consumes a SHARON graph G and the guaranteed weight of GWMIN. Algorithm 11 iterates over the vertices of G and removes conflict-ridden or conflict-free candidates until the graph G cannot be reduced anymore. The algorithm returns the reduced graph and the set of conflict-free candidates.

Complexity Analysis. The time complexity is determined by the nested loops that iterate $O(|V|)$ and $\Theta(|V|)$ times respectively. The time complexity of removing a candidate v from the graph in Line 7 is $O(|E|)$ since all conflicts of v are also deleted. Thus, the time complexity is quadratic $O(|V|^2|E|)$. The space complexity is determined by the size of the graph G and the set F. Since $|F| \leq |V|$, the space costs are linear, i.e., $O(|V| + |E|)$.

Example 23.6 *Figure 23.1 depicts the search space for an optimal plan for our running example. Since the conflict-ridden candidate $(p_3, \{q_3, q_4\})$ is pruned (Example 23.4), while the conflict-free candidate $(p_7, \{q_6, q_7\})$ is added to the optimal plan (Example 23.5), the search space is reduced by $2^7 - 2^5 = 96$ plans. This reduced space is indicated by a solid frame in Figure 23.1. It corresponds to 75.59% of the search space.*

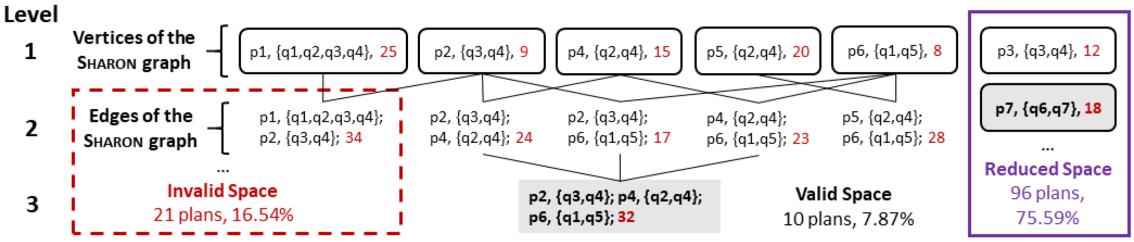


Figure 23.1: Search space of the sharing plan finder algorithm

23.3 Sharing Plan Finder

Based on the reduced SHARON graph, we now propose the optimal sharing plan finder. In addition to the non-beneficial and conflict-ridden candidate pruning principles, we define the invalid branch pruning. It cuts off those branches of the search space that only contain invalid plans early on.

Search Space for an Optimal Sharing Plan. The parent-child relationships between sharing plans, depicted as lines in Figure 23.1, are defined next.

Definition 23.10 *(**Parent-Child Relationship between Sharing Plans**.) Let $\mathcal{P}$ and $\mathcal{P}'$ be sharing plans. If $\mathcal{P} \subset \mathcal{P}'$, then we say that $\mathcal{P}$ is an ancestor of $\mathcal{P}'$ ($\mathcal{P}'$ is a descendant of $\mathcal{P}$). In addition, if $|\mathcal{P}| = |\mathcal{P}'| - 1$, then we say that $\mathcal{P}$ is a parent of $\mathcal{P}'$ ($\mathcal{P}'$ is a child of $\mathcal{P}$).*

The search space has a lattice shape. In Figure 23.1, the plans in the top level (Level 1) correspond to the sharing candidates V in a SHARON graph (Figure 5.1). Level s contains sharing plans of size s. The *size of the search space* is exponential in the number of candidates, denoted $|V|$. It is computed as the sum of the number of plans at each level:

$$\sum_{s=0}^{|V|} \binom{|V|}{s} = 2^{|V|} \tag{23.5}$$

Lemma 23.3 *If $\mathcal{P}$ is a parent plan of $\mathcal{P}'$, then $Score(\mathcal{P}') > Score(\mathcal{P})$.*

Proof: By Definition 23.10, $\mathcal{P} \subset \mathcal{P}'$ and $|\mathcal{P}| = |\mathcal{P}'| - 1$. Let $\mathcal{P}' \setminus \mathcal{P} = (p, Q_p)$. By

Definition 23.5, only a beneficial candidate (p, Q_p) is included into a SHARON graph, i.e., $BValue(p, Q_p) > 0$. Thus, the candidate (p, Q_p) increases the score of $\mathcal{P}'$ compared to $\mathcal{P}$.

■

A naive plan finder traverses all combinations of sharing candidates and keeps track of a valid plan with the maximal score seen so far. However, this solution constructs many invalid plans (Definition 23.2) that are subsequently discarded. To avoid such exhaustive search, we prove the following properties of the search space.

Lemma 23.4 *All descendants of an invalid plan are invalid.*

Proof: Let $\mathcal{P}$ be an invalid sharing plan and $\mathcal{P}_d$ be its descendant. By Definition 23.10, $\mathcal{P} \subset \mathcal{P}_d$. Thus, $\mathcal{P}_d$ "inherits" all sharing conflicts from $\mathcal{P}$ which makes $\mathcal{P}_d$ invalid. ■

Invalid Branch Pruning. Invalid plans of size two correspond to edges of a SHARON graph (Figures 5.1 and 23.1). Thus, all ancestors of invalid plans of size two can be safely pruned. Thus, our SHARON plan finder cuts off invalid branches of the search at their roots.

Example 23.7 *In Figure 23.1, only 7.87% of the search space is **valid**. It consists of 10 plans. This valid space is traversed to find the optimal plan* $\{(p_2, \{q_3, q_4\}); (p_4, \{q_2, q_4\}); (p_6, \{q_1, q_5\}); (p_7, \{q_6, q_7\})\}$ *highlighted by a darker background.*

*16.54% of the search space is **invalid**. The invalid space consists of 21 plans =* 2^5 *not reduced plans – 10 valid plans – 1 empty plan. The invalid space is indicated by the dashed frame. It is pruned by our plan finder.*

*The rest of the search space was **reduced** by pruning conflict-ridden and conflict-free candidates in Example 23.6.*

Valid Search Space Traversal. A plan of size one is valid by Definition 23.2. A plan of size two $\{v_1, v_2\}$ is valid if there is no edge (v_1, v_2) in the SHARON graph. Validity of a larger plan is determined as described next.

Lemma 23.5 *A sharing plan $\mathcal{P}$, $|\mathcal{P}| > 2$, is valid, if all its parents are valid.*

Proof: Let $\mathcal{P}$ be a valid plan and $\mathcal{P}_p$ be its parent. Assume $\mathcal{P}_p$ is invalid. By Lemma 23.4, $\mathcal{P}$ is invalid which contradicts our assumption. Thus, $\mathcal{P}_p$ is valid. ■

By Definition 23.10, a plan of size s has s parents. Instead of accessing all parent plans to generate one new valid plan, we prove that only two parents and one ancestor of size two must be valid to guarantee validity of a sharing plan (similarly to Apriori candidate generation [78]).

Lemma 23.6 *Let $G = (V, E)$ be a* SHARON *graph. Let $\mathcal{P}_1$ and $\mathcal{P}_2$ be any pair of valid parents of a plan $\mathcal{P}$, $|\mathcal{P}| > 2$. For two candidates $v_1 = \mathcal{P}_1 \setminus \mathcal{P}_2$ and $v_2 = \mathcal{P}_2 \setminus \mathcal{P}_1$, if $(v_1, v_2) \notin E$, then $\mathcal{P}$ is valid.*

Proof: Assume all the conditions above hold but $\mathcal{P}$ is invalid. Then $\mathcal{P}$ contains at least one pair of conflicting candidates. By Definition 23.10, $\mathcal{P} = \mathcal{P}_1 \cup \mathcal{P}_2$ and $\mathcal{P}$ has one additional candidate compared to $\mathcal{P}_1$ (or $\mathcal{P}_2$). Since $\mathcal{P}_1$ and $\mathcal{P}_2$ are valid, there can be only one pair of conflicting candidates v_1 and v_2 in $\mathcal{P}$ such that $v_1 = \mathcal{P}_1 \setminus \mathcal{P}_2$ and $v_2 = \mathcal{P}_2 \setminus \mathcal{P}_1$. By Definition 23.5, $(v_1, v_2) \in E$ which is a contradiction. ■

Level Generation Algorithm consumes a SHARON graph G and a set of sharing plans of size s, called $Parents$. It returns level $s + 1$ of the search space, i.e., the set of all sharing plans of size $s + 1$, called $Children$. Algorithm 12 iterates through all pairs of parent plans of size s (Lines 3–4). In the base case, the $Parents$ are the vertices of G and the $Children$ are non-adjacent pairs of vertices (Lines 5–6). In the inductive case, to generate a valid plan of size $s + 1$, the algorithm identifies two plans of size s, P_i and P_j, that begin with the same $s - 1$ decisions. If the plan containing the last decisions of P_i and P_j (denoted $P_i.v_s$ and $P_j.v_s$) is valid, the plan $P_i \cup P_j$ is also valid (Lines 7–8).

Algorithm 12 Level generation algorithm

```
Input: SHARON graph $G = (V, E)$, set of sharing plans of size $s$ $Parents = \{P_0, \dots, P_{s-1}\}$
Output: Set of sharing plans of size $s + 1$ $Children$
getNextLevel$(G, Parents)$ {
Children $\leftarrow \emptyset$
for all $(i = 0; i < s; i{+}{+})$ do
    for all $(j = i + 1; j < s; j{+}{+})$ do
        if $s = 1$ and $(P_i.v_1, P_j.v_1)$ not in $E$ then
            $Children$.add($P_i \cup P_j$)
        if $P_i.v_1 = P_j.v_1, \dots, P_i.v_{s-1} = P_j.v_{s-1}$ and $(P_i.v_s, P_j.v_s)$ not in $E$ then
            $Children$.add($P_i \cup P_j$)
return $Children$ }
```

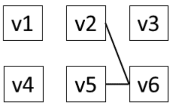


Figure 23.2: SHARON graph

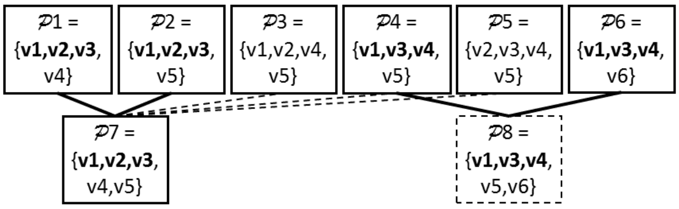


Figure 23.3: Generation of a new valid sharing plan

Complexity Analysis. The time complexity of Algorithm 12 is determined by the number of plans at one level, namely, the binomial coefficient $\binom{|V|}{s}$ in Equation 23.5. Due to two nested loops, the time complexity is $O(\binom{|V|}{s}^2)$. The space complexity is also determined by the number of plans at one level, i.e., $O\binom{|V|}{s}$.

Example 23.8 *Figure 23.3 shows a portion of a search space with six valid plans $\mathcal{P}_1$–$\mathcal{P}_6$ of size four. $\mathcal{P}_7$ is the only valid plan of size five. It is generated as follows. (1) We identify two plans of size four that start with the same three candidates, e.g., $\mathcal{P}_1$ and $\mathcal{P}_2$ start with $\{v_1, v_2, v_3\}$. (2) We compute their symmetric difference $\mathcal{P}_1 \Delta \mathcal{P}_2 = \{v_4, v_5\}$. (3) Since there is no edge (v_4, v_5) in the* SHARON *graph in Figure 23.2, $\mathcal{P}_7$ is valid. There is no need to check the other three parents of $\mathcal{P}_7$. In contrast, $\mathcal{P}_8$ is invalid since v_5 and v_6 are in conflict.*

Sharing Plan Finder Algorithm traverses valid search space level by level using Al-

Algorithm 13 Sharing plan finder algorithm

```
Input: SHARON graph $G = (V, E)$, set of conflict-free candidates $F$
Output: Optimal sharing plan $opt \cup F$
 1: $opt \leftarrow \emptyset$; $max \leftarrow 0$
 2: for all $v$ in $V$ do
 3:     if $BValue(v) > max$ then
 4:         $opt \leftarrow \{v\}$; $max \leftarrow BValue(v)$
 5: $Level \leftarrow getNextLevel(G, V)$
 6: while $Level \neq \emptyset$ do
 7:     for all $P$ in $Level$ do
 8:         if $Score(P) > max$ then
 9:             $opt \leftarrow P$; $max \leftarrow Score(P)$
10:     $Level \leftarrow getNextLevel(G, Level)$
11: return $opt \cup F$
```

gorithm 12. Algorithm 13 effectively prunes invalid branches at their roots. It constructs only valid plans and returns an optimal plan among them.

Correctness. We prove that Algorithm 13 considers all valid sharing plans, i.e., it returns the optimal sharing plan.

Lemma 23.7 *If a sharing plan is valid, then it is considered by the sharing plan finder algorithm.*

Proof: We prove Lemma 23.7 by induction. The base cases are $s = 1$ and $s = 2$. First, V is the set of all valid sharing plans of size 1. It is considered by Algorithm 13. Second, Algorithm 13 (Line 5) generates all plans of size 2 by considering all non-adjacent vertex pairs in Algorithm 12 (Lines 5–6).

Now, we assume that all valid plans of up to and including size s, such that $s \geq 2$, are considered. We will prove that all valid plans of size $s + 1$ are also considered. Let $\mathcal{P} = \{v_1, \dots, v_s, v_{s+1}\}$ be a valid plan of size $s + 1$. Then $\mathcal{P}_1 = \{v_1, \dots, v_{s-1}, v_s\}$, $\mathcal{P}_2 = \{v_1, \dots, v_{s-1}, v_{s+1}\}$, and $\mathcal{P}_3 = \{v_s, v_{s+1}\}$ are ancestors of $\mathcal{P}$. By Lemma 23.5, $\mathcal{P}_1, \mathcal{P}_2$, and $\mathcal{P}_3$ are valid. By the induction assumption, they were considered by Algorithm 13.

Algorithm 12 generates the plan $\mathcal{P}$ from $\mathcal{P}_1, \mathcal{P}_2$, and $\mathcal{P}_3$. Thus, Algorithm 13 considers $\mathcal{P}$. ■

Complexity Analysis. Since the entire valid search space is traversed, the algorithm has exponential time and space complexity in the worst case (Equation 23.5). However, the SHARON optimizer is efficient on average thanks to its effective pruning principles (Section 25.3).

Optimal versus Greedily Chosen Plan. While the greedy algorithm GWMIN is useful to reduce the search space (Section 23.2), the score of a greedily chosen plan might be considerably lower than the score of an optimal plan.

Example 23.9 *Even in our small example in Figure 5.1, the greedily chosen plan* $\mathcal{P}_{gre}$ = $\{(p_1, \{q_1, q_2, q_3, q_4\})$*;* $(p_7, \{q_6, q_7\})\}$ *has score 43, while the optimal plan* $\mathcal{P}_{opt}$ = $\{(p_2, \{q_3, q_4\})$*;* $(p_4, \{q_2, q_4\}$*;* $(p_6, \{q_1, q_5\})$*;* $(p_7, \{q_6, q_7\})\}$ *increases* $Score(\mathcal{P}_{gre})$ *by more than 16% to 50.*

24

Extensions of the Sharon Approach

In this chapter, we briefly describe the extensions of our approach to relax the simplifying assumptions in Chapter 21.

24.1 Sharing Conflict Resolution

Our analysis in Section 23.1 reveals that promising sharing opportunities might be excluded by sharing conflicts. Generally, the more queries share a pattern the higher the probability of sharing conflicts becomes (Definition 23.1). We now open up additional sharing opportunities by resolving sharing conflicts as follows.

Given a SHARON graph $G = (V, E)$, we expand each candidate $v = (p, Q_p) \in V$ with conflicts $E_v \subseteq E$ to a set of options $\mathcal{O}_p$. Each option $v' = (p, Q'_p) \in \mathcal{O}_p$ resolves a different subset of conflicts $E'_v \subseteq E_v$ of the original candidate v with other candidates $u \in V \setminus O_p$. In contrast to the original candidate v, an option v' considers sharing the pattern p by a *subset* of queries containing p, i.e., $Q'_p \subseteq Q_p$, $|Q'_p| > 1$.

Example 24.1 *In Figure 5.1, the sharing candidate* $(p_1, \{q_1, q_2, q_3, q_4\})$ *can be expanded to a set of options. The option* $(p_1, \{q_1, q_3\})$ *is not in sharing conflict with the candidates*

Algorithm 14 Sharing candidate expansion algorithm

```
Input: SHARON graph $G = (V, E),\ v = (p, Q_p) \in V$
Output: Set $O_p$ of sharing candidate options for $p$
getSet$(G, v)$ {
$L_c, L_n \leftarrow$ empty stacks; $L_c.push(v);\ O_p \leftarrow \{v\}$
while $!L_c.isEmpty()$ do
    $v \leftarrow L_c.pop()$
    for all conflict $(v, u)$ in $E$ do
        $Q_c \leftarrow$ queries in $Q_p$ that cause $(v, u)$
        for all combination $C$ of $Q_c$ that can resolve $(v, u)$ do
            $Q'_p \leftarrow Q_p \setminus C$
            if $|Q'_p| > 1$ and $Q'_p$ is new then
                $v' \leftarrow (p, Q'_p);\ L_n.push(v');$
                $O_p \leftarrow O_p \cup \{v'\}$
    if $L_c.isEmpty()$ then
        $L_c \leftarrow L_n;\ L_n \leftarrow$ empty stack
return $O_p$ }
```

$(p_4, \{q_2, q_4\})$ *and* $(p_5, \{q_2, q_4\})$*. Thus, they could belong to the same sharing plan which may have a higher score than a plan containing the original candidate* $(p_1, \{q_1, q_2, q_3, q_4\})$.

Definition 24.1 ***(Resolved Sharing Conflict.)*** *Let the candidates* $v_1 = (p_1, Q_1)$ *and* $v_2 = (p_2, Q_2) \in V$ *be in conflict* $(v_1, v_2) \in E$ *caused by the queries* $Q = Q_1 \cap Q_2$ *such that* $Q = Q'_1 \uplus Q'_2$*.*[1] *The conflict* (v_1, v_2) *is resolved by omitting* Q'_1 *and* Q'_2 *from* Q_1 *and* Q_2 *respectively.*

By Definition 23.1, the sharing candidates $v'_1 = (p_1, Q_1 \setminus Q'_1)$ and $v'_2 = (p_2, Q_2 \setminus Q'_2)$ are *not in conflict* since $(Q_1 \setminus Q'_1) \cap (Q_2 \setminus Q'_2) = \emptyset$. The conflict (v_1, v_2) is resolved if *any* query sets Q'_1 and Q'_2 that compose Q are omitted from Q_1 and Q_2 respectively. In the worst case, *all* combinations of queries Q are included into the sets of options for v_1 and v_2.

Sharing Candidate Expansion Algorithm. For a SHARON graph G and a candidate $v = (p, Q_p) \in V$, Algorithm 14 builds a tree of options O_p using Breadth First Search.

[1] $\uplus$ denotes disjoint set union, meaning that $Q = Q'_1 \cup Q'_2$ but $Q'_1 \cap Q'_2 = \emptyset$.

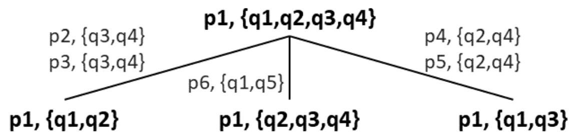


Figure 24.1: Sharing candidate options for pattern p_1

The root of this tree is the original candidate v. To generate a child of v, the algorithm skips the queries from Q_p that cause a conflict of v with another sharing candidate $u \in V \setminus O_p$. We label an edge between v and its child by the sharing candidate u. The algorithm terminates when no new option with at least two queries can be generated.

Complexity Analysis. The time and space complexity of Algorithm 14 are determined by the maximal size of a set $|O_p^{max}|$. Let d be the maximal degree of a candidate $v \in V$ and k be the maximal number of queries that cause a conflict. For each conflict $(v, u) \in E$, all combinations of queries causing this conflict are considered (nested for-loops in Lines 5–10 and 7–10). Thus,

$$|O_p^{max}| = \sum_{i=0}^{d} \binom{d}{i} \sum_{j=0}^{k-1} \binom{k}{j} \tag{24.1}$$

where i denotes the number of resolved conflicts, while j corresponds to the number of skipped queries to resolve one conflict.

Example 24.2 *Figure 24.1 illustrates the sharing candidate options for the candidate $v = (p_1, \{q_1, q_2, q_3, q_4\})$ in Figure 5.1. To resolve the conflict with $u_1 = (p_2, \{q_3, q_4\})$ and $u_2 = (p_3, \{q_3, q_4\})$, queries q_3 and q_4 are dropped from the set of queries of v. The edge between v and its child $(p_1, \{q_1, q_2\})$ is labeled by u_1, u_2. Other conflicts of v are resolved analogously.*

Sharing Conflict Resolution Algorithm. For a SHARON graph G and each candidate $v = (p, Q_p) \in V$, (Algorithm 15) expands v to a set of options O_p using Algorithm 14

Algorithm 15 Sharing conflict resolution algorithm

```
Input: Sharon graph $G = (V, E)$
Output: Expanded Sharon graph $G$
$V' \leftarrow \emptyset$; $E' \leftarrow \emptyset$
for all $v = (p, Q_p)$ in $V$ do
    $O_p \leftarrow getSet(G, v)$; $V' \leftarrow V' \cup O_p$
    for all $v'$ in $O_p$ do
        for all $u$ in $V'$ do
            if $v'$ and $u$ are in sharing conflict then
                $E'.add(v', u)$
return $G \leftarrow (V', E')$
```

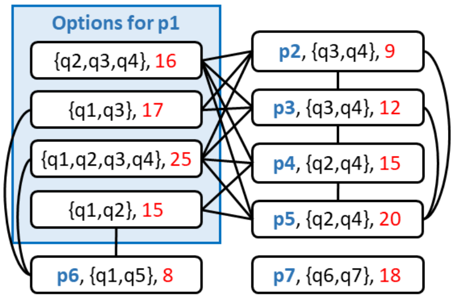


Figure 24.2: Expanded Sharon graph

to open up additional sharing opportunities. The algorithm updates the conflicts of these options and returns the expended graph.

Complexity Analysis. The time complexity is determined by three nested for-loops that are called $\Theta(|V|)$, $|O_p^{max}|$ and $\Theta(|V'|)$ times respectively where $|O_p^{max}|$ denotes the maximal size of a set (Equation 24.1). Since $|V| \leq |V'|$ and $|O_p^{max}| \leq |V'|$, the time complexity is cubic in the number of candidates in the expanded Sharon graph in the worst case, i.e., $O(|V'|^3)$. The space complexity is determined by the size of the expanded graph, i.e., $\Theta(|V'| + |E'|)$.

Example 24.3 *The* Sharon *graph in Figure 5.1 is expanded in Figure 24.2. The sharing candidate for pattern p_1 is expanded into a set of options and highlighted by a rectangle frame. The sets for other candidates contain only the original candidate. Conflicts within sets are omitted for readability.*

The expanded graph is then reduced (Section 23.2) and serves as input to our sharing plan finder (Section 23.3).

24.2 Different Predicates, Grouping, and Windows

Our SHARON approach can share event sequence aggregation among queries with different predicates, grouping, and windows. These query clauses partition the stream into sub-streams [2]. State-of-the-art shared event aggregation approaches [33, 35, 36] further partition these sub-streams into disjoint segments and share the intermediate aggregates per segment to compute the final results for each query. Our SHARON approach can be applied within each segment to handle different patterns of the shared queries.

24.3 Multiple Occurrences of an Event Type in a Pattern

If an event type E occurs k times in a pattern ($k > 1$), an event of type E updates the counts of k prefix patterns that end at E (Chapter 22). Then, the time complexity of both the Not-Shared and the Shared methods increases by the multiplicative factor k (Equations 22.2, 22.4, 22.6 and 22.9). Our SHARON optimizer is not affected by this extension.

25

Performance Evaluation

25.1 Experimental Setup

Infrastructure. We have implemented our SHARON approach in Java with JRE 1.7.0_25 running on Ubuntu 14.04 with 16-core 3.4GHz CPU and 128GB of RAM. We execute each experiment three times and report the average here.

Data Sets. We evaluate the performance our SHARON approach using the following data sets.

• ***TX: New York City Taxi and Uber Real Data Set***. We use the real New York City taxi and Uber data set [12] (330GB) containing 1.3 billion taxi and Uber trips in New York City in 2014–2015. Each event carries pick-up and drop-off locations and time stamps in seconds, number of passengers, price, and payment method.

• ***LR: Linear Road Benchmark Data Set***. We use the traffic simulator of the Linear Road benchmark [39] for streaming systems to generate a stream of position reports from vehicles for 3 hours. Each position report carries a time stamp in seconds, a vehicle identifier, its location and speed. Event rate gradually increases during 3 hours from few dozens to 4k events per second.

- ***EC: E-Commerce Synthetic Data Set***. Our stream generator creates sequences of items bought together for 3 hours. Each event carries a time stamp in seconds, item and customer identifiers. We consider 50 items and 20 users. The values of item and customer identifiers of an event are randomly generated. The stream rate is 3k events per second.

We ran each experiment on three data sets above. Similar charts are not shown here.

Event Queries. We evaluate a query workload similar to q_1–q_7 in Section 1.1 against the taxi and Linear Road data sets and a workload similar to q_8–q_{11} [76] against the e-commerce data set. Based on our cost model (Chapter 22), we vary the major cost factors, namely, number of queries from 20 to 120, the length of their patterns from 5 to 30, and the number of events per window from 5k to 1200k. Unless stated otherwise, we evaluate 20 queries. The default length of their patterns is 10. The default number of events per window is 200k.

Methodology. We run two sets of experiments.

1) ***Sharon Executor vs. State-of-the-Art Approaches*** (Section 25.2). We demonstrate the effectiveness of our SHARON executor (Chapter 22) by comparing it to the state-of-the-art techniques A-Seq [2], SPASS [38], and Flink [30] covering the spectrum of approaches to event sequence aggregation (Table 1.2). While Chapter 26 is devoted to a detailed discussion of these approaches, we briefly sketch their main ideas below.

- ***A-Seq*** [2] avoids sequence construction by incrementally maintaining a count for each prefix of a pattern. However, it has no optimizer to determine which queries should share the aggregation of which patterns. By default, it computes each query independently from other queries and thus suffers from repeated computations (Section 22.2).

- ***SPASS*** [38] defines shared event sequence construction. Their aggregation is computed afterwards and is not shared. Thus, SPASS is a two-step and only partially shared approach.

- ***Flink*** [30] is a popular open-source streaming system that supports event pattern matching and aggregation. We express our queries using Flink operators. Flink constructs all event sequences prior their aggregation. It does not share computations among different queries.

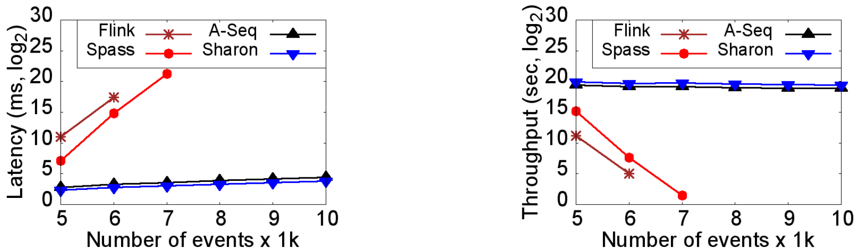
To achieve a fair comparison, we have implemented A-Seq and SPASS on top of our platform. We execute Flink on the same hardware as our platform.

2) ***Sharon Optimizer*** (Section 25.3). We study the effectiveness of our SHARON optimizer (Chapters 23–24) by comparing it to the greedy algorithm GWMIN [44] and the exhaustive search. We also compare the quality of a greedily chosen plan returned by GWMIN to an optimal plan returned by our SHARON optimizer and the exhaustive search.

Metrics. We measure three metrics common for streaming systems, namely, *latency, throughput,* and *peak memory*. We measure ***latency*** in milliseconds as the average time difference between the time point of the aggregation result output by a query in the workload and the arrival time of the latest event that contributed to this result. ***Throughput*** corresponds to the average number of events processed by all queries per second. ***Peak memory*** consumption is measured in bytes. For event sequence aggregation algorithms, it corresponds to the maximal memory for storing aggregated values, events, and event sequences. For the optimizer algorithms, the peak memory is the maximal memory for storing the SHARON graph and the sharing plans during the space traversal.

25.2 Sharon Executor versus State-of-the-Art Approaches

Two-step Approaches. In Figure 25.1, we vary the number of events per window and measure latency and throughput of the event sequence aggregation approaches using the Linear Road benchmark data set. Latency of the two-step approaches (SPASS and Flink) increases exponentially, while throughput decreases exponentially in the number of events.



(a) Latency

(b) Throughput

Figure 25.1: Two-step versus online approaches (Linear Road data set)

SPASS achieves 6–fold speed-up compared to Flink for 6k events per window because SPASS shares event sequence construction. Despite this sharing, SPASS constructs all event sequences and thus fails to terminate within several hours when the number of events per window exceeds 7k. These measurements are not shown in Figure 25.1.

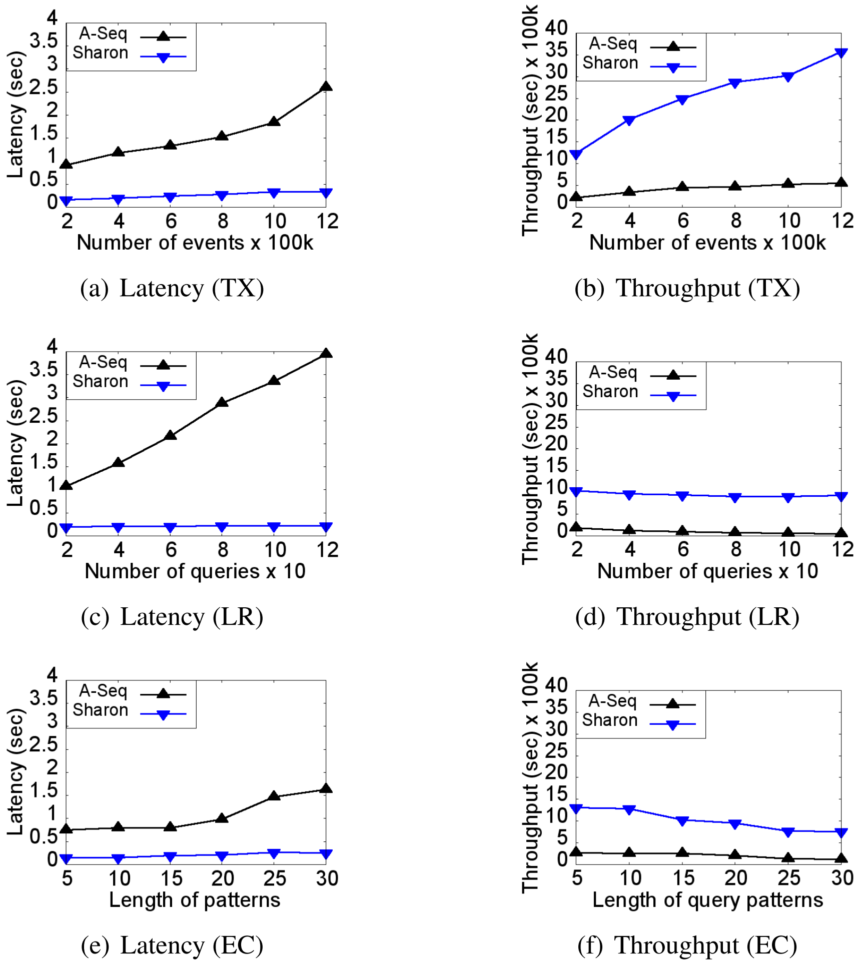
Flink not only constructs all event sequences but also computes each query independently from other queries in the workload. Flink fails for more than 6k events per window.

The event sequence construction step has polynomial time complexity in the number of events [2, 3] and may jeopardize real-time responsiveness for high-rate event streams (Figure 25.1). Thus, these two-step approaches cannot be effective for time-critical processing of high-rate streams.

Online Approaches. The online approaches (A-Seq and SHARON) perform similarly for such low-rate streams. They achieve five orders of magnitude speed-up compared to SPASS for 7k events per window because they aggregate event sequences without first constructing these sequences.

Figures 25.2 and 25.3 evaluate the online approaches against high-rate streams. We vary the number of events per window, the number of queries, and the length of their patterns and measure latency, throughput and memory consumption of the online approaches using the taxi (TX), Linear Road (LR), and e-commerce (EC) data sets.

Sharon Executor shares event sequences aggregation among all queries in the work-



(a) Latency (TX)

(b) Throughput (TX)

(c) Latency (LR)

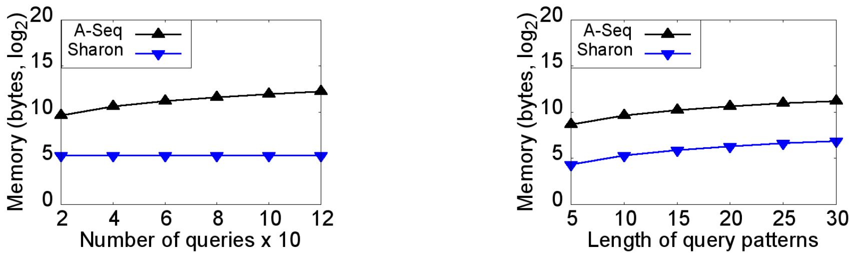
(d) Throughput (LR)

(e) Latency (EC)

(f) Throughput (EC)

Figure 25.2: Latency and throughput of the online approaches (Taxi (TX), Linear Road (LR), and e-commerce (EC) data sets)

load according to an optimal sharing plan. For low parameter values, SHARON defaults to ***A-Seq*** since the available sharing opportunities have low benefit. For example, the latency of SHARON increases linearly in the number of queries. SHARON achieves from 5–fold to 18–fold speed-up compared to A-Seq when the number of queries increases from 20 to 120. Indeed, the more queries share their aggregation results, the fewer aggregates are maintained and the more events can be processed by the system (Figures 25.2(c) and 25.2(d)). SHARON requires up to two orders of magnitude less memory than A-Seq for 120 queries (Figure 25.3(a)).



(a) Memory (LR)

(b) Memory (EC)

Figure 25.3: Memory of the online approaches (Linear Road (LR) and e-commerce (EC) data sets)

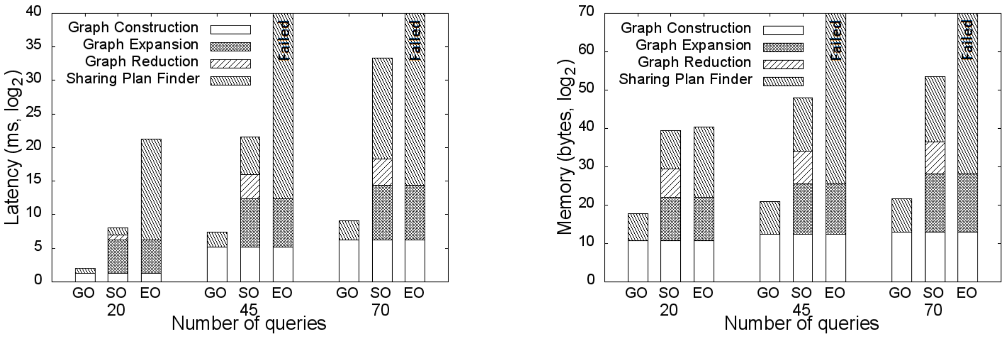
While SHARON processes each event by each shared pattern exactly once, each event can provoke repeated computations in A-Seq. Thus, the gain of SHARON grows linearly in the number of events per window. SHARON wins from 5–fold to 7–fold with respect to latency and throughput when the number of events increases from 200k to 1200k (Figures 25.2(a) and 25.2(b)). Similarly, the speed-up of SHARON grows linearly from 4–fold to 6–fold with the increasing length of patterns (Figure 25.2(e)). SHARON requires 20-fold less memory than A-Seq if the pattern length is 30 (Figure 25.3(b)).

Based on the experimental results in Figures 25.1, 25.2, and25.3, we conclude that the latency, throughput and memory utilization of event sequence aggregation can be considerably reduced by the seamless integration of *shared* and *online* optimization techniques as proposed by our SHARON approach to enable real-time in-memory event sequence aggregation.

25.3 Sharon Optimizer

In Figure 25.4 we compare three optimizer solutions, while varying the number of queries. Each bar is segmented into phases as described below.

Greedy Optimizer consists of the following two phases: the SHARON graph construction (Section 23.1) followed by the GWMIN plan finder. In the worst case, both phases



(a) Latency

(b) Memory

Figure 25.4: SHARON optimizer (SO) versus greedy optimizer (GO) and exhaustive optimizer (EO) (E-commerce query workload)

have polynomial latency and linear memory. However, our experiments show that on average more time and space is required to construct the SHARON graph than to run the GWMIN algorithm. For 70 queries, 90% of the total time is spent constructing the graph.

Exhaustive Optimizer consists of three phases, namely, SHARON graph construction, graph expansion to open up additional sharing opportunities (Chapter 24) followed by the exhaustive search that traverses the entire search space. Thus, its latency and memory consumption grow exponentially in the number of queries. The exhaustive optimizer fails to terminate for more than 20 queries. For 20 queries, its latency is 4 orders of magnitude higher than the latency of the greedy optimizer.

Sharon Optimizer consists of four phases, namely, SHARON graph construction, graph expansion to open up additional sharing opportunities, graph reduction followed by the sharing plan finder (Chapters 23–24). While its complexity is exponential in the worst case (Equation 23.5), its latency and memory usage are reduced by our pruning principles compared to the exhaustive optimizer. On average, 36% of the sharing candidates are pruned from the expanded SHARON graph, which is 99% of the plan finder search space. For 20 queries, SHARON outperforms the exhaustive optimizer by three orders of magnitude with respect to latency and by two orders of magnitude regarding memory

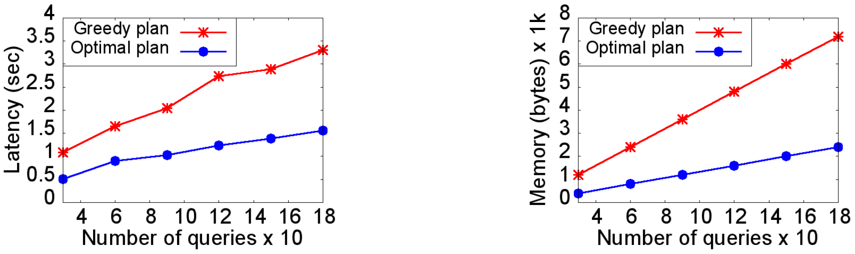


Figure 25.5: Sharing plan quality (Taxi data set)

usage.

Our SHARON plan finder traverses the entire valid space to find an optimal plan. In contrast, GWMIN greedily selects one candidate with the highest benefit and eliminates its adjacent candidates from further consideration. For example, for 70 queries, the latency of SHARON is three orders of magnitude higher, while its memory usage is two orders of magnitude larger compared to the greedy optimizer.

Sharing Plan Quality. The greedy optimizer tends to return a sub-optimal sharing plan for two reasons. One, it greedily selects a candidate v with the maximal benefit in each step. By deciding to share v it excludes all candidates adjacent to v even though they may be more beneficial to share than v alone. Two, the greedy optimizer does not resolve sharing conflicts (Chapter 24). However, the sharing opportunities in the original SHARON graph may be rather limited (Figure 5.1).

In Figure 25.5, we vary the number of queries and compare the latency and the memory consumption of our SHARON executor when guided by a greedily chosen plan versus an optimal plan. We run these experiments on the Taxi real data set. The latency of the SHARON executor is reduced 2–fold and its memory consumption decreases 3–fold when 180 queries are processed according to an optimal plan compared to a greedily chosen plan. Thus, an optimal plan ensures real-time light-weight event sequence aggregation.

26

Related Work

Complex Event Processing (CEP) approaches such as SASE [3, 7], Cayuga [28], and ZStream [29] support both event aggregation and event sequence detection over streams. SASE and Cayuga employ a Finite State Automaton (FSA)-based query execution paradigm, meaning that each event query is translated into an FSA. Each run of an FSA corresponds to a query match. In contrast, ZStream translates an event query into an operator tree that is optimized based on rewrite rules. However, these approaches evaluate *each query independently from other queries* in the workload – causing both repeated computations and replicated storage in multi-query settings. Furthermore, they do not optimize event sequence aggregation queries – which is the focus of our work. Thus, they require event sequence construction prior to their aggregation. Since the number of event sequences is polynomial in the number of events per window [2, 3], this *two-step approach* introduces long delays for high-rate streams (Chapter 25). Thus, they do not guarantee real-time responsiveness.

In contrast, A-Seq [2] defines *online* event sequence aggregation that eliminates the event sequence construction step. It incrementally maintains a set of aggregates for each pattern and discards an event once it updated the aggregates. We leverage this idea in our executor (Chapter 22). However, A-Seq has no optimizer to decide which patterns should be shared by which queries in a workload. By default, A-Seq does not share event sequence aggregation among multiple queries. GRETA [6] extends A-Seq by nested Kleene patterns and expressive predicates at the cost of storing of all matched events and their adjacency relationships in a match. Similarly to A-Seq, GRETA optimizes *single* queries.

CEP Multi-Query Optimization (MQO) approaches such as SPASS [38], E-Cube [4], and RUMOR [79] propose event sequence sharing techniques. SPASS exploits event correlation in an event sequence to determine the benefit of shared event sequence construction. E-Cube defines a concept and a pattern hierarchy of event sequence queries and develop both top-down and bottom-up processing of patterns based on the results of other patterns in the hierarchy. RUMOR proposes a rule-based MQO framework for traditional RDBMS and stream processing systems. It defines a set of rules to merge NFAs representing different event queries. However, no optimization techniques for online aggregation of event sequences are proposed by these approaches. By default, they construct all event sequences prior to their aggregation. Event sequence construction introduces polynomial time complexity and thus degrades system performance.

Data Streaming. Streaming systems typically support incremental aggregation [33, 34, 35, 36, 37, 67, 71, 72]. Some of them incrementally aggregate only *raw input events for single-stream queries* [36, 67]. Others share aggregation results among overlapping sliding windows [33, 36] or among multiple queries [35, 71, 72]. However, these approaches evaluate simple Select-Project-Join queries with window semantics over date streams. They do not support CEP-specific operators such as event sequence that treat the order of events as first-class citizens. Typically, they require the *construction of join results* prior to their aggregation.

Multi-Query Optimization techniques include materialized views [80] and common sub-expression sharing [81, 82] in relational databases. However, these approaches do not have the temporal aspect prevalent for CEP queries. Thus, they neither focus on event sequence computation nor their aggregation. Furthermore, they assume that the data is statically stored on disk prior to processing. They neither target in-memory execution nor real-time responsiveness.

Part IV

Conclusions and Future Work

27

Conclusions of This Dissertation

In this dissertation, we propose an event trend analytics methodology that enables real-time yet lightweight detection and aggregation of event sequences of arbitrary length. This functionality is required in a wide range of time-critical streaming applications from financial fraud detection to healthcare analytics. It can be plugged in as a module into existing streaming engines such as Microsoft Stream-Insight [41], Flink [30], Esper [31], and Oracle Stream Analytics [32] to extend their expressive power and optimize their performance. The key contributions of this dissertation can be summarized as follows.

First, we propose the Complete Event Trend (CET) detection approach with minimal CPU time given limited memory. It finds the middle ground between the CPU overhead caused by the re-computation of common sub-sequences in trends and the high memory usage to store and reuse these common sub-sequences. We compactly encode all CETs into the CET graph and find an optimal graph partitioning into time-centric graphlets. We cache CETs per each graphlet and reuse them while constructing CETs within one window and between overlapping sliding windows. Our experimental results demonstrate that our CET approach is up to 42–fold faster than state-of-the-art techniques.

Second, our GRETA approach computes aggregation of event trends that are matched by nested Kleene patterns without constructing these trends. We achieve this goal by compactly encoding all event trends into the GRETA graph and dynamically propagating the aggregates along the edges of the graph during graph construction. We prove that our approach reduces time complexity form exponential to quadratic and space complexity from exponential to linear in the number of events compared to state-of-the-art approaches. Our experiments demonstrate that GRETA achieves up to four orders of magnitude speed-up and requires up to 50–fold less memory than state-of-the-art solutions.

Third, our COGRA approach extends GRETA by rich event matching semantics and further reduces time and space complexity of event trend aggregation. COGRA incrementally maintains event trend aggregates at the coarsest possible granularity level. Thus, it minimizes the number of aggregates and reduces both time and space complexity compared to state-of-the-art approaches. Our experiments demonstrate that COGRA achieves up to four orders of magnitude speed-up and up to eight orders of magnitude memory reduction compared to state-of-the-art approaches.

Lastly, our SHARON approach enables *shared online* event sequence aggregation. The SHARON optimizer encodes sharing candidates, their benefits and conflicts among them into the SHARON graph. Based on the graph, we define three candidate pruning principles to reduce the search space of sharing plans. Our sharing plan finder returns an optimal plan to guide the executor at runtime. Our experiments demonstrate an 18–fold speed-up of SHARON compared to state-of-the-art approaches in diverse scenarios.

28

Future Research Directions

In this chapter, we briefly summarize possible future research directions towards a full-fledged distributed multi-query event trend analytics system (Section 28.1) and 28.2). We will also propose to extend the supported language features (Section 28.3).

28.1 Distributed Event Trend Analytics

The event trend techniques proposed in this dissertation form a solid foundation based upon which distributed solutions could be designed. Indeed, the scalability of these techniques can be further improved by leveraging modern distributed systems such as Flink [30], Spark [83], or Storm [57]. However, additional challenges have to be addressed including how to partition the graph (Section 28.1.1) and how to propagate aggregates in a partitioned graph (Section 28.1.2) to achieve balanced load distribution across computation nodes on a cluster (Section 28.1.3) and reduce communication overhead among the nodes (Section 28.1.4).

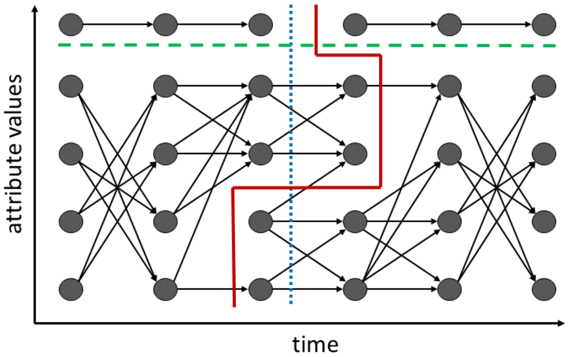


Figure 28.1: Graph partitioning schemas

28.1.1 Graph Partitioning Schemas

Horizontal Graph Partitioning by Event Attribute Values. As described in Section 11.3, a GROUP-BY clause and equivalence predicates partition the stream into non-overlapping sub-streams and the graph is constructed for each sub-stream separately. Thus, one straightforward solution is to partition the graph horizontally by constructing graphlets within each sub-stream on a different cluster node (dashed line in Figure 28.1). Since such graphlets are independent from each other, no communication between the nodes is needed. However, this strategy is not applicable to all queries. Also, such graphlet distribution may be highly unbalanced since different sub-streams rarely have similar rates.

Vertical Graph Partitioning by Time Intervals could leverage our graph partitioning strategy by time proposed in Chapters 6 and 7 (dotted line in Figure 28.1). These time intervals may have different lengths to account for bursty streams. Each node would then be responsible for computing results based on the assigned graphlets. This computation could be done independently by each node. Thus, all nodes could be operating in parallel. The final results could then be constructed using a distributed multi-way join combining partial results (Figure 5.2(c)). In this case, both load balancing and communication overhead reduction would have to be incorporated into the cost model to guide the graph partitioning strategy. The advantage of this solution is that it is query independent.

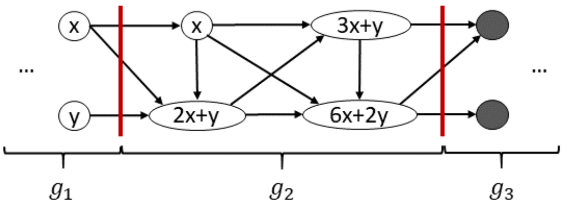


Figure 28.2: Aggregate propagation in a partitioned graph

Diagonal Graph Partitioning by Graph Connectivity would combine both horizontal and vertical graph partitioning. In contrast to them, the diagonal partitioning strategy is not restricted to partition the graph based on event attribute values (like horizontal partitioning) or based on event time stamps (like vertical partitioning). Due to its flexibility to cut the graph at any location, the diagonal partitioning strategy is expected to open up additional optimization opportunities and perform best among these partitioning strategies in various scenarios. For example, we could minimize the overhead of combining partial results per graphlet to form final results across graphlets by cutting as few edges as possible (solid line in Figure 28.1). This intuition must be confirmed by a cost model and an experimental study.

28.1.2 Aggregate Propagation in a Partitioned Graph

As defined in Section 11.1, aggregates are propagated along the edges from previously matched events to more recent events in a dynamic programming fashion. This idea implies sequential propagation of aggregates in the order of events by time stamps. When the graph is partitioned and distributed across machines, previous events may be "cut off" from more recent events (Figure 28.2). Nevertheless, aggregates within different graphlets can be computed in parallel as follows. First, we assign a variable to each last event in graphlet g_1, e.g., x and y in Figure 28.2. We then propagate these variables in all following graphs g_2 and g_3 in parallel as defined in Section 11.1. Once the values of these variables are available, they are plugged into the final aggregation results.

This parallel aggregate propagation will further reduce the latency of event trend aggregation at the cost of storing expressions in each vertex instead of single aggregation values. To speed up the update of these expressions, each expression could be stored as a hash table mapping each variable to its coefficient. For example, the expression $6x + 2y$ corresponds to the mapping $x \mapsto 6$ and $y \mapsto 2$. If k is the number of input variables per graphlet, each hash table has k entries. The time and space complexity of updating and storing an expression in a vertex increases from one to k. Thus, the number of input variables k per graphlet is the major cost factor of parallel aggregate propagation. Our graph partitioning strategy should partition the graph such that k is minimized across all graphlets as further discussed in Section 28.1.4.

28.1.3 Balanced Load Distribution

Event Trend Detection. Once the graph is constructed, the exact number of event trends captured by the graph is known (Chapter 11). The graph-based event trend aggregation has quadratic time complexity in the number of events (Section 11.5) which is negligible compared to the exponential costs of event trend detection (Chapter 5). Based on the number of event trends captured by each graphlet, we could partition the graph in such a way that about the same number of trends is extracted from each graphlet by each node to ensure balanced load distribution.

Event Trend Aggregation. The time complexity of event trend aggregation ranges from linear to quadratic in the number of matched events depending on the granularity level at which the aggregates are maintained (Chapters 11, 15, and 16). Based the aggregate granularity level and the number of matched events, we can partition the stream in such a way that roughly the same load is routed to each node on a cluster.

28.1.4 Communication Overhead Reduction

After each node on a cluster completed its computations based on the assigned graphlets, these partial results have to be combined to form the final results when the graph is partitioned vertically or diagonally (Section 28.1.1). This intermediate result combination introduces the communication overhead among nodes. Generally, the more edges are cut by the graph partitioning strategy and the more partial results have to be combined to form final results, the higher the communication overhead becomes. Based on the number of cut edges and the number of partial results, we need to develop a cost model and leverage this model while selecting a plan that minimizes the communication overhead.

28.2 Multi-Query Event Trend Analytics

While we explored multi-query event sequence aggregation in Part III of this dissertation, multi-query event trend analytics optimization techniques are still subject to future research. Without them, a separate graph would be constructed for each query in the workload. However, similar queries may match the same events or even whole graphlets. The analytics and storage of such graphlets should be shared to reduce both the time and space complexity of the workload.

To enable such shared graphlet processing, we propose to construct a single graph for all queries in the workload and partition it in such a way that graphlets are shared across multiple queries in the workload (Section 28.2.1). These shared graphlets can then be processed in parallel by distributing them across machines as described in Section 28.1. The time and space complexity of event trend aggregation are further reduced if the computation and storage of aggregates are also shared across multiple queries (Section 28.2.2).

28.2.1 Graphlet Sharing

Event trend analytics queries in a workload can differ significantly (Chapter 2). For example, queries q_1 and q_2 in Figure 28.5 have different event matching semantics, predicates, aggregation functions, and patterns. Nevertheless, their graphlets can be shared. Below we briefly sketch the scenarios when such graphlet sharing is possible.

Event Matching Semantics. Assume q_1 and q_2 are evaluated under the skip-till-any-match and skip-till-next-match semantics respectively (Figure 28.5(a)). Events a_1–b_8 are matched by both queries. However, only solid edges hold for both queries. Dashed edges hold only for q_1. Thus, each event must maintain a separate aggregate for each of these queries. While edges and aggregates are saved per each query separately, events a_1–b_8 are stored once and shared by queries q_1 and q_2. This event sharing can significantly reduce the memory costs if a large query workload is evaluated against a high-rate event stream.

Predicates. Let *attr* be an attribute of events of type A. Assume q_2 has a predicate that requires the values of attribute *attr* of adjacent a's to increase, i.e., *A.attr*< NEXT*(A).attr* (Figure 28.5(b)). The values of attribute *attr* are shown as numbers in the top left corner of events of type A. Then, the events a_3 and a_4 do not satisfy this predicate. Consequently, event a_4 is not matched by query q_2 and aggregates for query q_2 go down compared to the case without this predicate (Figure 28.5(a)).

Aggregation Functions. Since aggregates for queries q_1 and q_2 are maintained separately by each event, any aggregation function that is supported by our language (Chapter 2) could be computed by queries q_1 and q_2 (Figure 28.5(c)).

Patterns. The patterns of queries q_1 and q_2 may be different (Figure 28.5(d)). Nevertheless, a large portion of the graph (events $a_1, b_2, a_3, b_6, a_7,$ and b_8 highlighted by solid frame) is matched by both queries.

Event Stream Partitioning. The queries q_1 and q_2 may have different GROUP-BY clauses that partition the stream by the values of different attributes. Assume q_1 partitions

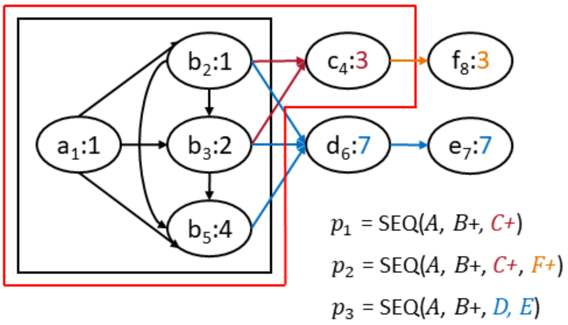


Figure 28.3: Hierarchical prefix sharing

the stream by the value of attribute A_1, while q_2 partitions it by the values of two attributes A_1 and A_2. Then, the stream can be partitioned by the value of A_1 (Partitioning 1). Additionally, each resulting sub-stream is partitioned by the value of A_2 (Partitioning 2). Then, query q_1 could build edges between events that are in different sub-streams according to Partitioning 2, while query q_2 would ignore these edges.

Windows. Queries in the workload may also have different windows. Our goal is to determine whether the state-of-the-art sharing techniques apply [35, 36] to graphlet sharing or a tailored solution is required. Furthermore, we will develop a cost-based sharing benefit model to guide our optimizer during the search for a high-quality graphlet sharing plan for all queries in the workload.

28.2.2 Aggregate Sharing

Conditions for Aggregate Sharing. The benefit of sharing event trend aggregation queries can be increased if each event matched by a set of queries Q maintains a single aggregate for all queries Q. Then, the aggregate computation would be shared across all queries Q – further reducing both time and space complexity. Such aggregate sharing is possible if the following conditions hold: (1) Queries compute the same aggregation function. (2) They have a common pattern prefix. (3) Queries are evaluated under the same event matching semantics. (4) They have the same windows, GROUP-BY clauses, and predicates. We will prove that these conditions guarantee that a graph prefix of several queries is exactly the same, i.e., it has same vertices and edges. Thus, the aggregates will also have same values and can be shared.

We need to explore whether these strict conditions could be relaxed. For example, queries may have overlapping windows and grouping attributes. Also, their predicates may be not mutually exclusive.

Hierarchical Prefix Sharing. Since queries may have different patterns, we propose hierarchical prefix sharing that allows sharing maximal graph prefixes by different queries. For example, the graphlet for the sub-pattern $\textsf{SEQ}(A+, B)$ is highlighted by the black frame in Figure 28.3. It is shared by the queries with patterns p_1, p_2, and p_3. At the same time, the graphlet for a longer sub-pattern $\textsf{SEQ}(A+, B, C+)$, highlighted by the red frame, is shared by queries with patterns p_1 and p_2.

Lazy Split of Aggregates. Since these queries may have different predicates, we introduce the lazy split of aggregates that allows maximal sharing of aggregates. Assume queries q_1 and q_2 have predicates θ_1 and θ_2 respectively. As soon as an event e arrives that satisfies θ_1 but not θ_2, the aggregates of e and all following events are maintained per each query q_1 and q_2 separately until the end of the current window.

Graph Sharing Optimizer. As descriptions above illustrate, in same cases the sharing decisions are made statically (e.g., hierarchical graph prefix sharing) while in other cases runtime decisions are necessary (e.g., lazy split of aggregates). We will develop a cost model and a graph sharing optimizer that seamlessly supports both static and dynamic sharing decisions.

28.3 Additional Language Features

In addition to the language features described in Chapters 2 and 12, we now briefly sketch possible extensions of our event query language by other event matching semantics (Section 28.3.1) and predicates of non-adjacent events in a trend (Section 28.3.2).

28.3.1 Alternative Event Matching Semantics

As described in Section 2.4, the skip-till-next-match and contiguous semantics are very restrictive, while skip-till-any-match detects an exponential number of trends. To avoid missing valuable trends, an application is forced to use skip-till-any-match and filter the resulting set of all trends using windows, GROUP-BY, and predicates.

Example 28.1 *Given the stream of price records* $I = \{10, 2, 9, 8, 7, 1, 6, 5, 4, 3\}$*, skip-till-any-match is the only semantics that detects the down-trend* $(10, 9, 8, 7, 6, 5, 4, 3)$ *by ignoring local fluctuations 2 and 1. Since longer stock trends are considered to be more reliable [84], this long trend can be more valuable to an algorithmic trading system than three shorter trends* $(10, 2)$*,* $(9, 8, 7, 1)$*, and* $(6, 5, 4, 3)$ *detected under the skip-till-next-match semantics.*

However, the skip-till-any-match semantics is prohibitively expensive [3, 5, 6]. It detects many trends that are subsequently deleted – wasting valuable resources. Thus, a possible future research direction is to define an alternative event matching semantics that is more flexible than skip-till-next-match, yet less expensive than skip-till-any-match.

Context-Aware Event Matching Semantics. Alternatively, we could switch between different semantics depending on the application context [85]. That is, once a situation of interest to the application is detected under one semantics, we could switch the context and process all events arriving during this context under a different semantics. In this

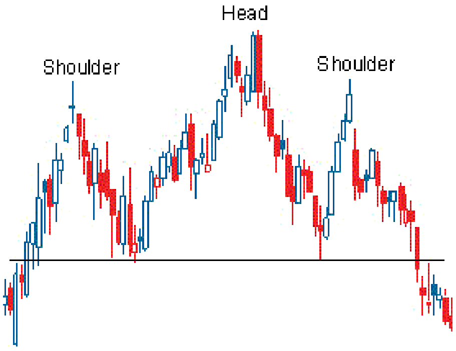


Figure 28.4: Head and shoulders stock trend (figure is from [1])

way, we would minimize the number of queries and shorten the time intervals during which these queries are evaluated under the skip-till-any-match semantics.

28.3.2 Predicates on Non-Adjacent Events in a Trend

In this dissertation, we focus on predicates on single events and predicates on adjacent events (Section 2.5). We also sketch how predicates of minimal length of a trend could be supported in Chapter 12. We left predicates on non-adjacent events for future research. They are useful in many streaming applications.

Example 28.2 *The query q'' computes the number of head-and-shoulders stock trends per sector during a time window of 1 hour that slides every minute. A head-and-shoulders stock trend is a sequence of single transactions as well as up and down sub-trends illustrated in Figure 28.4 and spesified in the* PATTERN *clause of the query. These sub-trends are of arbitrary length which is expressed by the Kleene closure operator. The query must be flexible enough to ignore price fluctuations which is enabled by the skip-till-next-match semantics. All events in the same head-and-shoulders trend must carry the same sector and company identifiers, expressed by the predicate [sector, company]. Other predicates specify the price variation in a trend. For example, the last 3 predicates require U to be an up trend between the start S and the left shoulder L of a trend. Similar predicates* *are necessary to restrict other sub-trends. They are omitted in q'' for compactness. The complete query can be found in [59].*

Query q'' must ensure that the price measurement H representing the head of a trend is higher than the price measurements L and R representing the left and right shoulders. However, the events matched by H, L, and R respectively are not adjacent in a trend.

```
q'' : RETURN sector, COUNT(*)
    PATTERN SEQ(S, U+, L, D+, X, U'+, H, D'+, Y, U''+, R, D''+, E)
    SEMANTICS skip-till-next-match
    WHERE [sector, company] AND S.price < U.price AND U.price < NEXT(U).price
          AND U.price < L.price
    GROUP-BY sector WITHIN 1 hour SLIDE 1 minute
```

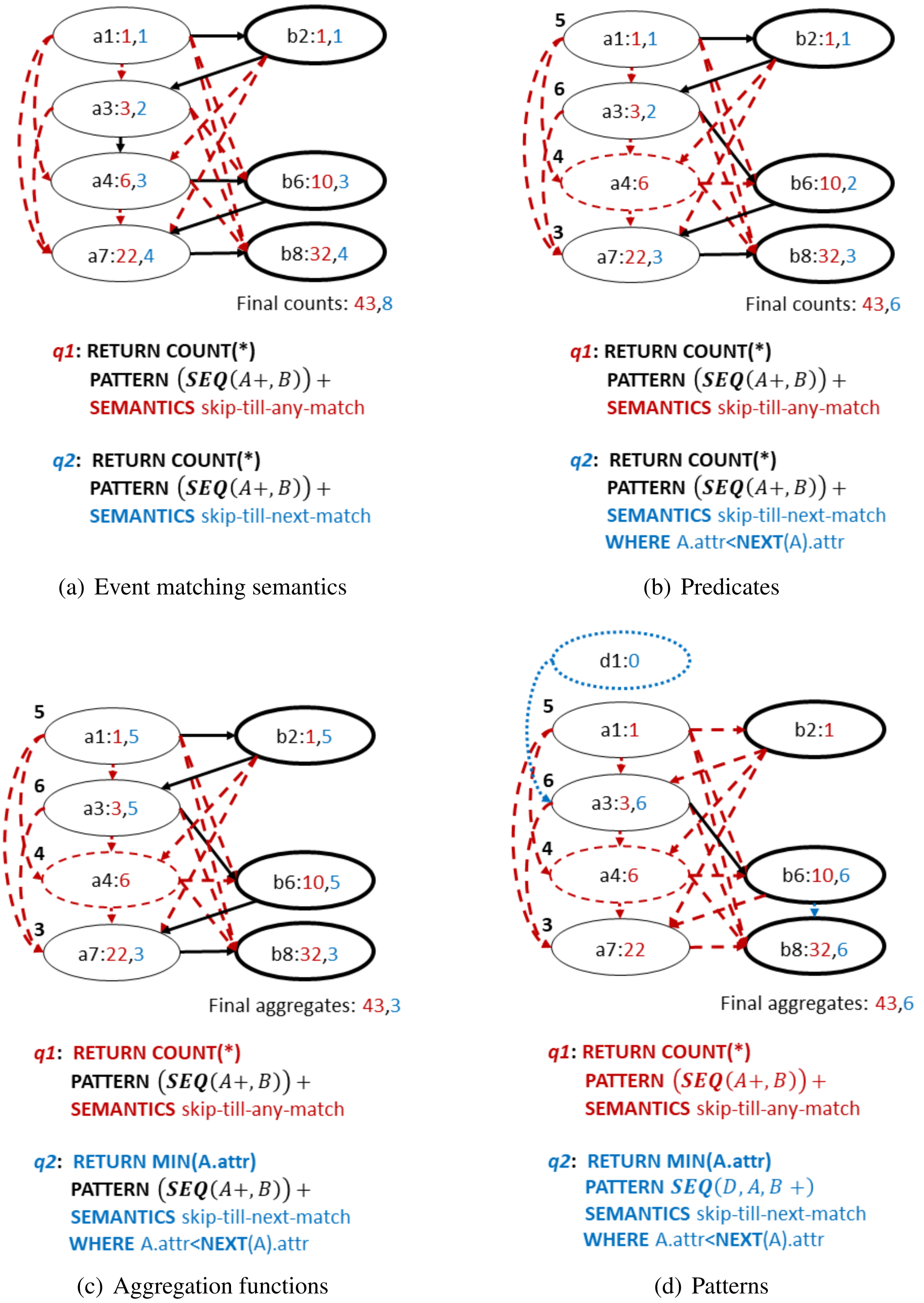
Predicates on non-adjacent events are expensive to evaluate since they determine event relevance relatively to previously matched events. Since these events can be anywhere in the graph, the graph must be traversed for each new event in the worst case. This traversal may significantly degrade system performance. Thus, special optimization techniques have to be designed to enable efficient evaluation of such predicates.

Postponed Predicates on Non-Adjacent Events. One possible solution would be to postpone predicates on non-adjacent events until event trend construction phase as follows. First, we break the pattern into sub-patterns such that each sub-pattern does not have predicates on non-adjacent events. In our example, these sub-patterns are SEQ$(S, U+, L)$, SEQ$(L, D+, X)$, SEQ$(X, U'+, H)$, etc. Then, we could maintain a separate graph for each sub-pattern. Lastly, we evaluate the predicates on non-adjacent events while connecting these graphlets. However, generating many invalid sub-trends will waste valuable computational resources. Also, events would be replicated across different graphlets provoking significant CPU and memory overhead. Thus, more efficient solutions are have to be developed.

Pivotal Points. Another possible solution is to determine a set of pivotal points. In our example, these points are the head H, the left and right shoulders L and R, the start and end of a trend S and E, and the points X and Y defining the neckline. Then, several trends may share the same set of pivotal points. When a new event e arrives, it is compared to its pivotal points to conclude which existing event trends the event e will extend.

Since several events may be a pivotal point, the following challenges have to be tackled. On the one hand, selecting a set of pivotal points reduces the number of detected event trends. On the other hand, considering all possible combinations of pivotal points would introduce exponential time overhead. Thus, we need an expressive yet efficient way of determining pivotal points and detecting event trends with respect to these points.

We will study the advantages and drawbacks of these (and other) possible solutions to determine the most effective option.



q1: RETURN COUNT(*)
PATTERN $(SEQ(A+,B))+$
SEMANTICS skip-till-any-match

q2: RETURN COUNT(*)
PATTERN $(SEQ(A+,B))+$
SEMANTICS skip-till-next-match

(a) Event matching semantics

q1: RETURN COUNT(*)
PATTERN $(SEQ(A+,B))+$
SEMANTICS skip-till-any-match

q2: RETURN COUNT(*)
PATTERN $(SEQ(A+,B))+$
SEMANTICS skip-till-next-match
WHERE A.attr<NEXT(A).attr

(b) Predicates

q1: RETURN COUNT(*)
PATTERN $(SEQ(A+,B))+$
SEMANTICS skip-till-any-match

q2: RETURN MIN(A.attr)
PATTERN $(SEQ(A+,B))+$
SEMANTICS skip-till-next-match
WHERE A.attr<NEXT(A).attr

(c) Aggregation functions

q1: RETURN COUNT(*)
PATTERN $(SEQ(A+,B))+$
SEMANTICS skip-till-any-match

q2: RETURN MIN(A.attr)
PATTERN $SEQ(D,A,B+)$
SEMANTICS skip-till-next-match
WHERE A.attr<NEXT(A).attr

(d) Patterns

Figure 28.5: Graphlet sharing by queries q_1 and q_2 evaluated against the stream I = { *d1, a1, b2, a3, a4, b6, a7, b8* }

References

[1] Head-and-shoulders stock pattern. `http://www.investopedia.com/terms/h/head-shoulders.asp`. xviii, 197

[2] Yingmei Qi, Lei Cao, Medhabi Ray, and Elke A. Rundensteiner. Complex event analytics: Online aggregation of stream sequence patterns. In *SIGMOD*, pages 229–240, 2014. xix, 7, 8, 9, 15, 16, 27, 28, 101, 135, 136, 140, 141, 144, 149, 173, 175, 177, 182

[3] Haopeng Zhang, Yanlei Diao, and Neil Immerman. On complexity and optimization of expensive queries in Complex Event Processing. In *SIGMOD*, pages 217–228, 2014. xix, 1, 6, 7, 8, 9, 10, 13, 15, 19, 24, 27, 73, 74, 83, 112, 113, 135, 136, 144, 177, 182, 196

[4] Mo Liu, Elke A. Rundensteiner, Kara Greenfield, Chetan Gupta, Song Wang, Ismail Ari, and Abhay Mehta. E-Cube: Multi-dimensional event sequence analysis using hierarchical pattern query sharing. In *SIGMOD*, pages 889–900, 2011. 1, 8, 74, 83, 144, 183

[5] Olga Poppe, Chuan Lei, Salah Ahmed, and Elke A. Rundensteiner. Complete event trend detection in high-rate event streams. In *SIGMOD*, pages 109–124, 2017. 1, 73, 112, 196

[6] Olga Poppe, Cuan Lei, Elke A. Rundensteiner, and David Maier. GRETA: Graph-based Real-time Event Trend Aggregation. In *VLDB*, pages 80–92, 2018. 1, 112, 135, 136, 137, 141, 183, 196

[7] Jagrati Agrawal, Yanlei Diao, Daniel Gyllstrom, and Neil Immerman. Efficient pattern matching over event streams. In *SIGMOD*, pages 147–160, 2008. 1, 3, 6, 7, 10, 19, 24, 73, 74, 83, 84, 144, 182

[8] Eugene Wu, Yanlei Diao, and Shariq Rizvi. High-performance Complex Event Processing over streams. In *SIGMOD*, pages 407–418, 2006. 1, 6, 7, 10, 19, 24, 73, 83, 84

[9] Wikipedia. `https://en.wikipedia.org/wiki/Check_kiting`. 2

[10] The Press Enterprise. `http://www.pe.com/articles/checks-694614-people-bank.html`. 2

[11] Cleveland Clinic. `https://my.clevelandclinic.org/health/articles/sudden-cardiac-death`, 2017. 4

[12] Unified New York City Taxi and Uber data. `https://github.com/toddwschneider/nyc-taxi-data`. 4, 16, 174

[13] Uber TLC FOIL Response. `https://github.com/fivethirtyeight/uber-tlc-foil-response`. 4

[14] Uber Releases Hourly Ride Numbers In New York City To Fight De Blasio. `https://techcrunch.com/2015/07/22/uber-releases-hourly-ride-numbers-in-new-york-city-to-fight-de-blasio/`. 4

[15] Oracle database. `https://www.oracle.com/database/`. 6, 84

[16] MySQL. `https://www.mysql.com/`. 6, 84

[17] Microsoft SQL server. `https://www.microsoft.com/en-us/sql-server/`. 6, 84

[18] Alberto Lerner and Dennis Shasha. AQuery: Query language for ordered data, optimization techniques, and experiments. In *VLDB*, pages 345–356, 2003. 6, 84, 145

[19] Eric Lo, Ben Kao, Wai-Shing Ho, Sau Dan Lee, Chun Kit Chui, and David W. Cheung. OLAP on sequence data. In *SIGMOD*, pages 649–660, 2008. 6, 84, 145

[20] Badrish Chandramouli, Jonathan Goldstein, and David Maier. On-the-fly progress detection in iterative stream queries. *VLDB*, 2(1):241–252, 2009. 6, 84

[21] Yi Chen, S. B. Davidson, and Yifeng Zheng. An Efficient XPath Query Processor for XML Streams. In *ICDE*, pages 1–12, 2006. 6, 84

[22] Todd J. Green, Ashish Gupta, Gerome Miklau, Makoto Onizuka, and Dan Suciu. Processing XML streams with deterministic automata and stream indexes. *ACM Trans. Database Syst.*, 29(4):752–788, 2004. 6, 84

[23] Mengmeng Liu, Nicholas E. Taylor, Wenchao Zhou, Zachary G. Ives, and Boon Thau Loo. Recursive computation of regions and connectivity in networks. In *ICDE*, pages 1108–1119, 2009. 6, 84

[24] Bertram Ludäscher, Pratik Mukhopadhyay, and Yannis Papakonstantinou. A transducer-based XML query processor. In *VLDB*, pages 227–238, 2002. 6, 84

[25] Barzan Mozafari, Kai Zeng, and Carlo Zaniolo. From regular expressions to nested words: Unifying languages and query execution for relational and XML sequences. *VLDB*, 3(1-2):150–161, 2010. 6, 84

[26] Barzan Mozafari, Kai Zeng, and Carlo Zaniolo. High-performance complex event processing over XML streams. In *SIGMOD*, pages 253–264, 2012. 6, 84

[27] Alexander Shkapsky, Mohan Yang, Matteo Interlandi, Hsuan Chiu, Tyson Condie, and Carlo Zaniolo. Big data analytics with Datalog queries on Spark. In *SIGMOD*, pages 1135–1149, 2016. 6, 84

[28] Alan Demers, Johannes Gehrke, Biswanath Panda, Mirek Riedewald, Varun Sharma, and Walker White. Cayuga: A general purpose event monitoring system. In *CIDR*, pages 412–422, 2007. 6, 7, 8, 83, 144, 182

[29] Yuan Mei and Samuel Madden. ZStream: A Cost-based Query Processor for Adaptively Detecting Composite Events. In *SIGMOD*, pages 193–206, 2009. 6, 7, 8, 83, 144, 182

[30] Flink. `https://flink.apache.org/`. 7, 8, 12, 13, 15, 16, 17, 73, 112, 113, 135, 145, 175, 186, 188

[31] ESPER 2009. http://esper.codehaus.org/. 7, 12, 17, 73, 113, 135, 145, 186

[32] Oracle Stream Analytics. `http://www.oracle.com/technetwork/middleware/complex-event-processing/documentation/index.html`. 7, 12, 17, 135, 145, 186

[33] Arvind Arasu and Jennifer Widom. Resource sharing in continuous sliding-window aggregates. In *VLDB*, pages 336–347, 2004. 8, 102, 145, 173, 183

[34] Thanaa M. Ghanem, Moustafa A. Hammad, Mohamed F. Mokbel, Walid G. Aref, and Ahmed K. Elmagarmid. Incremental evaluation of sliding-window queries over data streams. *IEEE Trans. on Knowl. and Data Eng.*, 19(1):57–72, 2007. 8, 145, 183

[35] S. Krishnamurthy, C. Wu, and M. J. Franklin. On-the-fly sharing for streamed aggregation. In *SIGMOD*, pages 623–634, 2006. 8, 145, 173, 183, 194

[36] Jin Li, David Maier, Kristin Tufte, Vassilis Papadimos, and Peter A. Tucker. No pane, no gain: Efficient evaluation of sliding window aggregates over data streams. In *SIGMOD*, pages 39–44, 2005. 8, 102, 145, 173, 183, 194

[37] Kanat Tangwongsan, Martin Hirzel, Scott Schneider, and Kun-Lung Wu. General incremental sliding-window aggregation. In *VLDB*, pages 702–713, 2015. 8, 145, 183

[38] Medhabi Ray, Chuan Lei, and Elke A. Rundensteiner. Scalable pattern sharing on event streams. In *SIGMOD*, pages 495–510, 2016. 8, 16, 74, 83, 175, 183

[39] Arvind Arasu, Mitch Cherniack, Eduardo Galvez, David Maier, Anurag S. Maskey, Esther Ryvkina, Michael Stonebraker, and Richard Tibbetts. Linear road: A stream data management benchmark. In *VLDB*, pages 480–491, 2004. 9, 16, 111, 174

[40] Martin Klazar. Bell numbers, their relatives, and algebraic differential equations. *J. Comb. Theory, Ser. A*, 102(1):63–87, 2003. 10, 54, 60

[41] Microsoft StreamInsight. `https://technet.microsoft.com/en-us/library/ee362541%28v=sql.111%29.aspx`. 12, 17, 73, 113, 186

[42] Attila Reiss and Didier Stricker. Creating and benchmarking a new dataset for physical activity monitoring. In *PETRA*, pages 40:1–40:8, 2012. 13, 15, 72, 136

[43] Stock data. `http://davis.wpi.edu/datasets/Stock_Trace_Data/`. 13, 15, 72, 111, 137

[44] Shuichi Sakai, Mitsunori Togasaki, and Koichi Yamazaki. A note on greedy algorithms for the maximum weighted independent set problem. *Discrete Appl. Math.*, 126(2-3):313–322, 2003. 16, 159, 160, 176

[45] Flink Forum. `https://issues.apache.org/jira/browse/FLINK-3318`. 17, 73

[46] Oracle. `https://oracle-base.com/articles/12c/pattern-matching-in-oracle-database-12cr1`, 2016. [Online; accessed 24-November-2016]. 17

[47] Nihal Dindar. Pattern matching over sequences of rows in a relational database system. Master's thesis, ETH Zurich, 2008. 17

[48] Jim Gray, Surajit Chaudhuri, Adam Bosworth, Andrew Layman, Don Reichart, Murali Venkatrao, Frank Pellow, and Hamid Pirahesh. Data Cube: A Relational Aggregation Operator Generalizing Group-By, Cross-Tab, and Sub-Totals. *Data Min. Knowl. Discov.*, 1(1):29–53, 1997. 28

[49] Liping Peng, Yanlei Diao, and Anna Liu. Optimizing probabilistic query processing on continuous uncertain data. *PVLDB*, 4(11):1169–1180, 2011. 31

[50] Thanh T. Tran, Liping Peng, Yanlei Diao, Andrew Mcgregor, and Anna Liu. Claro: Modeling and processing uncertain data streams. *PVLDB*, 21(5):651–676, 2012. 31

[51] Badrish Chandramouli, Jonathan Goldstein, and David Maier. High-performance dynamic pattern matching over disordered streams. *PVLDB*, 3(1):220–231, 2010. 31

[52] Jin Li, Kristin Tufte, Vladislav Shkapenyuk, Vassilis Papadimos, Theodore Johnson,

and David Maier. Out-of-order processing: a new architecture for high-performance stream systems. *PVLDB*, 1(1):274–288, 2008. 31

[53] Mo Liu, Ming Li, Denis Golovnya, Elke A. Rundensteiner, and Kajal T. Claypool. Sequence pattern query processing over out-of-order event streams. In *ICDE*, pages 784–795, 2009. 31

[54] Utkarsh Srivastava and Jennifer Widom. Flexible time management in data stream systems. In *PODS*, pages 263–274, 2004. 31

[55] James Stewart. *Calculus: Early Transcendentals*. Thompson Brooks/Cole, 8th edition, 2015. 36

[56] Isabelle Stanton and Gabriel Kliot. Streaming graph partitioning for large distributed graphs. In *KDD*, pages 1222–1230, 2012. 55, 84, 85

[57] Storm. `https://storm.apache.org/`. 73, 188

[58] Medhabi Ray, Elke A. Rundensteiner, Mo Liu, Chetan Gupta, Song Wang, and Ismail Ari. High-performance complex event processing using continuous sliding views. In *EDBT*, pages 525–536, 2013. 74, 83

[59] Cagri Balkesen, Nihal Dindar, Matthias Wetter, and Nesime Tatbul. RIP: Run-based intra-query parallelism for scalable Complex Event Processing. In *DEBS*, pages 3–14, 2013. 83, 198

[60] Joel Nishimura and Johan Ugander. Restreaming graph partitioning: Simple versatile algorithms for advanced balancing. In *KDD*, pages 1106–1114, 2013. 84

[61] Charalampos E. Tsourakakis, Christos Gkantsidis, Bozidar Radunovic, and Milan Vojnovic. Fennel: Streaming graph partitioning for massive scale graphs. Technical report, 2012. 84, 85

[62] Konstantin Andreev and Harald Räcke. Balanced graph partitioning. In *SPAA*, pages 120–124, 2004. 84, 85

[63] Robert Krauthgamer, Joseph (Seffi) Naor, and Roy Schwartz. Partitioning graphs into balanced components. In *SODA*, pages 942–949, 2009. 84

[64] Stephen T. Barnard. PMRSB: Parallel Multilevel Recursive Spectral Bisection. In *Supercomputing*, 1995. 84

[65] Bruce Hendrickson and Robert Leland. A multilevel algorithm for partitioning graphs. In *Supercomputing*, 1995. 84, 85

[66] George Karypis and Vipin Kumar. Multilevel graph partitioning schemes. In *Parallel Processing*, pages 113–122. CRC Press, 1995. 84, 85

[67] Jin Li, David Maier, Kristin Tufte, Vassilis Papadimos, and Peter A. Tucker. Semantics and evaluation techniques for window aggregates in data streams. In *SIGMOD*, pages 311–322, 2005. 100, 145, 183

[68] John Meehan, Nesime Tatbul, Stan Zdonik, Cansu Aslantas, Ugur Cetintemel, Jiang Du, Tim Kraska, Samuel Madden, David Maier, Andrew Pavlo, Michael Stonebraker, Kristin Tufte, and Hao Wang. S-Store: Streaming Meets Transaction Processing. In *VLDB*, pages 2134–2145, 2015. 103

[69] Uwe Schöning. *Theoretische Informatik - kurzgefaßt (3. Aufl.)*. Spektrum Akademischer Verlag, 1997. 108

[70] Google Dataflow. `https://cloud.google.com/dataflow/`. 113

[71] Rui Zhang, Nick Koudas, Beng Chin Ooi, and Divesh Srivastava. Multiple aggregations over data streams. In *SIGMOD*, pages 299–310, 2005. 145, 183

[72] Rui Zhang, Nick Koudas, Beng Chin Ooi, Divesh Srivastava, and Pu Zhou. Streaming multiple aggregations using phantoms. In *VLDB*, pages 557–583, 2010. 145, 183

[73] Iakovos Motakis and Carlo Zaniolo. Temporal aggregation in active database rules. In *SIGMOD*, pages 440–451, 1997. 145

[74] Reza Sadri, Carlo Zaniolo, Amir Zarkesh, and Jafar Abidi. Expressing and optimizing sequence queries in database systems. In *ACM Trans. on Database Systems*, pages 282–318, 2004. 145

[75] Praveen Seshadri, Miron Livny, and Raghu Ramakrishnan. Seq: Design and implementation of a sequence database system. In *VLDB*, pages 99–110, 1996. 145

[76] Olga Poppe, Allison Rozet, Chuan Lei, Elke A. Rundensteiner, and David Maier. Sharon: Shared Online Event Sequence Aggregation. `http://users.wpi.edu/~opoppe/papers/Sharon-full.pdf`, 2017. Technical report in progress. 149, 159, 175

[77] R.M. Karp. Reducibility among combinatorial problems. In *Complexity of Computer Computations*, pages 85–103. Plenum Press, 1972. 159

[78] Rakesh Agrawal and Ramakrishnan Srikant. Fast algorithms for mining association rules. In *VLDB*, pages 487–499, 1994. 165

[79] Mingsheng Hong, Mirek Riedewald, Christoph Koch, Johannes Gehrke, and Alan Demers. Rule-based multi-query optimization. In *EDBT*, pages 120–131, 2009. 183

[80] Surajit Chaudhuri, Ravi Krishnamurthy, Spyros Potamianos, and Kyuseok Shim. Optimizing queries with materialized views. In *ICDE*, pages 190 – 200, 1995. 183

[81] Upen S Chakravarthy and Jack Minker. Multiple query processing in deductive databases using query graphs. In *VLDB*, pages 384–391, 1986. 184

[82] Georgios Giannikis, Philipp Unterbrunner, Jeremy Meyer, Gustavo Alonso, Dietmar Fauser, and Donald Kossmann. Crescando. In *SIGMOD*, pages 1227–1230, 2010. 184

[83] Spark. `https://spark.apache.org/`. 188

[84] Arshad Khan. *501 Stock Market Tips and Guidelines*. Writers Club Press, 2002. 196

[85] Olga Poppe, Chuan Lei, Elke Rundensteiner, and Dan Dougherty. Context-aware event stream analytics. In *EDBT*, pages 413–424, 2016. 196