

Accelerating Event Stream Processing in On- and Offline Systems

Dissertation

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

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

M.Sc. Informatik
Michael Körber
geboren in Gießen

Marburg, im September 2021

Originaldokument gespeichert auf dem Publikationsserver der
Philipps-Universität Marburg
<http://archiv.ub.uni-marburg.de>



Dieses Werk bzw. Inhalt steht unter einer
Creative Commons
Namensnennung
Keine kommerzielle Nutzung
Weitergabe unter gleichen Bedingungen
3.0 Deutschland Lizenz.

Die vollständige Lizenz finden Sie unter:
<http://creativecommons.org/licenses/by-nc-sa/3.0/de/>

Vom Fachbereich Mathematik und Informatik der Philipps-Universität Marburg
(Hochschulkennziffer 1180) als Dissertation am 01.12.2021 angenommen.

Erstgutachter: Prof. Dr. Bernhard Seeger, Philipps-Universität Marburg

Zweitgutachter: Prof. Dr. Thorsten Papenbrock, Philipps-Universität Marburg

Tag der Einreichung: 17.09.2021.

Tag der mündlichen Prüfung: 17.12.2021.

— *Dedicated to Jeannine, Lara & Ella.*

Abstract

Due to a growing number of data producers and their ever-increasing data volume, the ability to ingest, analyze, and store potentially never-ending streams of data is a mission-critical task in today’s data processing landscape. A widespread form of data streams are event streams, which consist of continuously arriving notifications about some real-world phenomena. For example, a temperature sensor naturally generates an event stream by periodically measuring the temperature and reporting it with measurement time in case of a substantial change to the previous measurement.

In this thesis, we consider two kinds of event stream processing: online and offline. Online refers to processing events solely in main memory as soon as they arrive, while offline means processing event data previously persisted to non-volatile storage. Both modes are supported by widely used scale-out general-purpose stream processing engines (SPEs) like Apache Flink or Spark Streaming. However, such engines suffer from two significant deficiencies that severely limit their processing performance. First, for offline processing, they load the entire stream from non-volatile secondary storage and replay all data items into the associated online engine in order of their original arrival. While this naturally ensures unified query semantics for on- and offline processing, the costs for reading the entire stream from non-volatile storage quickly dominate the overall processing costs. Second, modern SPEs focus on scaling out computations across the nodes of a cluster, but use only a fraction of the available resources of individual nodes. This thesis tackles those problems with three different approaches.

First, we present novel techniques for the offline processing of two important query types (windowed aggregation and sequential pattern matching). Our methods utilize well-understood indexing techniques to reduce the total amount of data to read from non-volatile storage. We show that this improves the overall query runtime significantly. In particular, this thesis develops the first index-based algorithms for pattern queries expressed with the `MATCH_RECOGNIZE` clause, a new and powerful language feature of SQL that has received little attention so far.

Second, we show how to maximize resource utilization of single nodes by exploiting the capabilities of modern hardware. Therefore, we develop a prototypical shared-memory CPU-GPU-enabled event processing system. The system provides implementations of all major event processing operators (filtering, windowed aggregation, windowed join, and sequential pattern matching). Our experiments reveal that regarding resource utilization and processing throughput, such a hardware-enabled system is superior to hardware-agnostic general-purpose engines.

Finally, we present *TPStream*, a new operator for pattern matching over temporal intervals. *TPStream* achieves low processing latency and, in contrast to sequential pattern matching, is easily parallelizable even for unpartitioned input streams. This results in maximized resource utilization, especially for modern CPUs with multiple cores.

Zusammenfassung

Die stetig wachsende Anzahl an Datenproduzenten und deren ständig wachsendes Datenvolumen macht die Erfassung, Analyse und das Speichern potenziell unendlicher Datenströme zu entscheidenden Aufgaben der heutigen Datenverarbeitungslandschaft. Eine weit verbreitete Form von Datenströmen sind Ereignisströme, die aus kontinuierlich eintreffenden Meldungen über bestimmte Ereignisse der realen Welt bestehen. So erzeugt beispielsweise ein Temperatursensor einen Ereignisstrom, indem er regelmäßig die Temperatur misst und diese mit der Messzeit meldet, falls sich eine wesentliche Änderung gegenüber der vorherigen Messung ergibt.

In dieser Arbeit betrachten wir zwei Arten der Ereignisstromverarbeitung: online und offline. Online bedeutet, dass die Ereignisse verarbeitet werden, sobald sie eintreffen. Die Verarbeitung erfolgt hierbei ausschließlich im Hauptspeicher. Offline bedeutet hingegen, dass die Ereignisdaten zuvor in einem nichtflüchtigen Speicher abgelegt werden. Moderne scale-out Datenstromsysteme wie Apache Flink oder Spark Streaming unterstützen beide Verarbeitungsmodi. Allerdings weisen solche Systeme zwei gravierende Schwächen auf, welche Ihre Leistung negativ beeinträchtigen. Erstens laden sie für die Offline-Verarbeitung den gesamten Datenstrom vom Sekundärspeicher und überführen alle Daten sequentiell in die zugehörige Online-Engine. Hierbei dominieren die Kosten für den Sekundärspeicherzugriff schnell die gesamten Verarbeitungskosten. Zweitens konzentrieren sich diese Systeme auf die Verteilung von Berechnungen über die Knoten eines Clusters, nutzen jedoch die verfügbaren Ressourcen einzelner Knoten nur zu einem Bruchteil. Diese Arbeit beschreibt drei Ansätze zum Lösen dieser Probleme.

Zunächst werden neuartige Techniken zur Offline-Verarbeitung von zwei wichtigen Anfragetypen (fensterbasierte Aggregation und sequenzieller Musteranfragen) vorgestellt. Wir nutzen gut erforschte Indexierungstechniken, um die Gesamtmenge der aus dem Sekundärspeicher zu lesenden Daten zu reduzieren und zeigen, dass sich dadurch die Gesamtlaufzeit der Anfragen erheblich verbessert. Insbesondere entwickeln wir die ersten indexbasierten Algorithmen für Musteranfragen, die mit der `MATCH_RECOGNIZE`-Klausel ausgedrückt werden, einem neuen und leistungsstarken SQL Feature, das bisher nur wenig Beachtung gefunden hat.

Zweitens zeigen wir, wie man die Ressourcenauslastung einzelner Knoten maximieren kann, indem man die Möglichkeiten moderner Hardware voll ausschöpft. Zu diesem Zweck entwickeln wir ein prototypisches Shared-Memory-CPU-GPU Ereignisverarbeitungssystem. Das System bietet Implementierungen aller wichtigen Ereignisverarbeitungsoperatoren (Filterung, fensterbasierte Aggregation, fensterbasierter Join und sequenzielle Musteranfragen). In Experimenten zeigen wir, dass ein solch hardwarenahes System in Bezug auf die Ressourcennutzung und den Verarbeitungsdurchsatz den hardwareunabhängigen Systemen überlegen ist.

Schließlich stellen wir *TPStream* vor, einen neuen Operator für Musteranfragen über zeitliche Intervalle. *TPStream* benötigt zum Erstellen von Ergebnissen nur eine sehr geringe Latenz und ist im Gegensatz zu sequenziellen Musteranfragen auch für unpartitionierte Eingabeströme leicht parallelisierbar. Dies führt zu einer gesteigerten Ressourcennutzung, insbesondere bei modernen CPUs mit mehreren Kernen.

Erklärung

Hiermit versichere ich, dass ich meine Dissertation mit dem Titel

Accelerating Event Stream Processing in On- and Offline Systems

selbständig und ohne fremde Hilfe verfasst, nicht andere als die in ihr angegebenen Quellen oder Hilfsmittel benutzt, alle vollständig oder sinngemäß übernommenen Zitate als solche gekennzeichnet sowie die Dissertation in der vorliegenden oder einer ähnlichen Form noch bei keiner anderen in- oder ausländischen Hochschule anlässlich eines Promotionsgesuchs oder zu anderen Prüfungszwecken eingereicht habe. Dies ist mein erster Versuch einer Promotion.

Marburg, den 17.09.2021

Michael Körber

Acknowledgments

When I started my academic journey back in 2010, I never thought that I would end up here. However, without the help and support from my colleagues, family, and friends, this work would not have been possible.

First of all, I want to thank my thesis advisor Prof. Dr. Bernhard Seeger, for guiding me through the course of this thesis. The discussions with him were always inspiring and helped me to generate new ideas and overcome the problems I faced.

I also want to thank my colleagues from the Database Research Group of the University of Marburg. Whether I got stuck with theoretical problems, required help with an implementation, or asked for proofreading of a manuscript, I received support from everyone. I particularly want to thank my roommate Nikolaus “Niki” Glombiewski and Jana Holznigenkemper. Niki supported me in all my research, and his superpower to find every single weak spot of an idea or manuscript greatly strengthened my work. Jana always helped me out with her math skills and greatly improved my writings due to her careful proofreading.

I would like to thank my family and friends. They motivated me to take this step and always supported me during this journey. Ahead of everyone, I thank my wife, Jeannine. She set me straight whenever I got stuck and took care of our two girls, Lara and Ella, during my nightly writing sessions. Finally, I also have to thank my two little girls. Some of the best ideas popped up while pushing them on the swing, building yet another Lego castle, or brushing Barbie’s hair.

Contents

Abstract	v
Zusammenfassung	vi
Erklärung	vii
Acknowledgments	viii
1 Introduction	1
1.1 Continuous Queries	2
1.2 On- and Offline Processing	3
1.3 Motivation & Contributions	4
1.3.1 Index-Supported Offline Processing	6
1.3.2 iGPU-Accelerated Online Processing	6
1.3.3 Pattern Matching on Temporal Intervals	7
1.4 Publications	9
1.5 Thesis Structure	11
2 Preliminaries	12
2.1 Data Streams	12
2.2 Event Streams	13
2.3 Event Stream Processing	14
2.3.1 Windows	16
2.3.2 Filtering	18
2.3.3 Windowed Aggregation	20
2.3.4 Windowed Join	21
2.3.5 Sequential Pattern Matching	23
2.4 Situations	28
3 Index-Supported Offline Processing	29
3.1 Related Work	31
3.1.1 Windowed Aggregation	31
3.1.2 Sequential Pattern Matching	32
3.2 <i>ChronicleDB</i>	34
3.2.1 Lightweight Indexes	35
3.2.2 Replay-Based Continuous Query Evaluation	36
3.3 Index-Based Windowed Aggregation	37
3.3.1 Index-Based Processing	37

3.3.2	Cost Estimation	38
3.3.3	Arbitrary Slide Sizes and Count Windows	40
3.4	Index-Based Pattern Matching	41
3.4.1	Preliminaries	43
3.4.2	Replay Interval Computation	44
3.4.3	Index Selection	47
3.4.4	Extensions	55
3.5	Experimental Evaluation	58
3.5.1	System Setup	58
3.5.2	Windowed Aggregation	58
3.5.3	Sequential Pattern Matching	60
3.6	Summary	70
4	iGPU-Accelerated Online Processing	72
4.1	Related Work	73
4.2	Preliminaries	74
4.2.1	Memory Management	76
4.2.2	Signals	76
4.2.3	Data Exchange	77
4.3	Implementation	78
4.3.1	Event Queues	78
4.3.2	Filtering	79
4.3.3	Windowed Aggregation	82
4.3.4	Windowed Joins	83
4.3.5	Sequential Pattern Matching	84
4.4	Experimental Evaluation	88
4.4.1	Setup	89
4.4.2	Persistent Kernels	90
4.4.3	Operator Evaluation.	90
4.5	Summary	97
5	Pattern Matching on Temporal Intervals	98
5.1	State-of-the-Art	100
5.1.1	Straw Man’s Approach:	100
5.2	Related Work	101
5.3	Query Language	103
5.3.1	Syntax	103
5.3.2	Expressiveness	106
5.4	Algebra	107
5.4.1	Derivation	107
5.4.2	Pattern Matching	109

5.5	Algorithms & Implementation	111
5.5.1	Deriving Situations	111
5.5.2	Matching the Pattern	112
5.5.3	Low-Latency Matching	116
5.5.4	Computing the Evaluation Order	121
5.6	Parallel <i>TPStream</i>	123
5.6.1	Integration with Application Context	124
5.6.2	Partitioned Data	125
5.6.3	Unpartitioned Data	126
5.6.4	Auto-Tuning	131
5.6.5	Distributed <i>TPStream</i>	135
5.7	Experimental Evaluation	136
5.7.1	Setup	137
5.7.2	Processing Time	138
5.7.3	Low Latency	141
5.7.4	Plan Quality & Adaption	143
5.7.5	Parallel Approaches	145
5.8	Summary	152
6	Summary, Conclusion and Outlook	153
6.1	Outlook	154
	Appendices	156
	References	157
	List of Figures	176
	List of Tables	179
	List of Algorithms	180
	List of Abbreviations	182

Introduction

Due to a growing number of data producers and their ever-increasing data volume, the ability to ingest, analyze and store potentially never-ending streams of data has become a mission-critical task in today's data processing landscape. A widespread form of data streams are event streams. Conceptionally, an event is a notification that something happened in the real world at a specific point in time. When ordered by time, a sequence of these notifications forms an event stream. For example, a temperature sensor naturally generates an event stream by periodically measuring the temperature and reporting it with the measurement time. Technically, an event is composed of structured information (e.g., the temperature value) and a timestamp.

Many application domains dealing with massive event streams, like infrastructure monitoring, air traffic monitoring, health care, and financial applications, apply event processing technology¹. The following examples highlight some of the manifold use cases for this technology.

Infrastructure Monitoring. Large technical infrastructures employ a wide variety of sensors to monitor the proper functioning of servers, cooling systems, power supply, or computer networks [Ept+99]. Event processing technology allows consuming reports from heterogeneous sensors, expressing complex rules that involve readings from multiple sensors, and triggering actions based on rule evaluation results. For instance, the infrastructure monitoring system at the European Organization for Nuclear Research (CERN) continuously evaluates more than 1,300 rules over readings from more than 91 thousand sensors [Brä15].

Airspace Monitoring. Airborne flights produce massive event streams by continuously reporting status information (e.g., position, velocity, altitude). Moreover, air traffic volume is steadily increasing [Org18]. In order to manage the increasingly crowded air space and keep the workload of air traffic controllers at

¹In this thesis, we use the term event stream processing (or simply event processing) to refer to any tool, algorithm, or system that consumes and processes event streams.

an acceptable level, an important goal in airspace monitoring is to automate as many management tasks as possible. Event processing is ideally suited for this scenario, as it can analyze the massive event streams produced by airborne flights and notify operators of critical situations on time. For instance [AS19] uses event processing technology to monitor flight route conformance of aircraft and report anomalies, such as an aircraft leaving a planned route.

Health Care. Modern intensive care units monitor a patient’s current state via various sensors (e.g., heart rate, blood pressure). Event processing technology allows to monitor sensor streams from multiple patients continuously and immediately raise the alarm if a patient’s condition worsens. Due to expressive operators, such alarms are not restricted to simple threshold conditions like “blood pressure above 140”, but also allow, for instance, to detect complex patterns in electrocardiograms (ECGs) [HAB17].

Algorithmic Trading. Stock price patterns are widely used to analyze a specific stock and make a decision whether to buy or sell it [Kha02]. Event processing systems can detect those patterns in high-volume stock-ticker streams close to real-time, thus enabling quick reaction without the need for human intervention. For instance, Poppe et al. use event processing to count the number of downtrends per sector and sell stocks if this number exceeds a given threshold [Pop+19].

1.1 Continuous Queries

The above examples generate higher-level knowledge from raw event streams and translate those insights into actionable tasks. For this purpose, event stream processing systems employ the concept of continuous queries (CQs) [ABW06]. In contrast to queries in traditional database management systems (DBMSs) or data warehouses, which process and return finite data sets, CQs are designed to process potentially unbounded event streams. They consume one or more input streams and produce a single result stream. For instance, a filter query consumes a stream event by event and evaluates the filter condition on each of them. If an event fulfills the condition, the query forwards it to the output stream. Otherwise, it discards the event.

Stateless operations (e.g., the described filter) are a natural fit for unbounded streams because they consider a single event at a time and do not require auxiliary data to process this event. However, operations that require additional state information (stateful operations) suffer from two major problems when computed

over unbounded streams: They are either technically unfeasible or do not produce meaningful results. An example for the first issue is a join of two unbounded streams according to the semantics used in traditional DBMS. Here, the whole history of both streams needs to be kept, which would require infinite storage capacity. For the second issue, let us consider an aggregation query. Even with incremental aggregate computation that does not keep any history, there will never be a final answer to the query due to continuously arriving events. Thus no result is produced at all. To alleviate those problems, CQs limit the scope of stateful operations via so-called windows [KS09; Aff+17]. Windows capture finite fractions of the unbounded stream (e.g., the past five minutes) over which stateful computations are carried out. Besides continuous counterparts of well-known relational operations, like filter, projection, aggregation, and join, one of the key operations in event processing is sequential pattern matching [WDR06; DIG07; MM09; Bre+07]. It enables users to detect complex situations of interest that cannot be inferred by looking at a single event or an aggregation of events. An example of such a query is the detection of a gradually increasing temperature. Technically, this corresponds to a contiguous sequence of temperature readings where every reported value exceeds the value of the previous measurement. Systems featuring sequential pattern matching are commonly called complex event processing (CEP) systems [MC11]. This thesis uses the terms event processing and complex event processing interchangeably and refers to concrete operations (filter, join, sequential pattern matching) if required.

1.2 On- and Offline Processing

CQs can be processed in two ways: on- and offline. Online processing refers to the processing of incoming streams on the fly as new events arrive. The goal of online processing is to react to interesting situations in a timely manner [LWK14]. For instance, if two aircraft are approaching each other, the pilots should be informed as soon as possible to change their course early and avoid potential collisions. Thus, the primary performance metric of online processing is the processing latency of CQs, which is the elapsed time between the arrival of an event and the output of the corresponding CQ. In order to minimize the processing latency, online stream processing engines (SPEs) carry out computations solely in main-memory and process streams event by event (tuple-at-a-time) or in small batches (micro-batches).

In contrast, offline processing refers to the evaluation of CQs on persisted event streams, i.e., streams stored on non-volatile storage. Of course, offline process-

ing is not suitable for monitoring the current situation. However, in combination with online processing, it offers exciting opportunities such as parameter tuning for new online CQs on historical data, post-mortem stream analysis, or the re-evaluation of CQs with late-arriving events that were not considered by the corresponding online query. As can be seen from these examples, offline processing deals with finite fractions (e.g., the last year) of event streams interactively in an exploratory fashion. Especially in such interactive scenarios, quick response times are a major requirement. This requirement implies that the considered fraction of the stream as a whole should be processed as quickly as possible for offline processing. Thus, contrary to online processing, the primary performance metric for offline processing is throughput, which is the number of processed events per time unit.

1.3 Motivation & Contributions

Today, most on- and offline processing of continuous queries is carried out by scale-out general-purpose SPEs like Apache Flink [Car+15] or Spark Streaming [Zah+16]. Those systems process large volume event streams on clusters of commodity machines (nodes). They represent CQs as directed acyclic graphs (DAGs), also known as dataflow graphs. Such a graph consists of three types of nodes (sources, operators, sinks) and directed edges representing the flow of the events. Sources continuously produce events and submit them to the engine (e.g., a temperature sensor). Operators consume one or more input streams and generate a single result stream (e.g., joins, aggregation). Finally, sinks consume input streams without producing a result stream (e.g., a dashboard application visualizing the query results). Figure 1.1 (a) shows a simple dataflow graph consisting of two data sources, four operators, and a sink. Events submitted by Src_1 are processed sequentially by the unary operators Op_1 and Op_2 . Similarly, events of Src_2 are processed by Op_3 . Then, the binary operator Op_4 combines the result streams of Op_2 and Op_3 and forwards its results to the sink (Snk_1).

Scale-out SPEs use two parallelization techniques to distribute the computations across the cluster. First, every operator in the dataflow graph represents an independent computation step, which allows deploying adjacent operators to different cluster nodes (pipeline parallelism). Moreover, incoming streams get partitioned according to a predefined partitioning scheme (e.g., round-robin or based on a user-defined attribute). This way, multiple instances of the same operator are deployed to different cluster nodes, such that every instance processes only a fraction of the entire stream (data parallelism). Figure 1.1 (b) shows an example deployment for

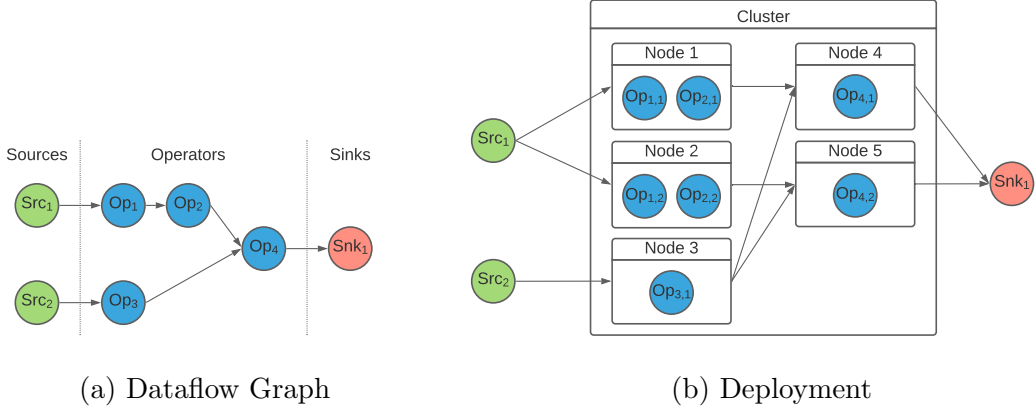


Figure 1.1: A dataflow graph consisting of two sources, four operators and one sink (a), and a possible deployment of this graph in a cluster of 5 nodes (b).

the dataflow graph of Figure 1.1 (a). Events arriving from Src_1 are partitioned and distributed to Nodes 1 and 2. Each of those nodes runs instances of Op_1 and Op_2 and processes the received partition independently. In contrast, Src_2 remains unpartitioned, and a single instance of Op_3 on Node 3 processes all its events. Op_4 runs on nodes 4 and 5. By reusing the partitioning of Src_1 , it is sufficient to forward the results of Node 1 only to Node 4 and those of Node 2 to Node 5. Depending on the requirements of Op_4 , Op_3 either partitions its results or completely sends them to both nodes.

Assuming a partition-friendly workload, modern SPEs scale almost linearly with the number of cluster nodes. However, as shown by a recent article [Zeu+19], those engines fail to utilize the available hardware resources of single cluster nodes fully. In other words, increasing the resource utilization of the cluster nodes would dramatically reduce the costs for running an SPE. Moreover, if a query or an operator prohibits partitioning the input stream, the system performance suffers because a single underutilized node carries out all computations. An example for such a query is detecting similar behaving stocks (i.e., stocks that face similar trends). If we partition the input stream (e.g., by company) and process the resulting partitions independently, two similar stocks might remain undetected since they belong to different partitions (which are processed by different nodes). Another drawback of those engines is offline processing. They load the entire stream from non-volatile secondary storage and replay it to the SPE for processing. While this naturally ensures unified query semantics for on- and offline processing, replay-based computation often results in high execution costs due to reading lots of data irrelevant to the query. For instance, consider a simple query that filters temperature sensor readings for values above $60^\circ C$. Assuming that 0.1% of a 1 TB

sensor stream fulfills this condition, only 1 *GB* of data is relevant to the query. Hence, most of the processing time for this query is spent on expensive secondary storage access reading irrelevant data.

This thesis tackles the problems sketched above from three different sides. First, we use indexing techniques similar to DBMS to minimize secondary storage access when processing offline CQs on persisted streams. Second, we use integrated GPUs (iGPUs) to maximize resource utilization of a single node. Third, we tackle single-node resource utilization by introducing *TPStream*, a new operator for pattern matching over temporal intervals. In the following, we present an outline and summarize the contributions for each of the three aspects.

1.3.1 Index-Supported Offline Processing

In the past few years, a considerable amount of research has been dedicated to the optimization of operators in online SPEs. Those optimizations range from incremental aggregate computation [Tan+15; THS17; SCL18] over lazy evaluation techniques for sequential pattern matching queries [ZDI14; KSS15] to multi-query optimization approaches [Run+15; RLR16; KS19]. Since those optimizations target an online scenario, they all assume that the relevant data resides in main memory, and thus aim at minimizing the cost for operator evaluation in terms of CPU cycles and main-memory consumption. However, as we will show in Chapter 3, when considering replay-based offline processing, the costs for reading an entire stream from secondary storage (i.e., HDD or SSD) quickly dominate the overall processing costs. While specialized systems for storing streams to and replaying them from secondary storage exist [SS17], techniques for the efficient evaluation of CQs over persisted streams have not been sufficiently addressed.

This thesis presents two novel approaches for answering windowed aggregation and sequential pattern matching queries on persisted event streams. In both cases, the main idea is to utilize well-understood indexing techniques to reduce the total amount of data to read from secondary storage. Moreover, since an over-utilization of indexes quickly leads to an overall worse runtime than a simple replay, we also develop cost models deciding if and which indexes to use for the query at hand.

1.3.2 iGPU-Accelerated Online Processing

General-purpose graphics processing units (GPUs) offer massive computational power for data-parallel tasks. In contrast to central processing units (CPUs)

which process data sequentially (i.e., one data item after another), GPUs apply the same operation to hundreds or thousands of homogeneous data items in parallel. This kind of data-parallel processing is called single instruction multiple thread (SIMT). At first glance, the SIMT model seems to be a natural fit for answering long-running event queries on high-volume streams since an event stream is an unbounded sequence of homogeneous data items. However, usually GPUs come as PCI express (PCIe) cards with dedicated memory, and thus data needs to be shipped to the GPUs before processing and sent back afterward. This data shipping incurs considerable latency, which conflicts with the low-latency requirements of online stream processing. Traditionally, this challenge is tackled via software-based scheduling such as pipelined execution [Kol+16]. However, the assumptions about transfer do not hold for iGPUs, which share the main memory with the CPU. Hence, they offer the possibility for low-latency GPU-powered online event processing.

In this thesis, we develop a prototypical iGPU-enabled event processing system and provide implementations of all major event processing operators (filtering, windowed aggregation, windowed join, and sequential pattern matching). Besides operator implementations optimized for (multi-threaded) CPU- and traditional dedicated GPU (dGPU)-based execution, we also provide iGPU-optimized versions that utilize the shared main memory to realize fine-grained task scheduling between the CPU and the iGPU. A thorough evaluation of the prototype reveals that the cooperative iGPU-based processing approach outperforms both other variants up to a factor of 5 in terms of throughput while keeping the latency penalty low.

1.3.3 Pattern Matching on Temporal Intervals

Over the past decade *sequential pattern matching* has received lots of attention in academia [DIG07; MM09; Bre+07] and also found its way into several commercial products like Apache Flink [Car+15] or Esper [esp20]. However, the sequential nature of pattern matching has two major deficiencies. First, it is hardly possible to express complex temporal relationships between situations lasting for periods (e.g., rainfall *while* temperature is below 0°C). Because events are equipped with a single timestamp only, the expressible temporal relations are limited to before/after/at the same time. Second, a sequential pattern maps to a subsequence of the input stream that may start at any event. Hence, the source stream cannot easily be sliced into disjoint, independent temporal partitions since a match will likely span multiple partitions. In turn, this makes efficient parallelization of sequential pattern matching a hard problem.

We present *TPStream*, a novel event processing operator for complex temporal pattern matching on event streams. *TPStream* first summarizes incoming events to situations lasting for periods before it matches temporal patterns. With situations, temporal patterns can easily be defined based on Allen’s interval algebra [All83]. In contrast to existing interval-based approaches, *TPStream* is able to detect matches without exact knowledge about the end of all involved intervals. As we will show, this feature greatly reduces the processing latency. Furthermore, we develop parallelization strategies and show that *TPStream* efficiently utilizes multiple threads on a single machine and scales to multiple machines in a cluster, even with unpartitioned input streams.

1.4 Publications

The following papers were published in the course of this thesis:

- Michael Körber, Nikolaus Glombiewski, Bernhard Seeger:
Index-Accelerated Pattern Matching in Event Stores.
SIGMOD '21: 1023-1036.
- Nikolaus Glombiewski, Philipp Götze, Michael Körber, Andreas Morgen, Bernhard Seeger:
Designing an Event Store for a Modern Three-layer Storage Hierarchy.
Datenbank-Spektrum 20(3): 211-222. (2020)
- Marc Seidemann, Nikolaus Glombiewski, Michael Körber, Bernhard Seeger:
ChronicleDB: A High-Performance Event Store.
ACM TODS 44(4): 13:1-13:45 (2019).
- Michael Körber, Nikolaus Glombiewski, Andreas Morgen, Bernhard Seeger:
Tpstream: low-latency and high-throughput temporal pattern matching on event streams.
Distributed and Parallel Databases: 1-52 (2019).
- Michael Körber, Jakob Eckstein, Nikolaus Glombiewski, Bernhard Seeger:
Event Stream Processing on Heterogeneous System Architecture.
DaMoN 2019: 3:1-3:10.
- Christian Beilschmidt, Johannes Drönner, Nikolaus Glombiewski, Christian Heigle, Jana Holznigenkemper, Anna Isenberg, Michael Körber, Michael Mattig, Andreas Morgen, Bernhard Seeger:
Pretty Fly for a VAT GUI: Visualizing Event Patterns for Flight Data.
DEBS 2019: 224-227.
- Christian Beilschmidt, Johannes Drönner, Nikolaus Glombiewski, Michael Körber, Michael Mattig, Andreas Morgen, Bernhard Seeger:
VAT to the Future: Extrapolating Visual Complex Event Processing.
OpenSky Workshop 2019: 25-36.
- Pablo Graubner, Christoph Thelen, Michael Körber, Artur Sterz, Guido Salvaneschi, Mira Mezini, Bernhard Seeger, Bernd Freisleben:
Multimodal Complex Event Processing on Mobile Devices.
DEBS 2018: 112-123.

- Michael Körber, Nikolaus Glombiewski, Bernhard Seeger:
TPStream: Low-Latency Temporal Pattern Matching on Event Streams.
EDBT 2018: 313-324.

1.5 Thesis Structure

The remainder of this thesis is structured as follows.

Chapter 2 introduces fundamental concepts used throughout this thesis. These concepts include the definition of events, event streams, and windows and an introduction into the semantics of all major event processing operators.

Chapter 3 presents our index-supported approaches for offline processing of continuous queries. This chapter also features a description of *ChronicleDB* [SS17], the event store system forming the basis for our approaches. Further, we discuss relevant related work and showcase the performance improvement achieved by our approaches in extensive experiments.

This chapter contains parts of [Sei+19] and [KGS21].

Chapter 4 presents our iGPU-enabled event-processing framework. Moreover, this chapter also includes an introduction to GPU computing, discusses relevant related research, and highlights the performance improvements achieved by iGPU-based online processing in various experiments.

This chapter is an enhanced version of [Kör+19a].

Chapter 5 presents *TPStream*, our novel event processing operator for low-latency pattern matching on temporal intervals. We formally define its semantics and detail important implementation aspects. Furthermore, we present parallelization strategies for modern multi-core CPUs as well as distributed cluster setups. We discuss related research and show that *TPStream* exhibits excellent online processing performance in an extensive experimental evaluation.

This chapter contains parts of [KGS18] and [Kör+19b].

Chapter 6 concludes this thesis. We summarize the main findings and highlight directions for future work.

2

Preliminaries

This section introduces the core concepts and notations used in the course of this thesis. We start with an introduction to data streams and event streams. Then, we define the semantics of the core operations of event stream processing (ESP) with an emphasis on sequential pattern matching. Finally, we introduce the concept of situations that are the foundation for pattern matching on temporal intervals.

Note that we introduce concepts relevant to a specific part of this thesis (e.g., event stores, GPU computing) in the corresponding chapters.

2.1 Data Streams

This thesis works with two particular kinds of streaming data: event streams and situation streams. Both kinds share commonalities, such as an ordering relation, which are captured by the notion of a data stream.

Definition 2.1 (Data Stream). A data stream D is a potentially unbounded sequence of data items $\langle d_1, d_2, \dots \rangle$ totally ordered by a relation $<_D$. $d_i \in D$ refers to the i -th data item in the stream according to that order, and all data items are from the same domain \mathcal{D} (i.e., $d_i \in \mathcal{D}, i = 1, \dots$). $\langle \rangle$ refers to an empty data stream and is mainly used to specify the case of *no output* in upcoming definitions. If a data stream is bounded, we use N to denote the number of contained items (i.e., $D = \langle d_1, d_2, \dots, d_N \rangle$).

In order to refer to multiple data streams, we will utilize the notation D^1, D^2, D^3, \dots with $D^i = \langle d_1^i, d_2^i, \dots \rangle$, i.e., a superscript labels separate streams, while a subscript refers to the order within a stream. For the sake of simplicity and readability, we will generally assume that each item in a data stream is unique (i.e.,

$\forall d_j^i, d_k^i \in D^i : j \neq k \implies d_j^i \neq d_k^i$) and refer to previous work on the matter of handling potentially equal elements (see [Bre+07]).

We often need to restrict unbounded streams to a finite fraction for stateful operations. Usually, such a finite fraction refers to a contiguous subsequence, which is defined as follows.

Definition 2.2 (Contiguous Subsequence). Based on a data stream D , $D_{[i,j]} = \langle d_i, \dots, d_j \rangle$ with $i < j$ refers to a contiguous subsequence containing every data item as it pertains to $<_D$.

Finally, two data streams can be unified if they use the same order relation and their data items share the same domain.

Definition 2.3 (Union). The union \uplus of two data streams D^1 and D^2 , both from the same domain \mathcal{D} and totally ordered with $<_D$, results in a data stream D' with the same order $<_D$.

$$\uplus(D^1, D^2) := D' = \langle d'_1, d'_2, \dots \rangle$$

such that D' contains each element from D^1 and D^2 . Analogous to set theory, the union of n data streams D^1, \dots, D^n is abbreviated with the notation $\biguplus_{i=1}^n D^i$.

Note that the data stream concept is not limited to structured data, such as an ordered stream of relational tuples. For instance, a video stream is a sequence of images ordered by frame numbers [PBH17]. Also large XML-documents [JFB05] or sequences of unstructured text such as tweets [HOS18] can be mapped to this definition.

2.2 Event Streams

As sketched in the introduction, events are notifications about observations from the real world. An event stream is a temporally ordered sequence of events and thus a special form of a data stream. In the following, we define events, event streams, and related concepts.

Definition 2.4 (Event). An event $e = (a_1, \dots, a_d, t)$ is a $(d+1)$ -dimensional tuple consisting of d attributes (a_1, \dots, a_d) and a timestamp t . We denote the domain (e.g., integer, string) of the i -th attribute with \mathcal{A}^i ; t is from a discrete,

totally ordered time domain \mathcal{T} . Without the loss of generality, we assume $\mathcal{T} = \mathbb{N}$. We use the dot notation to access an event's attributes ($e.a_i$) and timestamp ($e.t$). Moreover, we summarize the d attributes of an event as payload ($e.p$) over the d -dimensional domain $\mathcal{P} = A^1 \times \dots \times \mathcal{A}^d$, if appropriate.

Definition 2.5 (Event Stream). An event stream E is a potentially unbounded sequence of events $E = \langle e_1, e_2, \dots \rangle$. All events of an event stream are from the same domain and ordered according to their timestamps (i.e., it holds $e_i.t \leq e_j.t$ for $i < j$). Note that this definition allows duplicate timestamps (i.e., $e_i.t = e_{i+1}.t$). If a stream contains such duplicates, ties are broken via an additional attribute (e.g., a sequence number) in order to comply to the definition of a data stream (Definition 2.1).

Similar to data streams, we refer to a contiguous subsequence of event streams via $E_{[i,j]}$. However, in some cases, it is required to access a specific temporal region of a stream (e.g., all events of December 2020).

Definition 2.6 (Timestamp Mapping). The function $\tau_E : \mathcal{T} \rightarrow \mathbb{N}$ maps a timestamp $t \in \mathcal{T}$ to the index of first event in E with a timestamp greater than or equal to t :

$$\tau_E(t) := \arg \min_{i \in \mathbb{N}} \begin{cases} e_i.t, & \text{if } e_i.t \geq t \\ \infty, & \text{otherwise} \end{cases} \quad (2.1)$$

Based on the function τ , we can refer to all events of a stream whose timestamps are in the half-open interval $[t_1, t_2)$ as $E_{[\tau_E(t_1), \tau_E(t_2)-1]}$. When obvious from the context, we omit the subscript for the function τ .

2.3 Event Stream Processing

Event processing, as considered in this thesis, has roots in two different research areas: data stream processing and complex event processing (CEP). Data stream processing focuses on efficiently processing streaming relational data (i.e., tuples of structured data). In 2003, STREAM [Ara+03] was one of the first systems in this area, featuring window semantics and the CQL declarative query language [ABW06]. STREAM slices incoming streams into temporary relations and performs relational operations upon them. Afterward, it transforms the result relations back into data streams. Krämer et al. [KS09] pursue a similar approach.

They introduce the concept of snapshot reducibility to define the execution semantics of streaming operators based on their relational algebra counterparts. Parallel to STREAM, the Aurora/Borealis [Aba+03] system was developed. It is the first approach that models streaming queries as a dataflow graph, which was adopted by many modern SPEs like Apache Flink [Car+15] or Spark Streaming [Zah+16]. Beyond the basic semantics as presented above, research also explored many aspects to make stream processing practical for complex real world use cases, such as the handling of duplicate data items [Bre+07], disordered streams [Aki+15; Ji+16], or approximate query processing [CG08].

While data stream processing focuses on the efficient evaluation of relational operations over streams of structured data, CEP targets the extraction of higher-level knowledge from so-called primitive events. In other words, CEP focuses on the detection of patterns in streams of simple events, like temperature sensor readings. With active databases [MD89], the database community took its first steps in this direction in the late 1980's. They enabled traditional DBMS to autonomously trigger actions based on predefined events (e.g., the insertion of a record). Later, the active database community presented several complex event detection models (see [ZU99] for an excellent survey). Those models cover all features available in modern online systems like sequence detection, negation, or alternatives. One of the first systems featuring online CEP was Rapide [Luc98] in 1998. Rapide is a system to model distributed architectures with an embedded CEP engine. In the following years, the community focused on the efficient evaluation of online CEP queries. This includes automaton based approaches like SASE+ [DIG07] and Cayuga [Bre+07], tree-based approaches like ZStream [MM09], and also graph-based approaches like GraphCEP [May+16]. Aside from the bare processing performance, a great body of work on CEP deals with real-world data problems such as matching patterns over streams with imprecise timestamps [ZDI10] or disordered event streams [CGM10].

Modern SPEs like Apache Flink [Car+15] or Esper [esp20] feature aspects of both data stream processing and CEP. They provide powerful, highly optimized operator implementations for relational processing, as well as pattern matching operators for answering CEP-style queries. Following this approach, we represent a CQ as a dataflow graph composed of event processing operators. The operators we consider are filtering, windowed aggregation, windowed join, and sequential pattern matching. In the remainder of this section, we first introduce the concept of windows before defining each operator's semantics.

2.3.1 Windows

Windows are an integral part of every system that deals with potentially unbounded data streams. Besides time and count windows, many different types of windows exist, including session windows [Aki+15] that capture all events until a period of inactivity (e.g., a user click-stream), or data-driven approaches [Gro+16] that, for instance, capture all events with an attribute value above a given threshold. However, time and count windows are the most common forms of windows. For this reason, we only consider those two window types in this thesis.

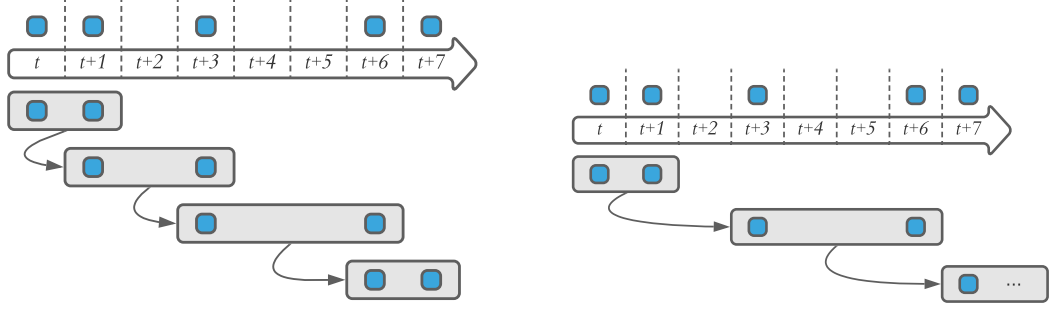
Count windows always hold a fixed number of events (e.g., the 100 most recent events). They are defined via the two parameters $size, slide \in \mathbb{N}$, with $0 < slide \leq size$. The $size$ parameter determines the number of events kept in the window, while the parameter $slide$ determines how the window changes with new arriving events. If $slide = 1$, the window is called a *sliding* count window, otherwise it is a *tumbling* count window.

Definition 2.7 (Count Window). For an event stream E , a count windows $W_{size,slide}^C$ applied to E is a sequence of tuples $w_i = (seq, t)$, $i \in \mathbb{N}$. For each such tuple, seq is a contiguous subsequence of E of length $size$, and t is the maximum timestamp of the events in seq .

$$W_{size,slide}^C(E) = \langle \begin{aligned} &w_1 = (E_{[1, size]}, e_{size} \cdot t), \\ &w_2 = (E_{[1+slide, size+slide]}, e_{size+slide} \cdot t), \\ &w_3 = (E_{[1+2 \cdot slide, size+(2 \cdot slide)]}, e_{size+(2 \cdot slide)} \cdot t), \\ &\dots \end{aligned} \rangle$$

Figure 2.1, shows two variants of a count window with $size = 2$. Both windows hold exactly two events, independent of their timestamps. The sliding variant (Figure 2.1 (a)) is updated with every incoming event. The eldest element leaves the window while the new event enters. In contrast, Figure 2.1 (b) shows a tumbling count window with $slide = 2$. Here, an update occurs after exactly two events and those two events replace the entire window content.

In contrast to count windows, time windows capture events of a predefined period (e.g., 5 minutes). They are defined via the two parameters $size$ and $slide$. $Size$ specifies the storage duration of events, while $slide$ defines the amount of time the window slides forward in each step. Both parameters are specified using a predefined time unit (e.g., seconds, milliseconds).



(a) Sliding count window ($slide = 1$) (b) Tumbling count window ($slide = 2$)

Figure 2.1: Example for the content of a sliding (a) and tumbling (b) count window of $size = 2$.

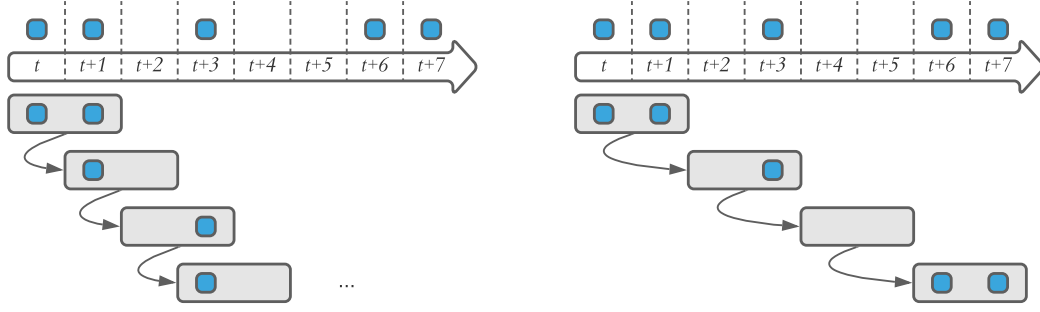
Definition 2.8 (Time Window). A time window $W_{size,slide}^T$ applied to an event stream E is a sequence of tuples $w_i = (seq, t)$, $i \in \mathbb{N}$. For each such tuple, seq is a contiguous subsequence of E , and t is the maximum timestamp covered by w_i .

$$W_{size,slide}^T(E) = \langle \begin{aligned} w_1 &= (E_{[1, \tau(e_1.t+size)-1]}, e_1.t + size - 1), \\ w_2 &= (E_{[\tau(e_1.t+slide), \tau(e_1.t+size+slide)-1]}, e_1.t + size + slide - 1), \\ w_3 &= (E_{[\tau(e_1.t+2 \cdot slide), \tau(e_1.t+size+2 \cdot slide)-1]}, e_1.t + size + 2 \cdot slide - 1), \\ &\dots \end{aligned} \rangle$$

Note that the number of events in a time window varies depending on the distribution of the events' timestamps. A time window may also be empty (i.e., $w.seq = \langle \rangle$). Figure 2.2 illustrates the behaviour of a time window with $size = 2$. The sliding variant (Figure 2.2 (a)) progresses by a single time unit in every step. In contrast, the tumbling variant (Figure 2.2 (b)) progresses by $slide = 2$ time units. As shown by the third window in the tumbling case, time windows may contain no events at all.

Independent of the window type, there is at most one window considered active at any point in time $t \in \mathcal{T}$.

Definition 2.9 (Active Window). Let E be an event stream, $t \in \mathcal{T}$ a timestamp, and W be a count or a time window (i.e., $W = W_{size,slide}^C$ or $W = W_{size,slide}^T$). Then, the active window $AW_{E,W}$ at time t is defined as follows.


 (a) Sliding time window ($slide = 1$)

 (b) Tumbling time window ($slide = 2$)

 Figure 2.2: Example for the content of a sliding (a) and tumbling (b) time window of $size = 2$.

$$AW_{E,W}(t) = \arg \max_{w \in W(E)} \begin{cases} w.t & , \text{ if } w.t \leq t \\ -\infty & , \text{ otherwise} \end{cases} \quad (2.2)$$

Active windows play an important role for operators with more than a single input stream, as we will show when introducing the windowed join later in this section.

In general, there exist two ways of managing windows. Either by coupling them tightly with a stateful operator [Li+05a; Kol+16] or by implementing them as dedicated operators, which encode window information directly into the events of the stream [KS09]. The tight coupling of windows with operators has two significant advantages. First, it reduces the memory footprint because it does not add window information to every event. Second, it allows optimizing the window implementation towards efficient operator execution (e.g., by assembling windows from disjoint substreams to avoid redundant computations for overlapping windows [Li+05a]). Thus, we opted for attaching windows directly to stateful operators.

2.3.2 Filtering

Filters remove all events from a stream not satisfying a given predicate. They are typically used to remove uninteresting events and reduce the input to more complex operators. For example, in a temperature monitoring use case we may consider

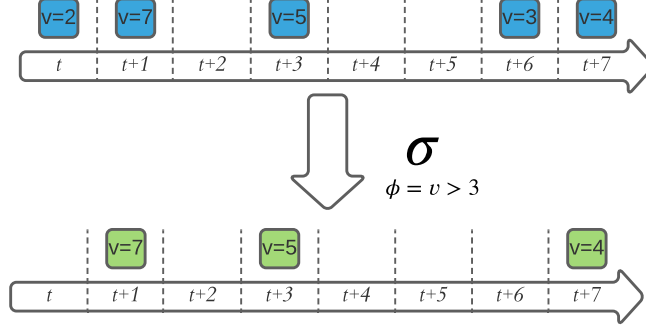


Figure 2.3: Example of a filter with $\phi = v > 3$ applied to an event stream.

room temperatures between $18^\circ C$ and $25^\circ C$ normal. Hence, measurements in this range are of no interest and thus discarded.

Definition 2.10 (Filter). Let E be an event stream with domain \mathcal{P} and $\phi : \mathcal{P} \rightarrow \{true, false\}$ a predicate. Then, a filter σ_ϕ applied to E generates a new event stream consisting of all events of E that fulfill ϕ .

$$\sigma_\phi(E) = \biguplus_{e \in E} \begin{cases} \langle e \rangle & , \text{ if } \phi(e.p) \\ \langle \rangle & , \text{ otherwise} \end{cases} \quad (2.3)$$

Figure 2.3 shows an example of a filter operation that removes all events from a stream not satisfying the user-defined predicate $v \leq 3$. In this case, the filter drops the events at timestamps t and $t+6$, but send the results at $t+1$, $t+3$ and $t+7$ downstream.

Since even a simple event-at-a-time filter implementation is very efficient, filters have received little attention from the research community. Typically, events are batched to apply the filter condition to multiple events in parallel via single instruction multiple data (SIMD) instructions on modern CPUs [Now+18] or on GPUs [RBP15]. However, thousands of filters running in parallel quickly become a bottleneck since all filter conditions must be evaluated for every element of a stream. This bottleneck is alleviated by employing so-called filter indexes [SJ11; SJ13; ZCT14; Hoß15]. Instead of indexing the data, like it is done with B⁺-trees [BM70] or R-trees [Gut84], they index filter conditions (i.e., boolean expressions). When inspecting a new data item, the index retrieves only those filters whose condition may evaluate to true and ignores the remaining ones. Thus, the index reduces the number of filter queries to evaluate per data item

and thus improves processing performance for workloads with many active filter queries.

2.3.3 Windowed Aggregation

Most stream analysis at some point involves windowed aggregation. Windowed aggregation computes a set of aggregations over finite fractions of the input stream and generates a result stream consisting of the aggregated values. An example of this is the calculation of the average room temperature on an hourly basis.

Definition 2.11 (Windowed Aggregation). Let W be a count or time window and γ a set of aggregation functions (e.g., sum, min, max). Then, the windowed aggregation $\alpha_{W,\gamma}$ of an event stream E is defined as follows.

$$\alpha_{W,\gamma}(E) = \biguplus_{w \in W(E)} \langle (\gamma(w.seq), w.t) \rangle \quad (2.4)$$

That is, every window $w_i \in W(E)$ generates a single result event by applying the aggregation functions γ to the events of w_i . The result events carry the timestamp of the respective window.

Figure 2.4 shows a windowed aggregation using a sliding time window with *size* = 2. The aggregation computes the sum over attribute v of the input stream (shown in blue in the upper part of the figure). Since we use a sliding time window, the window’s content changes with every time tick (w_1, \dots, w_7). Thus, the result stream (green events at the bottom) contains an aggregated event for every time instant.

Windowed aggregation has been extensively researched and optimized. The general idea of most approaches is to compute aggregations incrementally in two phases. The first phase computes partial aggregates for a certain number of events, and the second phase uses them to assemble the result for the entire window. For instance, Li et al. [Li+05b] compute partial aggregates for continuous fixed size subsequences of the data stream, while Carbone et al. [Car+16] starts a new partial aggregate at the beginning of new windows. Likewise, various approaches for the second phase exist using different data structures to manage and merge partial aggregates (for example binary trees [Tan+15] or queues [THS17; SCL18]). Besides clever data structures, modern hardware is also exploited for incremental aggregation. For instance, the SABER system [Kol+16], schedules

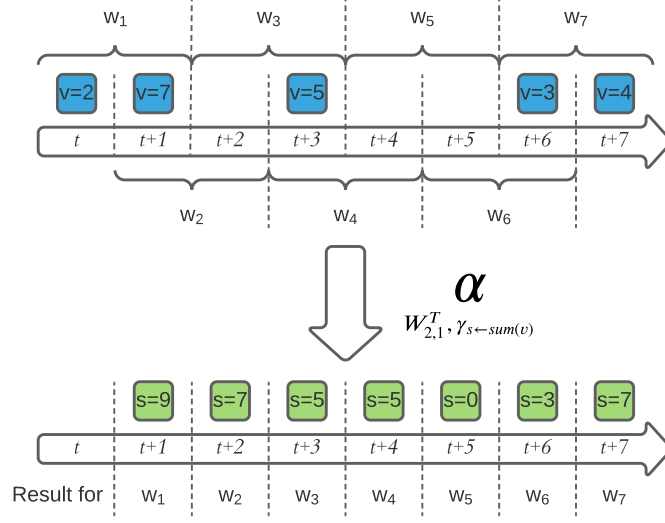


Figure 2.4: Example of a windowed sum-aggregation with a sliding time window of $size = 2$.

the computation of partial aggregates on the GPU, while LightSaber [The+20] uses a specialized tree structure to exploit the parallelism of modern multi-core CPUs.

2.3.4 Windowed Join

A windowed join combines two event streams based on a join predicate. Windows on both streams limit the number of events to consider when searching for join partners. As a first step towards a windowed join of two event streams, we define the result of joining a single event with an event stream.

Definition 2.12 (Single Event Join). Let e^1 be an event and E^2 an event stream with domains \mathcal{P}^1 and \mathcal{P}^2 , respectively. Moreover let W be a window and $\phi : \mathcal{P}^1 \times \mathcal{P}^2 \rightarrow \{true, false\}$ a join predicate. The result of joining e^1 with E^2 is then:

$$\bowtie_{\phi, W}(e^1, E^2) = \biguplus_{e^2 \in AW_{E^2, W}(e^1.t)} \begin{cases} \langle (e^1.p, e^2.p, e^1.t) \rangle & , \text{ if } \phi(e^1.p, e^2.p) = true \\ \langle \rangle & , \text{ otherwise} \end{cases} \quad (2.5)$$

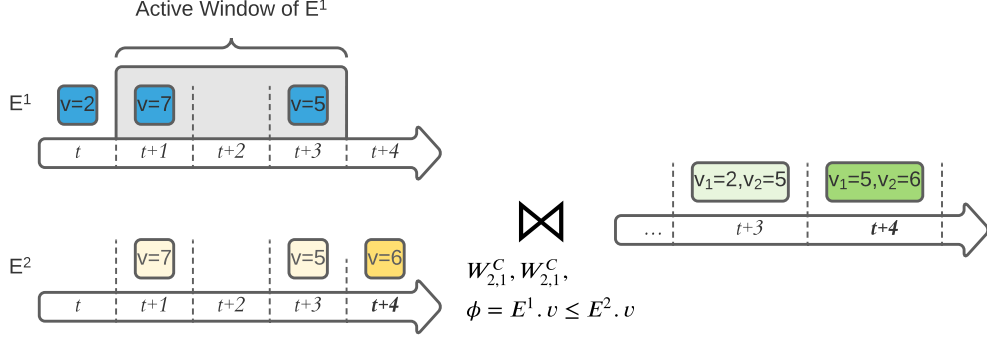


Figure 2.5: Example of a windowed join between stream E^1 and E^2 using two sliding count windows of $size = 2$ and $\phi = E^1.v \leq E^2.v$ as the join predicate.

That is, the event e^1 is combined with all events of E^2 's active window at time $e^1.t$. If such a combined event fulfills the join condition, we create a result event e with $e.t = e^1.t$ and the combined payload.

Based on the single event join, the windowed join of two event streams is defined as follows.

Definition 2.13 (Windowed Join). Let E^1 and E^2 be two event streams with domains \mathcal{P}^1 and \mathcal{P}^2 , respectively. Furthermore, let W^1 and W^2 be windows and $\phi : \mathcal{P}^1 \times \mathcal{P}^2 \rightarrow \{true, false\}$ a join predicate. Then, the windowed join \bowtie_{ϕ, W^1, W^2} of E^1 and E^2 is defined as:

$$\bowtie_{\phi, W^1, W^2} (E^1, E^2) = \bigcup \left\{ \bigcup_{e \in E^1} \bowtie_{\phi, W^2} (e, E^2), \bigcup_{e \in E^2} \bowtie_{\bar{\phi}, W^1} (e, E^1) \right\} \quad (2.6)$$

Note that the single event join between $e \in E^2$ and E^1 uses $\bar{\phi}(e_1, e_2) = \phi(e_2, e_1)$ as join predicate to ensure the correct argument order.

Figure 2.5 shows an example of a windowed join between the streams E^1 and E^2 using two sliding count windows of $size = 2$ and $\phi = E^1.v \leq E^2.v$ as the join predicate. We examine the event $(v = 6, t + 4)$ from E^2 . At $t + 4$, the active window of E^1 is composed of the two events $(v = 7, t + 1)$ and $(v = 5, t + 3)$. Since only the latter event fulfills the join condition, a single output event is created at $t + 4$.

Similar to join processing in traditional DBMSs, windowed joins over unbounded streams received lots of attention from the research community. In 2003 Kang et al. [KNV03] analyzed several join implementations, like nested loops, index nested loops, or hash joins, in the context of stream processing. Based on this analysis, they present a first cost model for selecting an appropriate algorithm using statistics like arrival rate and selectivity of the join predicate. Moreover, they summarize the main problems that arise when evaluating windowed joins in an online manner. These problems fall into two main categories: Plan generation (i.e., selection of the optimal join algorithm) and resource management (i.e., how to deal with scarce CPU and main memory resources). Since then, both categories have received considerable attention from the community. For instance, in resource-constrained scenarios, load shedding can help to reduce memory and CPU utilization by sacrificing accuracy of the join results [DGR03; SW04]. Other approaches swap data to non-volatile storage [UF00] or use punctuations [Din+04; DR04] to reduce main memory consumption. Plan generation was considered in various scenarios ranging from single-query n -way joins on single machines [GÖ03; VNB03] to multiple independent join queries in distributed setups [KRM19]. While single-query optimizers primarily focus on cost models to select proper algorithms and join orderings [GC08; Cam15], multi-query aware approaches usually try to select algorithms that maximize sharing opportunities among the considered queries [Wan+06; WR09]. Finally, modern hardware like field programmable gate arrays (FPGAs) [TM11], heterogeneous processor architectures [GBY09], and GPUs [Kol+16] were shown to be suitable accelerators for windowed join processing.

2.3.5 Sequential Pattern Matching

Sequential pattern matching detects subsequences of events that indicate a situation of interest. Therefore, it matches the stream to a regular expression comprised of user-defined predicates. These predicates provide a dynamic definition of the alphabet. An example of a pattern query is the detection of a faulty cooling system in a data center. If the room temperature gradually increases by at least $5^\circ C$, it is very likely that the cooling system is not working correctly. The pattern to detect such a gradual increase is formulated as regular expression AB^*C using the three symbols A , B , and C with the following conditions:

A : $true$

B : $temp > prev(temp)$

C : $temp \geq A.temp + 5$

The intuition behind this specification is as follows. A failure can occur at any time (A). Then, from this unknown point in time on, we face a continuous temperature increase (i.e., the currently measured temperature is above the temperature reported by the previous event, B). Finally, this continuous increase results in a temperature at least $5^\circ C$ above the temperature reported at the beginning of the sequence (C). Note that symbol C refers to a value previously bound by symbol A .

Definition 2.14 (Pattern Query). A pattern query $PQ^M = ((S_1, \dots, S_n), w, \text{map})$ over an event stream E consists of a sequence of symbols (S_1, \dots, S_n) , $n > 0$, a sliding time window of size $w > 0$, a mapping function map , and a matching strategy M .

There are two different types of symbols: basic symbols (BSs) and Kleene-star symbols (KSs). A BS essentially is a predicate on the event's payload. However, unlike filter predicates, which are restricted to the currently processed event, BSs may also refer to any prior event in the currently matched subsequence. For instance, in the temperature example, we compare the temperature of the current event with the previous measurement to detect increasing values. While a BS targets a single event, a KS S^* is a predicate defined on a subsequence of E with arbitrary length (including length 0). The map function creates a single result event from a matched subsequence. Finally, the matching strategy M restricts valid subsequences (e.g., by only allowing contiguous sequences or skipping irrelevant events).

In accordance with the literature [DIG07; KS18; Pop+19], we consider three matching strategies in this thesis: *skip-till-any*, *skip-till-next*, and *contiguous*. We first define the notion of a matching subsequence (match for short) for each of these strategies, before defining the output of the pattern matching operator.

Definition 2.15 (Match – *skip-till-any*). Let $PQ^M = ((S_1, \dots, S_n), w, \text{map})$ be a pattern query over an event stream E with $M = \text{skip-till-any}$. Then, a (not necessarily contiguous) subsequence $r = (r_1, \dots, r_p)$ of E of length p is a match for PQ^M if the following conditions hold.

- (1) $r_1.t < \dots < r_p.t$
- (2) $r_p.t - r_1.t \leq w$
- (3) If S_1 is a BS, then $S_1(r_1)$ holds and (r_2, \dots, r_p) is a match of $((S_2, \dots, S_n), w - (r_2.t - r_1.t))$.

- (4) If S_1 is a KS, then either $S_1(r_1)$ holds and (r_2, \dots, r_p) is a match of $((S_1, \dots, S_n), w - (r_2.t - r_1.t))$, or $S_1(r_1)$ does not hold and (r_1, \dots, r_p) is a match of $((S_2, \dots, S_n), w)$.

Condition (1) ensures the temporal order among the matched events. For the sake of simplicity and readability, we assume the input to pattern matching queries contains no duplicate timestamps¹. Condition (2) ensures that the match spans at most w time units (i.e., obeys the window constraint). Condition (3) addresses the case of a BS at the beginning, whereas (4) treats the case of a KS.

Skip-till-any is the most flexible matching strategy. A matching subsequence is not required to be contiguous in E . Moreover, events that satisfy a symbol's condition can be ignored in a match, which leads to multiple matches starting with the same event. The computational costs for detecting all subsequences with $M = \text{skip-till-any}$ semantics grow exponentially with the length of the pattern [ZDI14].

The *skip-till-next* strategy decreases this complexity. It does not allow to skip events that satisfy a symbol's condition. As a result, for any event e , there is at most one match starting with e .

Definition 2.16 (Match – *skip-till-next*). Let $PQ^M = ((S_1, \dots, S_n), w, \text{map})$ be a pattern query over an event stream E with $M = \text{skip-till-next}$. Then, a (not necessarily contiguous) subsequence $r = (r_1, \dots, r_p)$ of E of length p is a match for PQ^M if the following conditions hold.

- (1) r is a match according to *skip-till-any* semantics.
- (2) Between two adjacent events $r_i = e_j, r_{i+1} = e_k$ in r , there exists no event in $E_{[j+1, k-1]}$ that satisfies the predicate of the symbol corresponding to r_{i+1} .

The valid matches of *skip-till-next* are a subset of the matches of *skip-till-any*. Condition (2) prohibits skipping events that satisfy the predicate of the current symbol. In other words, the pattern is evaluated greedily by always adding the next possible event to the subsequence.

Finally, the *contiguous* strategy is even more restrictive since it requires a matching subsequence to be contiguous in E .

¹Typically, contemporary events are treated as alternatives [Hoß15]. While this generally increases the computational complexity, it does not affect the core ideas presented in this thesis.

Strategy	Symbol Trace / Matches
	$\langle a_1, b_1, a_2, c_1, b_2, c_2 \rangle$
<i>contiguous</i>	$\langle a_2, c_1 \rangle$
<i>skip-till-next</i>	$\langle a_2, c_1 \rangle, \langle a_1, b_1, c_1 \rangle$
<i>skip-till-any</i>	$\langle a_2, c_1 \rangle, \langle a_1, b_1, c_1 \rangle, \langle a_1, b_1, c_2 \rangle, \langle a_1, c_1 \rangle, \langle a_1, c_2 \rangle,$ $\langle a_1, b_1, b_2, c_2 \rangle, \langle a_1, b_2, c_2 \rangle, \langle a_1, c_2 \rangle, \langle a_2, b_2, c_2 \rangle$

Figure 2.6: Matching subsequences for each of the three matching strategies applied to an symbol trace.

Definition 2.17 (Match – *contiguous*). Let $PQ^M = ((S_1, \dots, S_n), w, \text{map})$ be a pattern query over an event stream E with $M = \text{contiguous}$. Then, a subsequence $r = (r_1, \dots, r_p)$ of E of length p is a match for PQ^M if the following conditions hold.

- (1) r is a match according to *skip-till-any* semantics.
- (2) r_1, \dots, r_p are contiguous in E (i.e., $r_1 = e_i \implies r_2 = e_{i+1} \wedge r_3 = e_{i+2} \wedge \dots \wedge r_p = e_{i+p-1}$).

We illustrate the behavior of the different matching strategies with an example. Consider the pattern AB^*C and the symbol trace² in Figure 2.6. The *contiguous* strategy detects only the match $\langle a_2, c_1 \rangle$ because those are the only contiguous symbols that match the pattern. Using *skip-till-next* results in an additional match because non-matching symbols are skipped. In our example, $\langle a_1, b_1, c_1 \rangle$ becomes a valid match by skipping a_2 between b_1 and c_1 . Finally, with *skip-till-any*, the number of matches increases to 9, since all subsequences matching the pattern are accepted (especially those that ignore relevant symbols).

After clarifying the semantics for each matching strategy, we can define the sequential pattern matching operator as follows.

Definition 2.18 (Sequential Pattern Matching). Let E be an event stream and PQ^M a pattern query. The sequential pattern matching operator ρ_{PM^M} detects all subsequences³ of E that are a match for PQ^M . Every match produces a single result event. The timestamp of a result is the maximum timestamp among the match's events; the payload is the result of applying the mapping function (*map*) to the match's events.

²The symbol trace is the result of applying the symbols' condition to the events of a stream.

³We denote the set of all subsequences of an event stream E with $\mathbb{P}(E)$.

$$\rho_{PQ^M}(E) = \biguplus_{r \in \mathbb{P}(E)} \begin{cases} \langle (map(r), \max_{e \in r} e.t) \rangle & , \text{ if } r \text{ is a match according to } M \\ \langle \rangle & , \text{ otherwise} \end{cases} \quad (2.7)$$

In addition to BSs and KSs, state-of-the-art CEP languages (e.g., SASE+ [DIG07]) include additional operators like negation (i.e., non-occurrence of events), Kleene-plus symbols (variable-length sequences of at least 1 event), or nested sub-patterns (e.g., $A(BC)^+D$). While integrating those features into the core language allows for specific runtime optimizations (e.g., negation push-down [ZDI14]), we opt for a slim core-feature set present in most CEP systems to increase the applicability of our approaches. However, we discuss the support of those advanced language features in the corresponding sections.

Due to the high costs for evaluating sequential pattern queries, especially when using the *skip-till-any* strategy, the optimization of online CEP received considerable attention over the past decade. For instance, Zhang et al. [ZDI14] avoid the materialization of intermediate results, which reduces the overall resource consumption (i.e., CPU and main memory utilization). Other approaches target the same goal via lazy evaluation techniques [KSS15] or by adopting join optimization techniques to find the cheapest evaluation order of the pattern’s symbols [MM09; KS18]. Moreover, Ray et al. [RLR16], and Kolchinsky et al. [KS19] explore the benefits of sub-pattern sharing in the presence of multiple concurrent queries.

Orthogonal to those algorithmic optimizations, various parallelization strategies like partition parallelism [Hir12], data parallelism [Bal+13] and pipeline parallelism [SMP09] are applied to improve the runtime performance on modern multi-core CPUs and in distributed setups. In addition to that, co-processors such as GPUs [CM12] or FPGAs [WTA10] have shown to be efficient accelerators for sequential pattern matching.

Finally, closely related to sequential pattern matching is the idea of event trend aggregation. The goal of event trend aggregation is not to detect and output every single match of a given pattern but instead compute aggregations over finite sets of matches (e.g., all matches within a time window). Poppe et al. [Pop+17; Pop+19] recently proposed an approach that computes event trend aggregations under *skip-till-any* semantics with quadratic time and linear space complexity (compared to exponential space and time complexity for computing every match).

2.4 Situations

Events are notifications about something that happened in the real world at a specific point in time (e.g., the temperature in Marburg on December 24th, 12:00 was 0°C). However, many real-world phenomena do not exist for a single point in time but last for a while (e.g., the temperature was below 5°C for the whole of December 2020). We call such events spanning a period of time situations.

Definition 2.19 (Situation). A situation $s = (a_1, \dots, a_d, ts, te)$ is a $d + 2$ dimensional tuple consisting d attributes (a_1, \dots, a_d) and two timestamps (ts, te) . We denote the domain (e.g., integer, string) of the i -th attribute with \mathcal{A}^i ; ts and te are from a discrete, totally ordered time domain \mathcal{T} with $ts < te$. The half-open time interval $[ts, te)$ specifies the validity of a_1, \dots, a_d . Without the loss of generality, we assume $\mathcal{T} = \mathbb{N}$. Analogous to events (Definition 2.4), we use the dot notation to access a situation's attributes $(s.a_i)$ and timestamps $(s.ts, s.te)$. We also summarize the d attributes of a situation as payload $(s.p)$ over the d -dimensional domain $\mathcal{P} = \mathcal{A}^1 \times \dots \times \mathcal{A}^d$.

Definition 2.20 (Situation Stream). A situation stream S is a data stream (Definition 2.1) of situations $S = \langle s_1, s_2, \dots \rangle$. All situations of a stream are from the same domain and ordered according to their end timestamps (i.e., it holds $s_i.te \leq s_j.te$ for $i \leq j$).

Since situations are associated with a duration, they allow the detection of complex temporal relationships between real-world phenomena. For instance, rainfall *during* a period with a temperature below 0°C indicates snowfall. Such relationships are hardly expressible with timestamped events. Despite these exciting opportunities, the potential of situations (or interval-based events) in streaming applications has been widely overlooked. Pipes [KS04] and Microsoft StreamInsight [Ali+09] associate time intervals to data, but only use them to express time windows. Similarly, Cayuga [Bre+07] and ZStream [MM09] use intervals only to represent the length of partially matched patterns. In contrast, *ISEQ* [Li+11] is a system that allows to detect complex temporal relations in streams of interval events. However, this approach is limited to interval events only and thus is incompatible with event-based systems. In this thesis, we bridge this gap by introducing situations in an event-based system.

3

Index-Supported Offline Processing

Traditional DBMSs utilize indexes, such as B⁺-trees [BM70] or R-trees [Gut84], to quickly access the data relevant to a query (and avoid reading irrelevant data). However, for the offline evaluation of CQs, indexes have been widely overlooked so far. Instead, the common strategy is to load the entire event stream from secondary storage and replayed into an online SPE. While this ensures unified query semantics for on- and offline processing, the cost for loading large streams from secondary storage severely hurt the overall query runtime.

This chapter, leverages indexes to accelerate the offline evaluation of two essential operators: windowed aggregation and sequential pattern matching. Therefore, we answer two crucial questions for efficient index utilization: (Q1) *How can an index improve the overall query runtime?*, and (Q2) *How many and which of the indexes to use?*

For basic filter operations, the results of an index query directly translate into an answer of the original query, making (Q1) a trivial question. In contrast, a match for a pattern query is a variable-length subsequence of a stream. Every event of such a subsequence fulfills a user-defined condition and obeys sequential and temporal distance constraints to its predecessor and successor elements (e.g., *A* followed by *B* within *w* time units). Moreover, while the bare symbols of a pattern frequently tend occur within the stream, the occurrence of their combination as given by the query is typically rare because most patterns aim to detect extraordinary situations of interest. Thus, an answer to (Q1) must effectively combine the positional and temporal properties of multiple symbols.

Unlike sequential pattern matching, aggregation queries do not benefit from traditional indexes on single attributes since aggregations do not require data filtering. Recently, Seidemann et al. [SS17] proposed a technique called lightweight indexing and showed that it dramatically improves the runtime of ad-hoc non-windowed aggregation queries over event streams. However, windowed aggregation computes a series of aggregations where adjacent windows may overlap to a great degree

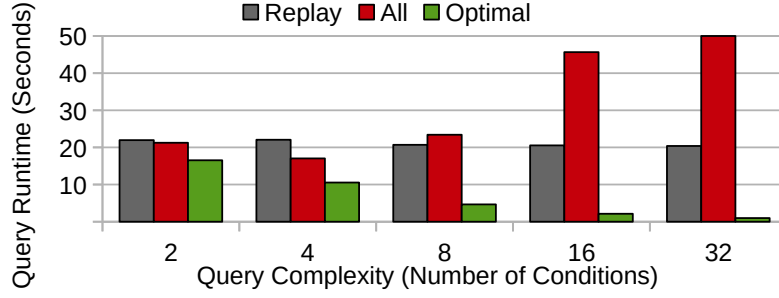


Figure 3.1: Query runtime for different pattern evaluation strategies as a function of query complexity.

(depending on the window parameters). Thus, answering windowed aggregation via a series of ad-hoc queries may lead to lots of redundant computations and especially redundant reads from secondary storage. Consequently, a satisfactory answer for (Q1) cleverly utilizes those lightweight indexes to avoid this redundancy.

The experimental results reported in Figure 3.1 illustrate the challenge of (Q2) for sequential pattern matching queries. They show the query runtime as a function of the query complexity (i.e., the number of BSs) for three evaluation strategies. The runtime of a full stream replay (Replay) is nearly constant since the query is I/O bound, and the actual pattern evaluation only contributes very little to the overall runtime. Using all available indexes (All) can speed up queries up to a certain degree of complexity. However, at some point, the costs of accessing the indexes are so high that the total execution costs exceed the costs of a full stream replay. Nevertheless, as shown by the third strategy (Optimal), an optimal selection of indexes drastically reduces the query processing time for all degrees of query complexity. A similar behavior can be observed for windowed aggregation queries when varying the window’s *size* and *slide* parameters.

In this chapter, we develop new index-based evaluation techniques that minimize secondary storage access and thus improve the overall query runtime compared to replay-based approaches (Q1). Moreover, we develop cost models and index selection strategies to determine the most efficient evaluation strategy for a given query (Q2). To the best of our knowledge, those are the first index-based evaluation techniques, cost models, and index selection strategies for CQs on persistent event streams.

The remainder of this chapter is structured as follows. In Section 3.1 we review related work. Section 3.2 introduces *ChronicleDB*, a special-purpose database system for event streams. We use *ChronicleDB* as the basis for our implementations

because it offers excellent stream replay performance and support for indexes on secondary stream attributes. Sections 3.3 and 3.4 detail our algorithms and cost-models for windowed aggregation and sequential pattern matching, respectively. We experimentally evaluate both approaches in Section 3.5 and summarize our findings in Section 3.6.

3.1 Related Work

This section, discusses research that serves as a basis for our index-based operator implementations. We first discuss work related to windowed aggregation before we elaborate on sequential pattern matching.

3.1.1 Windowed Aggregation

In Section 2.3.3, we already discussed evaluation techniques for processing windowed aggregation over event streams in an online fashion. Thus, here we focus on computing them offline. Windowed aggregation over persistent event streams has not been addressed in the literature so far. However, the basic idea is closely related to aggregation queries in temporal databases (so-called temporal aggregation queries) [KS95]. A (temporal) relation in a temporal database reflects its current state and captures its entire history (i.e., all insertions, deletions, and updates). Therefore, the database associates a validity interval of the form $[ts, te]$ to every record. For instance, for a record inserted at time t , the validity interval initially is $[t, \infty]$. An update at $t' > t$ to this record then modifies the interval to $[t, t' - 1]$ and inserts the new version of the record with interval $[t', \infty]$. A temporal aggregation query over such a relation processes the entire history and outputs a sequence of aggregation values that reflect every state of the relation.

In 1995, Kline and Sondgrass [KS95] presented a first approach to answer those queries efficiently. They propose an in-memory tree structure that is built online during a single scan of the temporal relation. The aggregates can then be extracted via a depth-first traversal of this tree. However, their approach has two major drawbacks. First, it is unfeasible for large temporal relations since it keeps all data in the main memory. Second, the worst-case runtime of aggregate retrieval is $\mathcal{O}(n^2)$, since the tree structure is unbalanced (i.e., it may degenerate to a linked list). Nevertheless, based on this approach, several optimizations were proposed. For instance Moon et al. [MLI03] use a balanced tree to reduce the worst-case runtime to $\mathcal{O}(n \log(n))$. In addition, they propose a partition technique to handle relations that do not entirely fit in the main memory.

Böhlen et al. [BGJ06] expand upon this classical temporal aggregation by introducing the idea of cumulative temporal aggregation. Besides the actual aggregation functions, cumulative temporal aggregation queries take an additional *duration* parameter. Then, an aggregated value at time t covers all records whose interval intersects with $[t - \textit{duration}, t]$. While this somewhat resembles sliding time windows, the results are still aligned at the record timestamps and not on window boundaries. Moreover, they do not use indexes but scan the entire temporal relation to compute query results.

The Timeline Index [Kau+13] is a general-purpose (persistent) index structure that is applicable to a wide range of temporal queries (temporal aggregation, time travel, temporal join). However, it does neither support windowed aggregation queries nor cumulative temporal aggregation. Moreover, temporal aggregation queries require a full index scan, which holds at least one entry per record of the temporal relation. Hence, the time complexity is similar to a full relation scan. The work most closely related to our idea is the SB-tree [YW03]. The SB-Tree is a disk-resident index that associates time intervals with partial aggregates and supports cumulative temporal aggregation. However, similar to the approach of Böhlen et al. [BGJ06] windowed aggregates are not supported. Moreover, for cumulative aggregation multiple indexes must be maintained and queried. In contrast, our approach only uses a single index to support windowed aggregation (even for multiple aggregation functions and count windows).

3.1.2 Sequential Pattern Matching

Besides evaluation strategies for online pattern matching, our approaches are related to pattern matching in (temporal) database systems and join processing techniques. We already discussed work related to online sequential pattern matching in Section 2.3.5. Thus, we focus on the remaining related concepts of index-based sequential pattern matching. Nevertheless, our approach can be combined with and benefit from all online optimizations mentioned in Section 2.3.5.

String matching. Finding all occurrences of a substring within a larger string is the most basic form of pattern matching over strings. In 1977, Knuth, Pratt, and Morris [KJP77] presented a linear time online algorithm that accomplishes this task with a single sequential scan of the input string. Indexes can considerably speed up the search process when searching for many different substrings in the same input. Suffix trees [Wei73; McC76] and suffix arrays [MM93] allow for substring searches in logarithmic time; wavelet trees reduce search complexity even further to sublogarithmic time [Nav14]. Moreover, Baeza-Yates and Gonnet [BG96] showed how to use suffix trees (or suffix arrays) to match regular

expressions (instead of substrings) in sublinear time. In order to reduce the super-linear space complexity of suffix trees and related approaches, Cho et al. [CR02] propose to index multigrams (i.e., short substrings) for regular expression matching. Their index maps multigrams to positions in the source string. This mapping allows the pruning of irrelevant data, similar to our approach. Tsang et al. [TC11] further improved this idea by providing a method to select the optimal set of multigrams.

Another line of research considers the case of matching a (possibly evolving) set of regular expressions to a stream of input strings. For example, the RE-Tree [CGR03] indexes finite automata representing the regular expressions in an R-tree-like structure. The inner nodes store a coarse representation of the underlying expressions, allowing early pruning of non-promising expressions. Similarly, sigmatch [KTP10] utilizes a combination of tries and bloom filters to discard inputs where none of the indexed patterns may occur. Thereby they focus on cache-efficiency for modern CPUs. Each of the discussed indexes requires that the alphabet of the regular expressions is finite and known a-priori. However, in event pattern matching, the alphabet is dynamically built from user-defined boolean expressions per symbol (e.g., range conditions), which renders the use of those specialized indexes impossible. Therefore, we evaluate the boolean expressions via widely used indexes, like B⁺-trees, and utilize the results to prune irrelevant data from the stream.

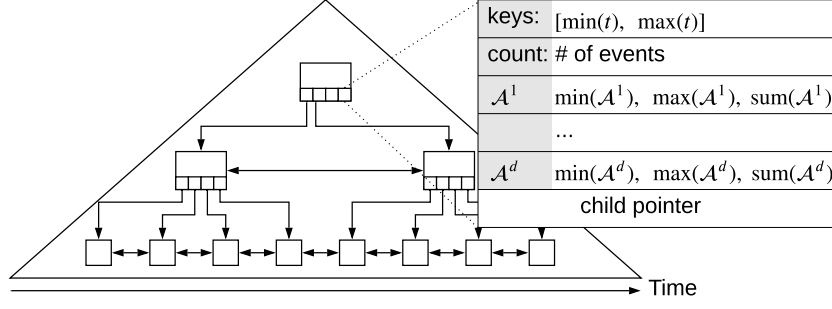
Pattern Matching in Database Systems. In the 1990s the active database community presented several complex event detection models (see [ZU99] for an excellent survey). Those models cover all features available in modern online systems like sequence detection, negation, or alternatives. However, efficient evaluation strategies or index support are not discussed. At the same time, relational database systems were extended with sequence processing capabilities. SEQ [SLR95] introduces a sequence data type allowing to correlate sequence elements based on their position in the sequence, thus enabling the evaluation of simple sequential patterns (without recurring symbols). SRQL [Ram+98] includes those features into the SQL language. SQL-TS [Sad+01; KMS08] adds explicit support for sequential pattern matching, including recurring (nested) patterns and presented efficient evaluation techniques. However, those techniques focus on avoiding multiple passes over the sequence data and do not consider indexes to exclude fractions of the sequence from processing. Deja-Vu [Din+11] allows the user to perform pattern matching on live and historical data. However, they focus on how to combine the results from queries in on- and offline systems and do not consider indexes for pattern evaluation on historical data. Valdés et al. [VG14] use indexes to speed up pattern matching queries in trajectory databases. In contrast to event pattern

matching, they match whole trajectories and not subsequences of a stream. Hence, this approach does not support time window constraints. Furthermore, instead of selecting a subset, this approach uses all indexes available, which can hurt processing performance considerably. Garcia-Arellano et al. [Gar+20] introduce a new type of event store, but they neither support pattern matching queries nor secondary indexes to improve query processing. However, they announced secondary indexes for future releases. Seidemann et al. [Sei+19] present a first approach for pattern matching in event stores. However, this approach is limited in two ways. First, it greedily selects indexes until the index access costs exceed the cost gain from pruning events. This approach often uses less than the optimal number of indexes because it quickly gets stuck in a local minimum. Second, it does not consider sequential distance constraints, which leads to reduced pruning power of the indexes.

Join Processing. Sequential pattern matching essentially is an n-way self join with limited temporal and sequential scope. The temporal database community extensively studied the efficient execution of manifold temporal join variants in the presence of indexing [EWK90; ZTS02; Kau+13] as well as on non-indexed data [Gao+05]. However, they all focus on the temporal dimension and do not consider sequential distance constraints (e.g., A before B with no events in between) present in a pattern query. Another important research direction regarding join processing is cardinality estimation and access path selection. A major part of our work is a cost model that decides whether or not to use an index for every element of a pattern query, i.e., select an access path. The seminal work in this area [Sel+79] uses simple statistics based on the cardinalities of the input relations to choose an access path. Over the years, the community developed more sophisticated cardinality estimation methods. Mannino et al. summarizes many of them in [MCS88]. More recent models, as presented by Kester et al. [KAI17], include aspects like concurrent queries, the in-memory data layout, data compression, and the cache-hierarchy of modern CPUs. In addition, Dutt et al. [Dut+19] increase the accuracy of selectivity estimates using machine learning techniques. However, those methods focus on estimating the number of join results and do not provide any information about the sequential distance of join partners, which is vital when dealing with sequential patterns.

3.2 ChronicleDB

We implement our approaches in *ChronicleDB* [SS17; Sei+19], a special-purpose database system for managing high volume event streams. It stores events in a


 Figure 3.2: TAB^+ -tree index layout with lightweight indexing.

primary index that is a streaming version of an append-only B⁺-tree with \mathcal{T} as its key domain. This index is called *Temporal Aggregated B⁺-tree* (TAB^+ -tree). For fast insertions, *ChronicleDB* primarily adopts an append-only model, where the data log is also the database. This model is reflected by the index design. As the default behavior, insertions are a continuous bulk loading operation in a traditional B⁺-tree index. Under the assumption that the event stream E is in temporal order, a new event is appended to the leaf node containing the most recent data (see [SS17] for the handling of out-of-order events). Consequently, both insertions and range queries on the temporal domain exhibit a sequential I/O pattern, which results in excellent insert and replay performance.

Besides the primary index, *ChronicleDB* allows adaptive and ad-hoc creation of two kinds of secondary indexes referred to as heavyweight and lightweight indexes. Heavyweight indexes are traditional secondary index structures such as LSM [ONe+96] or COLA [Ben+07], built for one or more attributes of the stream. The entries in leaf pages of heavyweight indexes refer to primary index pages with a record offset. Lightweight indexes are subject to the upcoming subsection.

3.2.1 Lightweight Indexes

Lightweight indexes are an adaptation of small materialized aggregates (SMAs) [Moe98]. SMAs store partial aggregates over a small subset of the data (e.g., a page) in a separate file. Aggregation queries are then answered by scanning this file and merging the partial aggregates. In contrast, *ChronicleDB* stores those aggregates within the inner index nodes of the TAB^+ -tree. Every child reference is associated with aggregated information of their respective nodes as shown in Figure 3.2. An

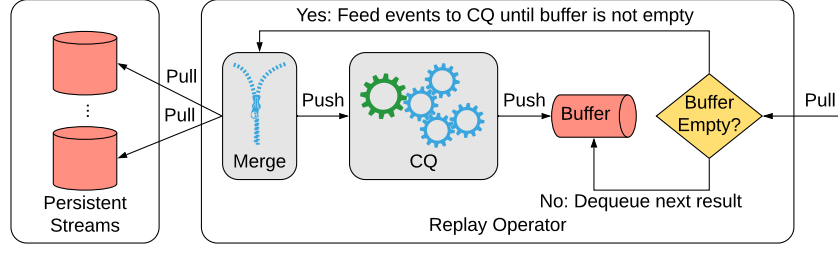


Figure 3.3: Dataflow of the generic replay operator.

index entry consists of an overall event count aggregate and the minimum, maximum, and sum values for each attribute. Those materialized aggregates allow to answer ad-hoc temporal aggregation queries (i.e., aggregations over a user-defined time interval) in logarithmic time. Moreover, lightweight indexing can boost the performance of filter queries tremendously. For instance, large portions of the stream can be excluded from processing by intersecting a query range on a secondary attribute with its materialized min/max range.

We call this technique lightweight because the required aggregates are computed incrementally during ingestion, offering indexing support at minimal (i.e., linear) build-up costs.

3.2.2 Replay-Based Continuous Query Evaluation

We use replay-based query execution as a baseline during the evaluation of our index-based approaches. However, *ChronicleDB* offers a pull-based iterator interface, while ESP systems typically use a publish/subscribe interface to deliver results in a push-based manner. This means events arriving at an ESP system trigger the processing of operators, which in turn send their results to downstream operators.

Thus, to seamlessly integrate event-driven query execution into the pull-based interface of *ChronicleDB*, we implemented a generic replay operator. The operator wraps a CQ and offers a pull-based interface for accessing the query results. Figure 3.3 details the architecture of the replay operator. We treat the CQ as a black box and forward query results to an in-memory buffer. In order to feed the CQ with input data from *ChronicleDB*, we issue a replay query for every stream specified as input for the CQ. Then, we merge the results of those queries into a single virtual stream ordered by the events' timestamps. When an application asks for the next result from the replay operator, we first probe the in-memory result buffer.

If there is a result, it is removed from the buffer and returned. Otherwise, we pull events from the source streams and push them into the CQ via the virtual input stream. We repeat this step until one or more results are reported by the CQ, or the virtual input stream is empty.

3.3 Index-Based Windowed Aggregation

Our online implementation of windowed aggregation is an adoption of the tree-based aggregation method presented by Tangwongsan et al. [Tan+15]. The implementation exploits the fact that windowed aggregation produces exactly one result per window slide (cf. Definition 2.11). It manages a balanced tree (2–3–4 tree) of partial aggregates. Every leaf entry of the tree holds aggregated event data for exactly one window slide. The inner nodes also hold partial aggregates, each of them composed from the partial aggregates of its children. Thus, by ensuring that all leaf nodes constitute a window, the result for this window can be obtained from the root node in constant time. When the window slides forward, the eldest leaf is evicted, and a new leaf is inserted into the tree. Both operations incur logarithmic costs. Moreover, every incoming event is processed to compute the partial aggregates of a slide/leaf. Thus, online aggregation incurs constant cost per event plus logarithmic cost per window slide.

3.3.1 Index-Based Processing

Independent of the slide parameter value, the online implementation processes every event of the input stream exactly once, which is already very efficient for small *slides*. For larger *slides*, we can take advantage of *ChronicleDB*'s lightweight indexing by issuing appropriate temporal aggregation queries to omit visiting every event of the current window/slide. Of course, this is only possible if the requested aggregate is materialized (cf. Section 3.2.1).

A straightforward implementation of this approach is to issue a temporal aggregation query for every window from scratch (Figure 3.4 left). However, this approach has two deficiencies. First, two consecutive windows may considerably overlap (depending on the slide parameter), which leads to redundant computations. Second (and more critical), this method may induce a random disk access pattern. In general, a temporal aggregation query accesses the leaf level of the *TAB⁺-tree* twice: Once for the left bound and once for the right bound of the specified time range – except for the unlikely case that the queried time range aligns with the

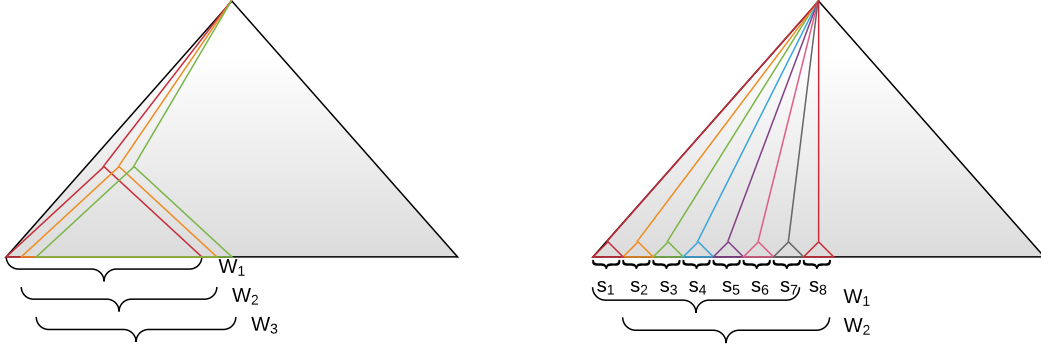


Figure 3.4: Index-based processing of sliding window aggregates: naive approach (left) and optimized approach (right).

time bounds on any level of the tree. Note that the tree path traversed for the left bound at window i is the same as the right path for window $i - \frac{\text{size}}{\text{slide}}$. Consequently, this approach induces a forward scan only if the database buffer is large enough to hold all pages required for answering $\frac{\text{size}}{\text{slide}}$ queries. Otherwise, there are two (random) accesses per window.

To overcome these deficiencies, we implemented a different approach. We use the tree structure of the online aggregation as described above. However, instead of aggregating every single event, we issue a temporal aggregation query to compute the partial aggregate of a single *slide* in one go. As can be seen in Figure 3.4 (right) the queries for slide s_1, \dots, s_8 do not overlap. Hence, we do not execute redundant computations. Furthermore, the left tree path of the query of s_i is equal to the right path of the query of s_{i-1} . Thus, this method requires only a fraction of the database buffer (twice the tree height) for a forward scan access pattern. The results of those queries are then directly inserted into the 2-3-4 tree, from which we extract results of the entire window (e.g., W_1, W_2 in Figure 3.4 right). As we will show in our experimental evaluation, this approach outperforms the naive approach (i.e., one temporal aggregation query per window) for any *slide* value.

3.3.2 Cost Estimation

In the following, we present how to estimate the costs for both the replay-based and the optimized index-based version in order to decide which method to use

Symbol	Description
r	Event rate (events per time unit).
h	Height of the TAB^+ -tree.
B	Leaf blocking factor (events per leaf node)
D	Fan-Out (child references per inner node).
L	leaves per slide; $L := \frac{slide \cdot r}{B}$.

Table 3.1: Symbols used in the cost estimation for windowed aggregation.

in a given scenario. Table 3.1 summarizes the symbols used throughout the next paragraphs.

From an I/O perspective, the index-based approach should be faster than a replay when the slide parameter covers a certain amount of leaf nodes. Hence, we should see improvements if $slide \cdot r$ is greater than the average number of events per page since this allows us to skip the loading of (at least some) leaf nodes. In fact, modern solid state drives (SSDs) and hard disk drives (HDDs) merge multiple small random reads into a single sequential read, up to a certain degree. Moreover, those disks feature sophisticated caching and read-ahead strategies, making it impossible to predict the actual access pattern reliably.

Thus, we base our cost estimation on the number of required computations. Because we reuse the online aggregator’s tree structure, the only difference in computational costs between both methods is the number of partial aggregate merges performed per slide. The costs for obtaining a window’s final aggregate values and the management of the 2-3-4 tree are identical because both approaches insert and evict partial aggregates on a per-slide-basis. Thus, it is sufficient to estimate the number of partial aggregate merges per slide to compare both approaches. As already stated above, the online implementation performs exactly one merge per event. Hence, the costs for this variant are estimated as: $C_{\text{replay}}(slide) = slide \cdot r$. For the index-based approach, we count the merges while traversing the tree. This number is composed of the following four cases (cf. Figure 3.5).

1. 1 merge per tree level, as long as the queried time interval is fully covered by one child reference.
2. $j - i$ merges on the level where the path splits (l_s). Here, i denotes the index of the child reference intersecting the query range’s lower bound and j the index of the reference intersecting the upper bound.

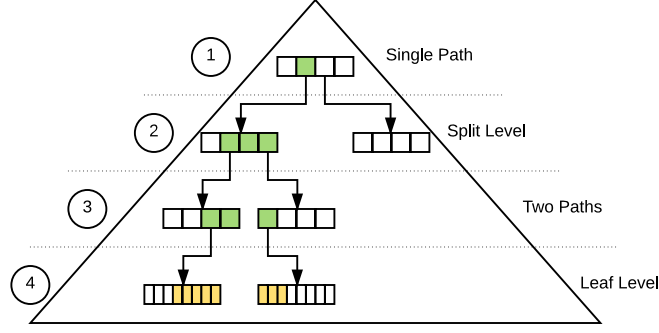


Figure 3.5: Illustration of performed partial aggregate merges during a temporal aggregation query.

3. For every level below (except for the leaves), we estimate $\frac{D}{2}$ for each of the two paths, resulting in approx. D merges per level.
4. We do the same on the leaf level, resulting in a constant of B merges. There is one exception to that rule: if $slide \cdot r < B$, the query visits only a single path down to that leaf. Thus, we add $(slide \cdot r)$ instead of B .

With the split-level computed as $l_s := \lceil \log_D(L) \rceil$ (0 is the leaf level), the costs are summarized as

$$C_{\text{index}}(s) = \underbrace{(h - 1 - l_s)}_{(1)} + \underbrace{\left(\min(1, l_s) \cdot \left\lceil \frac{L}{D^{l_s-1}} \right\rceil \right)}_{(2)} + \underbrace{(\max(0, l_s - 1) \cdot D)}_{(3)} + \underbrace{(\min(1, L) \cdot B)}_{(4)}$$

3.3.3 Arbitrary Slide Sizes and Count Windows

So far, we assumed the size to be a multiple of the slide parameter. To overcome this limitation, we split slide into two parts: $slide_1 = slide - (size \bmod slide)$ and $slide_2 = (size \bmod slide)$. Thus, the full size of a window can be composed as

$$size = slide_2 + \left\lfloor \frac{size}{slide} \right\rfloor \cdot (slide_1 + slide_2)$$

After populating the tree aggregator with alternating slide slices ($\langle slide_2, slide_1, slide_2, \dots, slide_1, slide_2 \rangle$), the computation for each slide removes the first two and inserts two new slices into the tree before computing the result of the next window. Note that we always remove and insert slide pairs of sizes $slide_1$ and $slide_2$ and hence guarantee the correctness of the produced results. In the general case of

an arbitrary slide size, the costs per slide are the sum of the costs for both slide slices.

$$C_{\text{index}}(\text{slide}) = C_{\text{index}}(\text{slide}_1) + C_{\text{index}}(\text{slide}_2).$$

Count Windows

Count windows are processed similar to time windows: We merge aggregates at the highest possible tree-level and proceed to lower levels only if the granularity of the materialized aggregates is too coarse to fit the current slide. In order to determine the slide boundaries, we utilize the materialized event count stored with every index entry of the *TAB⁺-tree* (cf. Figure 3.2). Since the count is not an absolute value (like the timestamps), we require an additional variable to track the number of already processed events.

3.4 Index-Based Pattern Matching

The predicates associated with the symbols of a pattern query may be arbitrary complex. For instance, the cooling system example in Section 2.3.5 compared current with previous temperature readings. Moreover, predicates may involve multiple conditions connected via boolean operations (e.g., logical AND, logical OR).

However, many pattern queries utilize fixed threshold ranges that limit a symbol’s valid attribute values. Use cases featuring such threshold ranges include the medical field (heart rate monitoring), traffic monitoring (car speed), technical infrastructure monitoring (temperature sensors), and financial analysis (stock prices). As an example, consider the detection of illegal insider trading activities in financial market surveillance. According to the U.S. Securities and Exchange Commission, “illegal insider trading refers generally to buying or selling a security, in breach of a fiduciary duty or other relationship of trust and confidence, on the basis of material, nonpublic information about the security” [20]. Thus, a sharp increase in stock prices due to large order placements indicates insider trading activities. In Figure 3.6, we describe this situation with the pattern ABC^*D , using the four conditions A, B, C, D and $M = \text{contiguous}$. The intuition behind this specification is as follows. The price starts underwhelming (A). After the placement of a large order (B), the price increases to new heights (D). To account for reaction time between B and D , we allow an always true condition of variable length (C^*) in between. Note that in this example, every BS is associated with a fixed threshold condition on either the *price* or the *volume* attribute.

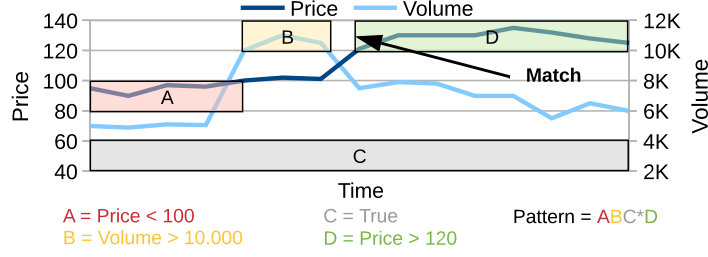


Figure 3.6: Definition of the insider trading detection pattern using the *contiguous* matching strategy. The shaded areas highlight for every symbol the fractions of the stream where the corresponding predicate holds.

The core idea of our approach is to delegate those threshold predicates to secondary indexes. A clever combination of the index results returns a set of temporal intervals, termed replay intervals in the following. Those intervals represent the regions of the event stream where all possible matches exist. In order to compute the results of a pattern matching query, it is then sufficient only to replay the events falling into those replay intervals. At first glance, using all available indexes to construct the replay intervals with minimal coverage minimizes the processing cost. However, as shown in Figure 3.1, using too many indexes quickly drives the total query execution cost beyond the cost for a full stream replay.

We illustrate this effect and our approach using the insider trading detection query. Assume that an index is available on attributes *price* and *volume*. Hence, all BSs (A, B, D) can be found via index queries. Figure 3.7 shows the replay intervals (indicated by the grey boxes) when different indexes are selected. In Figure 3.7 (a) the intervals containing potential matches are found by using a single index on symbol A . For every occurrence a_i of symbol A , $i = 1, 2, 3$, we compute the replay interval as $[a_i.t, a_i.t + w]$. It is obvious that no match is outside of these intervals. In Figure 3.7 (b), the combined information obtained from two index queries for symbols A and B allows us to prune two of the replay intervals. We dismiss the first interval because, even though b_1 occurs within the interval of a_1 (and vice versa), they are not adjacent as the pattern requires. Additionally, we discard the third interval because no B exists within the interval of a_3 . Finally, Figure 3.7 (c) uses all three indexes to compute the replay intervals. However, this does not reduce the intervals compared to Figure 3.7 (b) anymore and causes additional cost due to the extra utilization of an index.

Thus, we have to address two non-trivial problems. The first is the calculation of replay intervals with minimum coverage for a given set of indexes. This is discussed

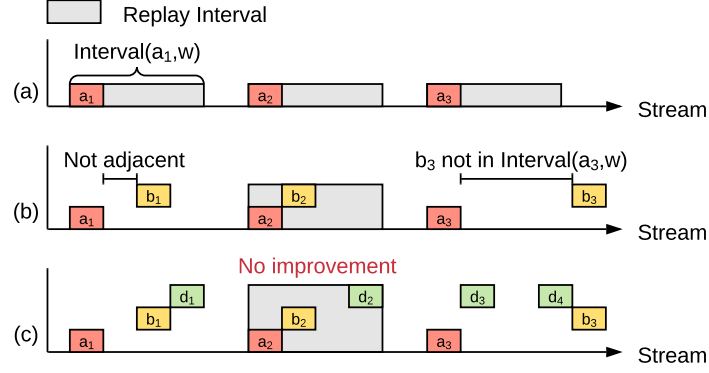


Figure 3.7: Index-based matching of the example pattern ABC^*D highlighting the replayed fraction of the stream when using (a) an index for symbol A , (b) indexes for symbols A and B , and (c) indexes for symbols A, B and D .

in Section 3.4.2. The second problem is the selection of indexes that minimize the overall processing cost, which is the subject of Section 3.4.3.

3.4.1 Preliminaries

As shown by the previous example, our approach uses secondary indexes to evaluate the predicates of the BSs of a given pattern query PQ^M . In the following, we use \mathcal{S}_{idx} to refer to the set of BSs of PQ^M that offer an index for predicate evaluation and denote its with m . Furthermore, we use $\mathcal{S}'_{idx} = \{S'_1, \dots, S'_k\}$ to refer to a subset of \mathcal{S}_{idx} of size k .

We restrict our approach to BSs because they are mandatory for any match of the pattern. In contrast, a successful match not necessarily contains events for a KS. Thus, we can not discard replay intervals based on the non-occurrence of KSs. However, in combination with adjacent BSs, KSs could be exploited to impose tighter sequential distance constraints. We leave this opportunity to future work.

Pattern Matching Queries

In order to support tight sequential distance constraints, like A followed by B with no events in between, we primarily consider the *contiguous* matching strat-

egy¹. Sequential constraints add additional pruning opportunities that should be exploited and reflected in the cost model.

We also assume that symbols are associated with range conditions on single stream attributes because these directly translate into queries on widely used indexes (e.g., B⁺-trees). In addition, pattern queries frequently use such range conditions to define thresholds (e.g., on heart rates, temperatures, or stock prices).

In Section 3.4.4, we show how to apply our approach to matching strategies other than *contiguous*, how to handle more complex and compound predicates, and how to apply our approach to richer languages (e.g., languages featuring negation or Kleene-plus symbols).

Secondary Indexes

Our approach requires range-query support by the secondary indexes to efficiently look up the pattern’s symbols. We opt for *ChronicleDB*’s built-in LSM-trees [ONe+96], as they offer excellent read and write performance. However, it is possible to transfer our approach to other ordered indexes like B⁺-trees [BM70] or LHAM [Mut+98].

For our purpose, the entries of secondary indexes consist of a triple (key, seq, t) and a pointer to the corresponding event in the primary index. The triple is composed of the indexed attribute value (key), the absolute position of the event within the stream (seq), and the event’s timestamp (t). These triples allow us to check both temporal and sequential distance constraints of a pattern query without accessing the primary index.

3.4.2 Replay Interval Computation

Our approach first computes a set of candidates from which we derive the replay intervals. Let Idx_1, \dots, Idx_k be the secondary indexes that correspond to the symbols in \mathcal{S}'_{idx} . Furthermore, let $i_1 < \dots < i_k$ denote the positions/indices of the associated symbols within PQ (i.e., $S'_1 = S_{i_1}, \dots$). A candidate $c = (c_1, \dots, c_k)$ is a sequence of k events such that

- (C1) c_j is a result of the range query on index Idx_j , i.e., c_j satisfies the range condition of Symbol S'_j ,
- (C2) $c_k.t - c_1.t \leq w$,

¹If not stated otherwise PQ refers to PQ^M with $M = \text{contiguous}$ for the remainder of this section.

(C3) $c_j.seq - c_{j-1}.seq = i_j - i_{j-1}$ if none of the symbols S_q , $i_{j-1} < q < i_j$ is a KS.

Condition (C1) only permits candidates where the selected events satisfy the range conditions of the associated symbols, while (C2) ensures that a candidate is entirely within the window w . Finally, condition (C3) expresses sequential distance constraints. For instance, two adjacent BSs require a sequential distance equal to 1.

Due to the temporal window w , it follows that the replay interval for a candidate c is at most as large as $[c_k.t - w, c_1.t + w]$. However, in general, S'_1 and S'_k are not the first and the last symbol in the original pattern query, respectively. Because the temporal distance between two adjacent events is at least 1, we can shorten the replay interval at the upper bound by the number of BSs between S_1 and S'_1 . Similarly, for the lower bound, we shorten the interval by the number of BSs between S'_k and S_n . This leads to the following definition for the replay interval of a candidate c :

$$Interval(c, w) = [c_k.t - w + \oplus_{bs}(i_k + 1, n), c_1.t + w - \oplus_{bs}(1, i_1 - 1)].$$

The function $\oplus_{bs}(i, j)$, $i < j$, returns for $i \leq j$ the number of BSs from S_i to S_j in the original query. Otherwise ($i > j$), it returns 0. This interval computation represents the base case. We can shorten the interval even further for the three special cases $i_1 = 1$ and/or $i_k = n$, i.e., an index is accessed for the first, last or for each of these symbols of the pattern. Let us consider these special cases in the following:

(S1) If $i_1 = 1 \wedge i_k < n$ then $Interval(c, w) = [c_1.t, c_1.t + w]$.

(S2) If $i_1 = 1 \wedge i_k = n$ then $Interval(c, w) = [c_1.t, c_k.t]$.

(S3) If $i_1 > 1 \wedge i_k = n$ then $Interval(c, w) = [c_k.t - w, c_k.t]$.

Depending on the temporal distribution of candidates, the corresponding replay intervals may overlap or even contain each other. Hence, the final step of our approach is to merge overlapping intervals. Then, the result is a sequence of disjoint replay intervals.

The candidates for a given pattern query, and thus the replay intervals, can be computed efficiently by sorting the index results according to the events' sequence number followed by two merge steps. The first merge step discovers all occurrences of the basic blocks (BBs) for the given pattern. A BB is a contiguous sequence of BSs without any KS in between. For instance, the BBs of the pattern (A, B, C^*, D) are (A, B) and (D) . Within a BB all symbols must obey fixed sequential distance constraints with each other. In our example, an occurrence of A must directly

Algorithm 3.1: Create Replay Intervals

Input: $(BB_1 = (S'_1, \dots, S'_{i_1}), \dots, BB_l = (S'_{i_{l-1}+1}, \dots, S'_k))$: Selected BBs
Input: (Idx_1, \dots, Idx_k) : Associated secondary indexes
Input: w : Time window
Output: Set of disjoint replay intervals

```

1  $res \leftarrow \emptyset$ ;
2  $bbs_1, \dots, bbs_l \leftarrow \emptyset$ ;
3 foreach  $i \in 1 \dots l$  do
4    $results \leftarrow \emptyset$ ;
5   foreach  $S_j \in BB_i$  do
6      $results_j \leftarrow query(S_j, Idx_j)$ ;
7      $sort(results_j, by = seq)$ ;
8      $results \leftarrow results \cup \{results_j\}$ ;
9    $bbs_i \leftarrow merge(results)$ ;
10 foreach  $cand \in merge(bbs_1, \dots, bbs_k)$  do
11   if  $cand.t_{max} - cand.t_{min} < w$  then
12      $res \leftarrow res \cup \{Interval(cand, w)\}$ ;
13 return  $MergeIntervals(res)$ ;
    
```

be followed by an occurrence of B . We validate these conditions by comparing the events' sequence numbers (seq) stored for every entry in the secondary indexes.

The second merge step then assembles all candidates from the results of the first merge step. A candidate contains one occurrence for every BB of the pattern in the specified order and within the window constraint w . In our example, a candidate must contain an occurrence of (A, B) followed by an occurrence of (D) , where $D.ts - A.ts < w$. Note that the result sets for every BB are still sorted according to the sequence number. For instance for two occurrences $(a_1, b_1), (a_2, b_2)$ of the BB (A, B) it holds $a_1.seq < a_2.seq$ and $b_1.seq < b_2.seq$. Moreover, since event streams are ordered by the timestamp, it also holds that $a_1.t < a_2.t$ and $b_1.t < b_2.t$ (i.e., the result sets are also temporally ordered). Thus, candidates can be found by performing a multi-way merge between the results sets of all BBs and validating the window constraint using the timestamp information of the contained events. While we used all symbols/BBs in our example, this approach can also be applied if only a subset of symbols/indexes ($S'_{idx} \subseteq S_{idx}$) is used, by ignoring the constraints/symbols not contained in S'_{idx} .

We summarize our sort-merge-merge approach in Algorithm 3.1. The algorithm

receives a subset of BSs (grouped into BBs as described above) and the corresponding indexes as parameters. For every BB we first query the indexes of the contained BSs. Then, we sort and merge the query results to obtain occurrences for each of the BBs (Lines 3-9). In the second phase, we find candidates by merging occurrences of BBs. If such a candidate obeys the window constraint, we apply the $Interval(c, w)$ function and add the corresponding replay interval to the result set (Lines 10-12). Finally, we merge overlapping intervals and return them (Line 13).

We find all candidates within a single sweep over the sorted index results by implementing both merge steps and the interval merge in a streaming fashion. Moreover, it is easy to see that we do not miss possible matches because we apply the $Interval(c, w)$ function to every candidate before merging.

3.4.3 Index Selection

Next, we present our cost-based index selection strategy for computing the optimal set of indexes $\mathcal{S}_{idx}^* \subseteq \mathcal{S}_{idx}$ to answer a given pattern query PQ . We base our cost estimation on the selectivity of BSs and assume knowledge about those selectivities. For a symbol $S \in \mathcal{S}_{idx}$ we denote the corresponding selectivity with φ_S .

The remainder of the section is organized in the following way. We first identify different factors that determine the execution cost of a pattern query and introduce a cost formula to be minimized. Then, we detail our index selection strategy. Finally, we discuss the I/O cost for both primary and secondary index access.

Cost Model

The costs for executing a pattern matching query using Algorithm 3.1 consist of four components:

- C_{PIO} the I/O cost for accessing the primary index (in pages),
- C_{SIO} the I/O cost for accessing the secondary indexes (in pages),
- C_{sort} the cost for sorting the secondary index results, and
- C_{eval} the cost for the pattern evaluation.

Independent of the specific sorting technique (e.g., external or in-memory), C_{sort} is monotonically increasing in the number of results returned from the secondary index queries. Assuming equal-sized entries in secondary indexes, we can derive the number of results, and hence the sorting cost, directly from C_{SIO} . Similarly, C_{eval} monotonically increases in the fraction of the stream covered by the replay intervals (i.e., in C_{PIO}). Both, C_{PIO} and C_{SIO} , are determined by the set of selected BSs with index support \mathcal{S}'_{idx} . The more symbols \mathcal{S}'_{idx} contains, the more data is read from secondary indexes, and thus the higher is C_{SIO} . However, the higher the number of symbols in \mathcal{S}'_{idx} is, the smaller the replay intervals get, and thus the lower is C_{PIO} . Given this, we estimate the query execution cost C_{query} for a given set \mathcal{S}'_{idx} as follows:

$$C_{query}(\mathcal{S}'_{idx}) = f(C_{PIO}(\mathcal{S}'_{idx})) + g(C_{SIO}(\mathcal{S}'_{idx})) \quad (3.1)$$

Here, f and g are monotonic functions that depend on the physical index layout and pattern matching and sorting algorithms. To give an intuition about the functions f and g , consider the following setting as an example. Secondary index results are sorted with an in-memory algorithm (e.g., quicksort). Hence, the expected runtime for this step is $\mathcal{O}(n \log n)$ in the number of results. The pattern matching algorithm is an automaton-based implementation using the *contiguous* matching strategy (similar to SASE+ [DIG07]). Based on this, we define f and g as follows:

$$\begin{aligned} f(x) &= \omega_{PIO} \cdot x \\ g(x) &= \omega_{SIO} \cdot x + \omega_{sort} \cdot (xB \cdot \log_2(xB)), \end{aligned}$$

where x is the number of physical pages given by C_{PIO} and C_{SIO} , respectively. For the *contiguous* matching strategy we experimentally determined that the pattern evaluation cost per event is nearly constant, because there exist only very few partial matches at any point in time. This means that C_{eval} is linear in C_{PIO} and f simply scales its argument with an appropriate weight (ω_{PIO}). Note that for other matching strategies, f needs to be adjusted accordingly. The function g consists of two summands. The first scales its argument (C_{SIO}) with a weight constant ω_{SIO} to account for different physical layouts of primary and secondary indexes. The second, weighted with ω_{sort} , models the cost for sorting the index results. Note that we need to multiply x with a blocking factor B to obtain the number of elements from the given number of pages. Moreover, the weights must be calibrated for the actual setting (hardware, physical index layout). We discuss this calibration step in Section 3.5.3.

Given Equation 3.1, the definition of the index selection problem for a pattern

query is as follows. Select a set of index-supported BSs $\mathcal{S}_{idx}^* \subseteq \mathcal{S}_{idx}$ such that $C_{query}(\mathcal{S}_{idx}^*)$ is minimal.

$$\mathcal{S}_{idx}^* = \arg \min_{\mathcal{S}'_{idx} \in \mathbb{P}(\mathcal{S}_{idx})} C_{query}(\mathcal{S}'_{idx}).$$

Here, $\mathbb{P}(\mathcal{S}_{idx})$ denotes the power set of \mathcal{S}_{idx} . According to Equation 3.1, C_{PIO} and C_{SIO} determine C_{query} . We first assume that those costs are known when discussing our selection strategy before showing how to estimate them afterward.

Selection Strategy

A naïve index selection approach is to try out all combinations of indexes and select the one with the lowest estimated cost. For m possible indexes, 2^m combinations need to be checked, making such an exhaustive search unfeasible even for small values of m . Furthermore, neither a greedy approach that successively selects BSs from \mathcal{S}_{idx} in increasing selectivity order nor a dynamic programming approach that combines optimal solutions of sub-problems can provide an optimal index selection. We sketch the problem of the greedy strategy in the following example. Consider the query $PQ = ((A, B^*, C, D), 100, map)$ with $\mathcal{S}_{idx} = (A, C, D)$ and the corresponding selectivities $\varphi_A = 1\%$, $\varphi_C = 2\%$, $\varphi_D = 3\%$. Let us assume that symbols are independent and uniformly distributed in the event stream, and that a window holds 100 events. When selecting a single index, it is obvious that symbol A with the lowest selectivity is the optimal choice. For two indexes, a greedy strategy would select C next. However, A and C are separated by a KS, and hence C has to occur within 100 events after an occurrence of A . This will be fulfilled with high probability $(1 - (98/100)^{100} \sim 87\%)$, rendering the use of an index for C inefficient in this case. A better choice is to use symbols C and D that have to be adjacent within stream E for a match. The combined selectivity of such an adjacent pair is equal to $\varphi_C \cdot \varphi_D = 0.06\%$. Thus, this selection reduces the number of replay intervals at the expense of a slightly increased secondary index cost. In general, this is a better choice than using A and C . However, the symbols in the solutions for one and two indexes are disjoint. Thus, a greedy approach is not suitable for finding the optimal configuration.

While greedy fails in general, it succeeds in the case where no KS exists. Thus, we use this observation to produce the optimal index selection for every BB (cf. Section 3.4.2). We first group \mathcal{S}_{idx} into BBs: $BBS(\mathcal{S}_{idx}) = (BB_1, BB_2, \dots)$ with a KS between each pair of adjacent BBs. Thus, the symbols within a BB have a fixed sequential distance to each other. Assuming independent and uniformly

distributed symbols, the selectivity φ_{BB} for any $BB \in BBS(\mathcal{S}_{idx})$ is then the product of selectivities of the contained BSs:

$$\varphi_{BB} = \prod_{S \in BB} \varphi_S \quad (3.2)$$

As a consequence, for any $BB_i \in BBS(\mathcal{S}_{idx})$ with $|BB_i|$ symbols and any number of indexes $0, \dots, |BB_i|$, we can determine the optimal set of indexes in a greedy fashion, since both C_{PIO} and C_{SIO} are minimized by choosing the symbols with the lowest selectivity. This greedy selection would also be a subroutine for dynamic programming where we split up the sequence of BBs into two, solve each of the two subproblems, and combine the optimal solutions to an overall solution. In the following, we give a counterexample that this kind of dynamic programming does not return the optimal solution. Let us consider the query $PQ = ((A, t^*, B, C, t^*, D), 6, map)$, with the three BBs (A) , (B, C) , (D) and a window size of 6. Due to the window size, at most two symbols are permitted between adjacent BBs (i.e., between (A) and (B, C) and between (B, C) and (D)). Let us assume an independent and uniform distribution of the symbols with selectivities $\varphi_A = \varphi_D = 0.1$, $\varphi_B = 0.05$ and $\varphi_C = 0.25$. Our goal is to compute the optimal solution for three indexes. For the subpattern (A, t^*, BC) , the best two indexes would be the ones associated with B and C . The probability that A occurs within three events before an occurrence of B is $0.1 + 0.9 \cdot 0.1 + 0.9^2 \cdot 0.1 = 0.262$, and thus not as high as the selectivity of C . However, it is easy to see (using elementary combinatorics) that the optimal selection of three indexes for the original pattern consists of the ones associated with the symbols A, B , and D . The total selectivity for using indexes on A, B, D is 0.002615, whereas for A, B, C and B, C, D it is 0.0070125. Thus, the optimal solution for three indexes does not contain any of the two possible optimal solutions for subproblems of size two, (A, t^*, B, C) and (B, C, t^*, D) . This demonstrates that dynamic programming is not suitable for finding the optimal configuration.

However, due to the monotonicity of the functions f and g , it follows that for a given number of indexes, the optimal configuration is in the two-dimensional Pareto frontier (aka skyline [BKS01]) over C_{PIO} and C_{SIO} . As shown by Rosenman and Gero [RG83], the Pareto frontier of a given problem can be computed from Pareto frontiers of disjoint subproblems. We use this result in our approach.

Let L and R be two subproblems (i.e., disjoint subsets of $BBS(\mathcal{S}_{idx})$). Furthermore, let L_i and R_j be the Pareto frontiers for L with i indexes, and R with j indexes, $i, j \in \{0, \dots, m\}$. Then, for any $k \in \{0, \dots, m\}$, we compute the Pareto frontier $(L + R)_k$ from the cross-product of the Pareto frontiers for the correspond-

Algorithm 3.2: SelectIndexes

Input: (BB_1, \dots, BB_u) : BBs to consider
Input: m : Number of BSs with index support
Output: Pareto frontiers $[res_0, \dots, res_m]$ with res_k containing the optimal configuration for k indexes

```

1   $res \leftarrow [\emptyset_0, \dots, \emptyset_m]$ 
2  if  $u = 1$  then
3      foreach  $k \in \{0, \dots, m\}$  do
4           $res[k] \leftarrow greedySelect(k, BB_1);$ 
5  else
6       $L \leftarrow SelectIndexes((BB_1, \dots, BB_{u/2}), m);$ 
7       $R \leftarrow SelectIndexes((BB_{u/2+1}, \dots, BB_u), m);$ 
8      foreach  $k \in \{0, \dots, m\}$  do
9          foreach  $i \in \{0, \dots, k\}$  do
10              $res[k] \leftarrow res[k] \cup (L[i] \times R[k-i]);$ 
11              $res[k] \leftarrow Pareto(res[k]);$ 
12 return  $result;$ 
    
```

ing sub-problems:

$$(L + R)_k = Pareto \left(\bigcup_{i=0}^k L_i \times R_{k-i} \right)$$

Based on this, Algorithm 3.2 recursively computes the m Pareto frontiers over C_{PIO} and C_{SIO} from which we extract the optimal index configuration. The input parameters are the BBs of the pattern query and the total number of BSs with index support (m). In Line 1, we create the variable res that stores the final results (i.e., the Pareto frontiers over C_{PIO} and C_{SIO} for each $k = 0, \dots, m$). The recursion ends if the input consists of a single BB. In this case, we greedily compute the optimal solution for each k by selecting the k symbols with the lowest selectivity value (Lines 2-4). Note that $res[k]$ remains empty if the given BB contains less than k BSs with index support, which reduces the number of computations in the recursive steps. If the input consists of $u > 1$ BBs, we split them at $u/2$ and recursively compute the Pareto frontiers for the resulting sets of BBs (L, R , Lines 6,7). From those results, we assemble the solutions for u BBs as described above (Lines 8-11). The m Pareto frontiers are returned in Line 12. \mathcal{S}_{idx}^* is then the configuration with the lowest overall execution cost (C_{query} , Equation 3.1) among all results in the m Pareto frontiers.

The runtime complexity of Algorithm 3.2 depends on the number of BBs (u) and the number of BSs with index support (m). It is given by the following recurrence relation.

$$T(u) = \underbrace{2T\left(\frac{u}{2}\right)}_{\text{recursion}} + \underbrace{m}_{\text{outer loop}} \underbrace{PT\left(\underbrace{m}_{\text{inner loop}}, \underbrace{X_u^2}_{\text{cross product}}\right)}_{\text{Pareto computation}} \quad (3.3)$$

In Equation 3.3, $PT(n) = \mathcal{O}(n \log n)$ denotes the time for computing the two-dimensional Pareto frontier for an input of size n [KLP75]. Moreover, X_u is an upper bound for the size of the $2m$ Pareto frontiers obtained from the recursive steps (Lines 6,7). Due to our greedy selection at the level of BBs, it holds that $T(1) = m$ and $X_1 = 1$. Thus it is easy to see that, if the size of the Pareto frontiers is smaller than a constant c (i.e., $X_u \leq c, X_{u/2} \leq c, \dots$), the time complexity of our algorithm is $\mathcal{O}(u m^2 \log m)$. However, in the worst case, the merge of two Pareto frontiers equals the cross-product of its inputs (i.e., none of the merged entries dominates another one). Moreover, every frontier participates in at most m merges (cf. inner loop of Algorithm 3.2, Lines 9,10). This leads to the following worst-case expression for the size of X_u :

$$X_u = m \cdot X_{u/2}^2 \quad (3.4)$$

Thus, the Pareto frontiers grow exponentially ($X_1 = 1, X_2 = m, X_4 = m^3, X_8 = m^7, \dots$) in the worst case, and so does the runtime of our algorithm. However, in practice, the Pareto frontiers rarely contain more than one element because there is a strong correlation between \mathcal{C}_{PIO} and \mathcal{C}_{SIO} . As a result, in our experiments, we experienced a runtime of $\mathcal{O}(u m^2 \log m)$.

I/O Cost Estimation

The I/O costs for answering a pattern query are composed of the cost for the primary index (C_{PIO}) and the cost for the secondary indexes (C_{SIO}). C_{SIO} is a weighted sum of a linear function of the result size and an index specific term (e.g., $\theta(\log n)$ for B^+ -trees, $\theta(\log^2 n)$ for LSM trees). Because the first term is generally dominant, we assume that C_{SIO} is linear in the number of results returned from the secondary indexes. It follows that C_{SIO} is then the size of the index files times the selectivities of the range predicates associated with BSs in \mathcal{S}'_{idx} .

For a given set \mathcal{S}'_{idx} , the fraction α of the source stream E covered by the generated replay intervals determines C_{PIO} . We estimate α by approximating Algorithm 3.1

as follows. First, we construct a probability distribution \mathcal{L} over the candidate lengths. Then, we use this \mathcal{L} -distribution to compute the average number of candidates that obey the window constraint. Finally, we approximate the merge of overlapping intervals with an urn model. This leads to the following equations for α :

$$\alpha = 1 - \left(1 - \frac{\text{extend}(\mu_c)}{e_N.t - e_1.t + 1}\right)^{N_I}, \text{ with} \quad (3.5)$$

$$N_I = N \cdot \varphi_{S'_1} \cdot P(X < w), \quad X \sim \mathcal{L} \quad (3.6)$$

$$\mu_c = \mathbb{E}(X \mid X < w), \quad X \sim \mathcal{L} \quad (3.7)$$

Equation 3.7 returns the average length μ_c of all candidates that obey the window constraint (expressed as conditional expectation). The number of candidates N_I (Equation 3.6) is derived from the product of total number of events within the stream (N), the selectivity of the first selected symbol ($\varphi_{S'_1}$), and the probability that a candidate obeys the window constraint ($P(X < w)$). Equation 3.6 directly follows from Algorithm 3.1: Initially, every occurrence of S'_1 is considered a candidate ($N \cdot \varphi_{S'_1}$). After adding more indexes, the candidate length increases, and candidates that exceed the time window are filtered out ($P(X < w)$).

The function $\text{extend}(\mu_c)$ returns the replay interval length from the given candidate length μ_c . Its definition is directly derived from the function $\text{Inverval}(c, w)$ in Section 3.4.2. For example, in the base case, the replay interval length is $2(w - (c_k.t - c_1.t))$, and thus $\text{extend}(\mu_c) = 2(w - \mu_c)$. For the special cases (S1) - (S3) of $\text{Interval}(c, w)$, closed formulas for $\text{extend}(\mu_c)$ are obtained in a similar way. Based on N_i and $\text{extend}(\mu_c)$, we compute α as follows. We partition the covered time interval of E ($[e_1.t, e_N.t]$) into disjoint buckets of length $\text{extend}(\mu_c)$; for the sake of simplicity assume that $\text{extend}(\mu_c)$ is a divisor of $e_N.t - e_1.t + 1$. Then, we uniformly distribute N_I intervals over the buckets as it is known from an urn model [Fel71]. This allows us to apply the well-known formula of Cardenas [Car75] as shown in Equation 3.5.

Probability Distribution \mathcal{L} Next, we show how to construct the probability distribution \mathcal{L} over candidate lengths. Let us first make the following assumptions:

(A1) Every BS S'_i is uniformly distributed in E with probability φ_i .

(A2) The temporal distance between two adjacent events in E is constant: $e_i.t - e_{i-1}.t = \Delta t = (e_N.t - e_1.t)/N$, $i = 2, \dots, N$.

We first study a base case with two Kleene-star-separated BSs A and B with selectivities φ_A and φ_B , respectively. Then, the timespan between an occurrence of A and the next occurrence of B determines a candidate's temporal extent. **(A1)** and **(A2)** allow us to model the arrival of symbols as a Poisson process. Hence, the inter-arrival time between two occurrences of B is exponentially distributed with parameter $\lambda_B = \varphi_B \cdot \Delta t^{-1}$ [Bha08]. Moreover, due to the memorylessness property² of the exponential distribution, the probability that B occurs within w time units after an occurrence of A equals $P(X < w)$, $X \sim \text{Exp}(\lambda_B)$. Hence, in this case $\mathcal{L} = \text{Exp}(\lambda_B)$.

Next, we consider the case of multiple BSs. Let $\mathcal{S}'_{idx} = (S'_1, \dots, S'_k)$ be a sequence of index-supported BSs in the order they appear in a pattern, and $\varphi_1, \dots, \varphi_k$ the corresponding selectivities. Furthermore, assume that a KS in PQ is added between a pair of adjacent BSs. Then, the length of a candidate is the timespan between an occurrence of S'_1 and the first occurrence of S'_k **after** all remaining symbols (S'_2, \dots, S'_{k-1}) occurred in proper order. Under assumptions **(A1)** and **(A2)**, the candidate length is given by a compound random variable $Z = X_2 + \dots + X_k$, with $X_i \sim \text{Exp}(\lambda_i = \varphi_i \cdot \Delta t^{-1})$, $2 \leq i \leq k$. The distribution of Z is called hypoexponential [Bol+06] with parameters $\lambda_2, \dots, \lambda_k$. Hence, it follows that $\mathcal{L} = \text{Hypo}_{\lambda_2, \dots, \lambda_k}$. In order to compute μ_c (Equation 3.6) and N_I (Equation 3.7), we use the closed forms for the probability density function and the cumulative distribution for *Hypo*, see [Bha08] for details.

Finally, we consider the general case where blocks of more than one BS exist. In other words, we consider a sequence of Kleene-star separated BBs. For every BB, the selectivity is the product of the corresponding symbol selectivities (see Equation 3.2). Thus, we compute arrival rates for every BB and apply the hypoexponential distribution for our estimation on the granularity of BBs.

Adaption to Non-Uniform Data So far, we assumed a uniform symbol distribution **(A1)** and a constant Δt **(A2)** in our estimation of C_{PIO} . However, these assumptions may not hold in real-world applications because streams generally follow trends and periodic behaviors.

In order to address this problem, we utilize *ChronicleDB*'s lightweight indexing. Recap that every index entry of the *TAB⁺-tree* stores the event count, the covered

²For any exponentially distributed random variable X and two values t, x it holds: $P(X < x + t | X \geq t) = P(X < x)$.

period, and the minimum and maximum values per attribute for a contiguous subsequence of the stream (cf. Section 3.2). Thus, a certain level of the TAB^+ -tree, can be viewed as a temporal histogram of the stream, with the index entries being the histogram buckets. Such a histogram is essentially a coarse representation of the stream that we use to detect the substreams required to answer a given query and to improve our cost model's accuracy. Let $e.a_j \in [low, high]$ be the range condition of a BS $S \in \mathcal{S}_{idx}$, and $[\mathcal{A}_{min}^j, \mathcal{A}_{max}^j]$ the min/max range of attribute \mathcal{A}^j for a given histogram bucket. If the intersection of those ranges is empty, S does not occur within the substream covered by the bucket. We compute this information for every $S \in \mathcal{S}_{idx}$ and every histogram bucket. Then, we perform pattern matching over histogram buckets. In analogy to pattern matching on our original stream, we move a window of size w over the histogram buckets in temporal order to filter out substreams (and histogram buckets) without a match. Thus, it is sufficient to check the remaining substreams associated with histogram buckets not excluded in the filter step. We term these remaining buckets coarse pattern map (CPM).

In addition to pruning irrelevant substreams, CPM serves to improve the quality of our cost model as follows. Instead of assuming a uniform temporal symbol distribution (**A1**) and a fixed Δt (**A2**) for the entire stream, we only require that these assumptions hold within a substream (bucket). For such a substream, we compute the symbol selectivity and Δt , and use them in our cost model. For instance, assume a stream with $N = 10^6$, and a symbol S with $\varphi_S = 0.1\%$. In this setting, S will occur 1000 times within the stream. If we find that only 2 out of $M = 1000$ substreams may contain S , we can estimate that S occurs every second event in those substreams. Hence, we adjust its arrival rate to $2\Delta t$. Similarly, Δt is adjusted by dividing the covered period by the number of contained events for every remaining substream. Clearly, the quality of the results improves with the resolution of the histogram (i.e., M). However, with increasing M , the costs also increase. We will discuss this trade-off in Section 3.5.3.

3.4.4 Extensions

In the following, we discuss how to overcome the limitations introduced at the beginning of this section. First, we show how to apply our approach to matching strategies other than *contiguous*. Then, we discuss the handling of advanced features of state-of-the-art CEP languages and complex symbol predicates. Finally, we show how to further shorten the replay intervals by exploiting knowledge about the minimum and maximum temporal distance between adjacent events.

Original	Semantics	Rewrite
$\neg P$	Negation	<i>true</i>
$P_1 P_2$	Alternative: P_1 or P_2	<i>true</i>
P^+	1 or more occurrences of P	$(P, true^*)$
$P?$	0 or 1 occurrence of P	<i>true</i> [*]
$P\{n, m\}$	Between n and m occurrences of P	$(\underbrace{P, \dots, P}_{n \text{ times}}, true^*)$

Table 3.2: Pattern rewrite rules.

Matching Strategies

The difference between *contiguous* and the other strategies is that only *contiguous* defines tight sequential distance constraints between adjacent BSs. In contrast, *skip-till-next* and *skip-till-any* allow to skip irrelevant events between BSs. Our approach already deals with variable distances between BBs. Hence, to support those matching strategies, it is sufficient to insert a *true*^{*} symbol between every pair of adjacent BSs, because this way, we map every BS to a dedicated basic block.

Complex Patterns

The basic idea is to rewrite more complex patterns to match our definition and cover all matches of the original query.

As a first step, we modify the symbol predicates to only consist of single-attribute range conditions. Therefore, we inspect every atomic predicate of every symbol. If the predicate is a supported range condition, it remains unchanged. In all other cases, we replace it with the constant value *true*. After simplifying the resulting boolean expressions, each symbol is either associated with the constant value *true* or a sequence of range conditions connected via boolean operators (AND, OR), which our approach can handle. Obviously, the results fulfilling the modified predicates are a superset of those fulfilling the original condition. Hence, these modifications do not affect the correctness of our approach.

The second step applies a set of rewrite rules to the original pattern in order to eliminate unsupported quantifiers and nested sub-patterns. Therefore, we recursively apply the rewrite rules sketched in Table 3.2 in a top-down manner. The recursion ends if the pattern only consists of BSs and *true*^{*} symbols. Note that

the simplifications to true^* for KSs does not affect our approach, since we do not consider their predicates anyway (cf. Section 3.4.1).

Negations and alternatives have no direct support in our approach. Depending on whether P is a single symbol or a complex pattern, we map those quantifiers to true or true^* respectively. For the remaining quantifiers, our mapping preserves the lower but dismisses the upper bound for the number of occurrences. It is easy to see that the results of the rewritten pattern contain all matches of the original one. Hence, the rewrites do not affect the correctness of the results.

Example. Consider the pattern query $PQ = ((A, (B, C^+)^+, D), w, \text{map})$. The first step is to resolve the Kleene-plus on the nested sub-pattern $(B, C^+)^+$. This results in the following pattern: $(A, B, C^+, \text{true}^*, D)$. Then, we resolve C^+ and receive $(A, B, C, \text{true}^*, \text{true}^*, D)$. Finally, the adjacent true^* symbols are consolidated and we obtain a query that matches our definition.

Replay Interval Shortening

Suppose the system provides the minimum and maximum temporal distance between two adjacent events (denoted as Δt^- and Δt^+ , respectively). In that case, we can further shorten the length of the replay intervals as generated in Section 3.4.2.

First, in the **base case** of $\text{Interval}(c, w)$, we shift both interval bounds depending on how many BSs must occur before/after the candidate c . Therefore, we assumed a minimum temporal distance of 1. For instance, if $S'_1 = S_i$, we could shrink the right interval bound by i time units. With knowledge of the minimum temporal distance, we can extend this to $i \cdot \Delta t^-$ units since we know that the symbols S_1, \dots, S_{i-1} span at least this amount of time.

Second, knowledge of Δt^+ allows us to apply the three special cases (S1) - (S3) to a broader range of configurations. Recap that these cases required the first and/or last symbol to be present in the index results. In contrast, with knowledge of Δt^+ , it is sufficient to have a member of the first and/or last BB in the index results. For instance, consider the case $i_1 = 1, i_k = n - 2$. For a given candidate, the generated interval would be $[c_1.t, c_1.t + w]$. However, if we know that i_k belongs to the last BB, we can create a shorter interval: $[c_1.t, c_k.t + 2\Delta t^+]$, since we know that a valid match ends within two events from c_k .

3.5 Experimental Evaluation

This section reports the most important findings from the extensive experimental evaluation of our approaches. We first describe the system setup used throughout all experiments. Then, we present the results for each of the operators.

3.5.1 System Setup

We use a workstation equipped with an AMD Ryzen5 2400G CPU@3.4GHz (4 cores, 8 threads), 32GB of memory, and a Samsung SSD 970 EVO for the primary index and secondary indexes. The system runs on a Debian Linux (sid, kernel version 5.7). We implement our approaches in Java as an extension of *ChronicleDB*, and use Oracle's Java Virtual Machine (JVM) 14.0.2 for execution.

For every data set the TAB^+ -tree was bulk-loaded with a fill-factor of 90%. The secondary indexes are LSM-trees using a single data file per level. Both indexes utilize 8 KiB pages.

3.5.2 Windowed Aggregation

We compare the processing time of both index-based approaches (**Optimized**, **Naive**) and a pure **Replay**-based solution. Therefore, we create two synthetic event streams with random data.

Data Sets & Queries

The streams cover one week of data with a frequency of 100 events/second, resulting in 60,480,000 events. The first stream consists of 16 byte-sized events (a single double-precision floating-point attribute and the timestamp), which leads to a fan-out of 152 for the inner TAB^+ -tree nodes and 458 events per leaf node. The events of the second stream are 80 bytes in size (9 double-precision floating-point attributes and the timestamp), resulting in a fan-out of 23 and 90 events per leaf. The disk sizes of both streams are 555 MB and 2.9 GB, respectively.

We compute sum, average, and count aggregates for a single attribute and a window duration of one day. We vary the slide parameter from 10ms (1 event per slide)

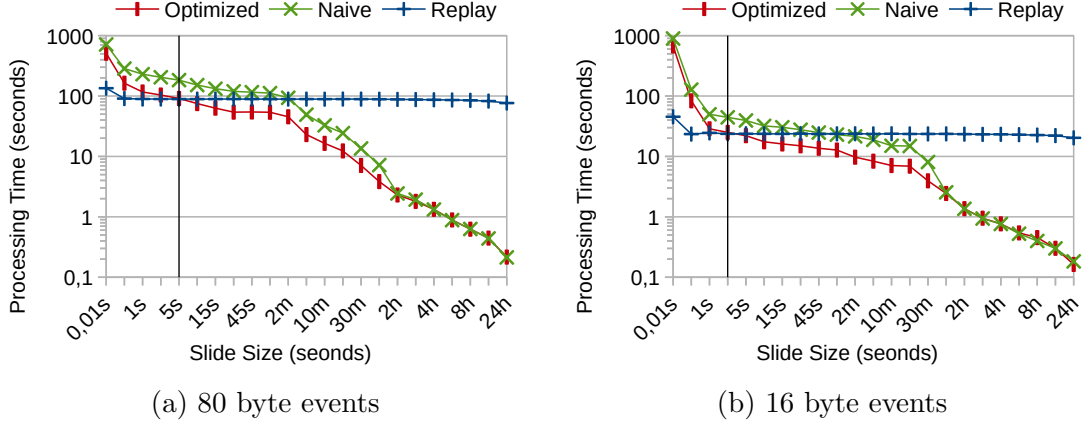


Figure 3.8: Processing time of sliding time window aggregation for varying values of slide using different event sizes.

to 1 day (whole window) and measure the execution time for all approaches. Moreover, we limit *ChronicleDB*'s page cache (LRU) to 50 pages to highlight the impact of the disk access pattern induced by each method.

Note that the performance of the replay-based variant solely depends on the number and size of events to read. Similarly, the performance of the index-based variants depends on the height and the fan-out of the *TAB⁺-tree*, which in turn depends on the number of materialized aggregates (i.e., the number of attributes). Thus, we omit experiments with varying aggregate functions and data distributions.

Results

Figure 3.8 shows the results for executing the operator over both streams. As expected, the execution time of **Replay** is nearly constant since it always reads the entire event stream. Both index-based approaches are significantly slower for very small *slide* values because the *TAB⁺-tree* is traversed for every output event. However, **Optimized** outperforms **Replay** for slide values greater than 5 seconds (80-byte events) and 2 seconds (16-byte events) because the temporal aggregation query performs fewer merges per slide. **Naive** needs much larger slide values to outperform **Replay**: 120 seconds (80 byte events) and 30 seconds (16 byte events). For slide values of 300 seconds (5 min, 80-byte events) and 900 seconds (15 min, 16-byte events), the processing time for both index-based solutions drops faster than for smaller slide values. The reason is that at this point, the disk actually begins to skip pages instead of merging scattered random reads into one large

sequential read. The vertical black line shows the break-even point estimated by our cost model and clearly confirms its validity.

3.5.3 Sequential Pattern Matching

We evaluate our approach in a broad set of use-cases on both synthetic and real-world data. Before discussing the results of our experiments, we first describe the methods and systems under evaluation, the data sets, and explain the structure of the queries.

In addition to the method **Replay** that replays the entire stream, we examine our method for computing replay intervals (Algorithm 3.1) and the following four index selection strategies. **All** uses all available secondary indexes. **Greedy** selects the k least selective symbols for $k = 1, \dots, m$ and chooses from these m options the one with the lowest expected cost according to our cost model. **Optimal** performs index selection as described in Section 3.4.3. We also implement **Budget**, which is the index selection strategy proposed in [Sei+19].

Our testbed uses the in-memory sort algorithm from the Java standard library (a quick-sort variant) and a pattern implementation based on a finite state automaton [DIG07]. For the monotonic functions f and g as presented in Section 3.4.3, we set the three weights as follows after a series of calibration experiments: $\omega_{PIO} = 1.13$, $\omega_{SIO} = 2.53$ and $\omega_{sort} = 12.73$.

For every data set, we conduct two types of experiments. First, we compare query runtimes of our approach with Apache Flink [Car+15] (**Flink**) and a commercial relational database system with support for sequential pattern matching (**DBMS**). Both systems support pattern matching via the `MATCH_RECOGNIZE` (MR)-clause and process those queries via a full stream replay³. To highlight the versatility of our approach, we also integrate our prefiltering technique into those systems. Second, we discuss selected aspects of our approach using our Java-based implementation.

We run Flink in a single node configuration because our queries do not use key-based partitioning. In those cases, Flink shifts all load to a single node anyway [21]. Thus, a single node setup is preferable over a cluster setup since no data is transferred over the network. We connect our primary index implementation to Flink and use it as the data source since this combination was superior to any other data source. For index-based execution, we prefilter the stream at

³The `MATCH_RECOGNIZE` clause for pattern matching was recently added to the SQL standard [16].

the data source and only replay the events falling into the generated replay intervals.

For the DBMS, we store the event streams in an index-organized table on the timestamp attribute and create secondary indexes for each attribute used in the respective queries. The queries use the MR-clause. For index-based execution, we compute the replay intervals using our Java framework/index implementation and execute a MR query for each of the generated intervals. In all comparisons with **Flink** and **DBMS**, we use **Optimal** as index selection strategy.

Synthetic Data The synthetic event stream contains 50 million (50M) events, each 128 bytes in size. The timestamps of the events are $e_i.t = i$, $i = 1, \dots, 50M$. Moreover, the events contain five double-precision floating-point attributes $\mathcal{A}_1, \dots, \mathcal{A}_5$ with values from the unit interval $[0, 1)$. Furthermore, we build an index for each of the attributes. The attributes values follow normal distributions given by

$$e_i.\mathcal{A}_j \sim \mathcal{N}\left(\frac{i}{50M}, \sigma_{\mathcal{A}_j}^2\right), \text{ for } j \in \{1, \dots, 5\}.$$

Note that the i -th event has a mean $i/50M$ for all attributes, i.e., it is a linear function in i . The variance $\sigma_{\mathcal{A}_j}^2$ of the normal distributions only depend on \mathcal{A}_j , $1 \leq j \leq 5$. A setting of the variance allows us to control the temporal correlation of the underlying distributions. We use the following settings in our experiments: $\sigma_{\mathcal{A}_1} = 0$, $\sigma_{\mathcal{A}_2} = 0.001$, $\sigma_{\mathcal{A}_3} = 0.01$, $\sigma_{\mathcal{A}_4} = 0.1$, and $\sigma_{\mathcal{A}_5} = 1$. For example, the distribution of \mathcal{A}_1 returns values varying linear with i , while for \mathcal{A}_5 we obtain a uniform distribution because values outside of the unit interval $[0, 1)$ are not considered.

Pattern Queries Pattern queries consist of m BSs S_1, \dots, S_m and a varying number of KSs. In order to generate BSs, we assign one of the attributes ($\mathcal{A}_1, \dots, \mathcal{A}_5$) to a basic symbol S_i , $i \in \{1, \dots, m\}$, which determines the distribution of the BSs. Then, we create the corresponding range condition. The query range randomly covers between 0% and 10% of the domain, and its position is also randomly generated. In order to insert KSs⁴, we randomly pick a number $x \in \{1, \dots, m-1\}$ and insert a KS between the BSs S_x and S_{x+1} . Then we proceed recursively on the index set $\{x+1, \dots, m-1\}$ until the index set is empty. This results in $\log_2(m)$ KSs on average and BBs of varying length. The window for all queries is set to 300 time units.

⁴Recap, that we treat KSs as variable length sequences. Hence, an always-true predicate is sufficient for our purpose.

Real-World Data We examine two real-world event streams in our evaluation. The first stream contains crimes occurring in the city of Chicago from January 2001 to June 2020⁵. It consists of about 7M events. Every event represents a reported crime with 22 attributes (130 bytes in size). In our experiments, we create a secondary index on four attributes: The primary category of the crime (e.g., assault, battery), latitude and longitude reflecting the location of the crime, and beat. A beat is the smallest unit in terms of geographic region used by the police.

The second data set contains real-world trajectory events from the OpenSky Network [Sch+14], an open infrastructure for collecting flight data. The examined event stream consists of 50M events representing the movement of a single aircraft over three years. Every event consists of 17 attributes and occupies a total of 155 bytes. We use three attributes (velocity, altitude, vertical speed) to define the pattern symbols, each equipped with a secondary index.

Results for Synthetic Data

The first results of our experiments address various aspects (query runtime, cost estimation, selection algorithm) of our approach under ideal conditions for the cost model, i.e., a uniform data distribution. Therefore, the range conditions of all BSs refer to attribute \mathcal{A}_5 . We vary the amount of BSs (m), and executed 100 generated queries for $m = 2, 4, 8, 16, 32$.

The runtime comparison in Figure 3.9 (a) reveals that for the replay-based variants (**Replay**, **Flink**, **DBMS**), the query complexity has almost no impact, which supports our claim that the queries are I/O bound. Even though they use the same storage backend, the runtime of **Replay** is superior to **Flink**. The reason is that running a full-fledged system induces a considerable overhead compared to our lightweight implementation. However, both methods benefit significantly from indexes (**Index**, **Flink Index**), which reduce the runtime up to an order of magnitude. We observe the same behavior for **DBMS** and **DBMS Index**, even though executing a MR query for each of the replay intervals introduces additional overhead.

Next, we compare different index selection strategies using our Java-based implementation. Therefore, we execute the queries using the four index selection strategies **All**, **Greedy**, **Optimal**, **Budget**. Figure 3.9 (b) shows the average speedup compared to **Replay** for every strategy as a function of m . Clearly, **All**

⁵Available at <https://data.cityofchicago.org/Public-Safety/Crimes-2001-to-Present/ijzp-q8t2>

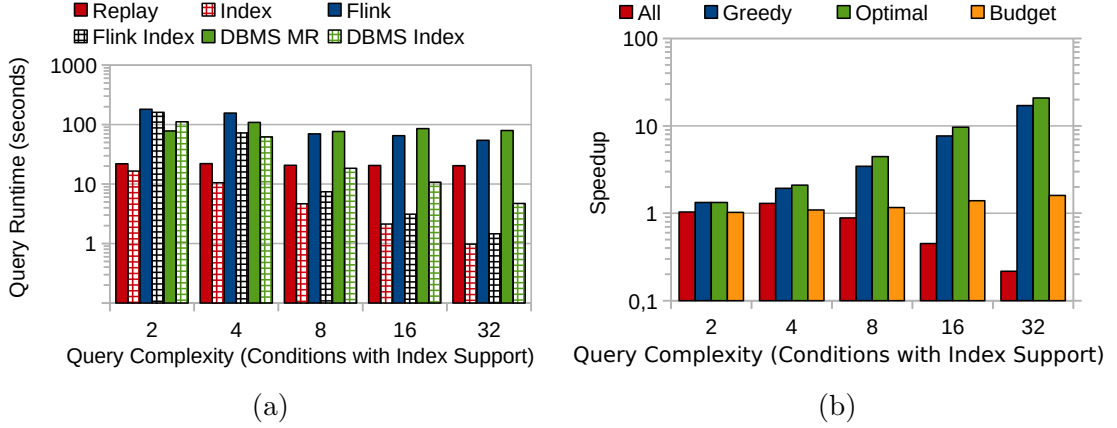


Figure 3.9: Query runtimes (a) and speedup of different index selection strategies compared to replay (b) as a function of the query complexity.

quickly leads to severe performance degradation (5x slower than **Replay**). **Budget** performs only slightly better than **Replay** on average (up to 5x in the best case). The reason is that if none of the symbols is highly selective ($< 1/w$), the algorithm stops after the first index. The low low-selectivity query fails to filter out any primary index data (i.e., the replay intervals cover the entire stream), and thus the algorithm gets stuck in a local optimum. In contrast, **Greedy** and **Optimal** are able to boost query performance in all cases, and achieve up to 20x speedup compared to **Replay**. On average, **Optimal** provides a 20% performance improvement over **Greedy**. In some test cases, the improvements reached a factor of 2.

Next, for each of the queries we consider the relative cost estimation error Err_{PIO} that is determined by the actual and estimated primary index I/O (C_{PIO}^{actual} , $C_{PIO}^{estimated}$) as follows.

$$Err_{PIO} = | C_{PIO}^{actual} - C_{PIO}^{estimated} | / C_{PIO}^{actual}.$$

Figure 3.10 shows the results of this experiment as a boxplot. The boxes span the 25th to 75th percentiles; the orange line indicates the median, and the whiskers are the 5th and 95th percentile of Err_{PIO} . A great majority of the errors are beneath 5%, which clearly confirms the accuracy of our cost model. However, for $m = 32$, the error slightly increases. In this case, the secondary indexes prune most of the primary data, and thus there is a small absolute error but a high relative one.

In addition to the query runtime and estimation error, we also study the performance overhead of our selection strategy **Optimal** in terms of runtime and number

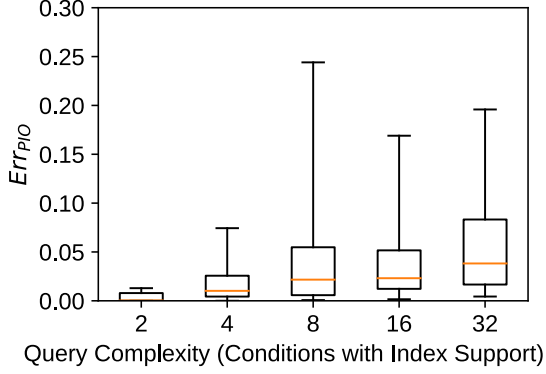


Figure 3.10: Err_{PIO} as a function of the query complexity.

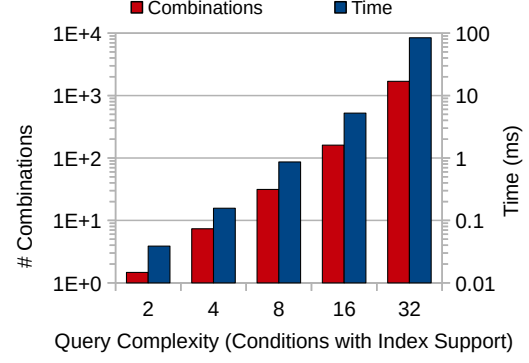


Figure 3.11: Average index selection time and number of examined index combinations for **Optimal** as a function of the query complexity.

of examined index combinations. Figure 3.11 shows that the number of examined combinations is much smaller than for the naïve variant. For instance, our strategy checks only 1689 out of 2^{32} combinations on average for $m = 32$. Also, the average runtime of 70 ms ($m = 32$) is negligible compared to the runtime of a full replay (~ 20 seconds). Note that the runtime of selection strategies **Greedy** and **All** are only a few nanoseconds independent of m , because the number of examined combinations is very low (one for **All** and m for **Greedy**).

In order to give more insight into the accuracy of our cost model, we present additional results for the estimation of C_{PIO} on a single query with $m = 8$. For the 256 possible execution plans of the query, Figure 3.12 (a) shows the relative estimation error for each plan (ranked with respect to the estimated cost). With an average Err_{PIO} value of less than 5% and only 5 values above 30%, it confirms the accuracy of our C_{PIO} estimation. Those outliers correspond to small absolute errors (< 3 MiB) that translate into high relative errors. Figure 3.12 (b) shows the measured processing time as well as the estimated overall query cost C_{query} (see Equation 3.1) for each of the 256 execution plans. The results are ranked on the x-axis according to C_{query} . In summary, the small differences between processing time and estimated cost confirm the accuracy of our cost model.

Results for Non-Uniform Data In the following, we present the results for query runtime and cost estimation accuracy for the case of non-uniform data distributions. For $m = 8$, we assign one of the BSs to a selected attribute from $\mathcal{A}_1, \dots, \mathcal{A}_5$, while the remaining $m - 1$ symbols still refer to \mathcal{A}_5 . Again, we run 100 randomly

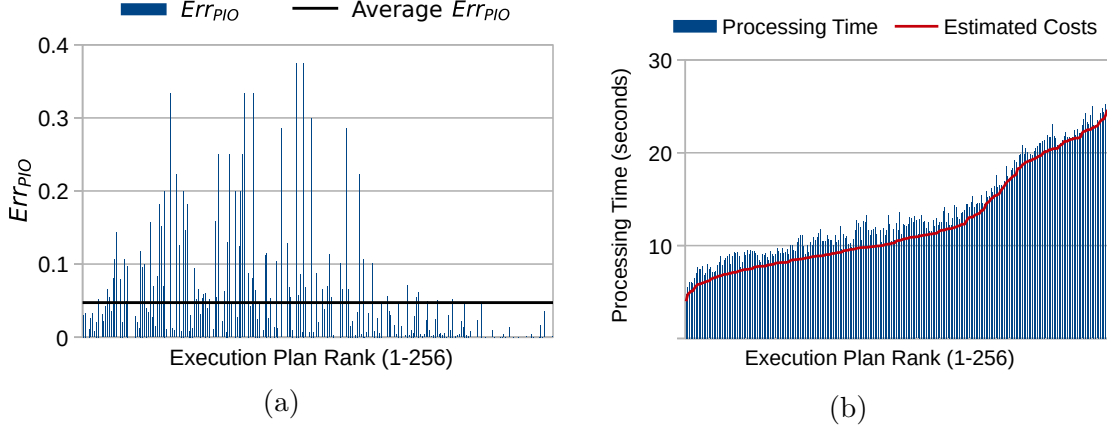


Figure 3.12: Err_{PIO} (a), and query processing time (b) for all execution plans of a single query with $m = 8$.

generated queries for every setting. The results of the runtime comparison are shown in Figure 3.13 (a). The replay-based variants are insensitive to the data distribution because the queries are I/O bound. Similar to the uniform case, indexes improve the runtime in all tested cases. However, for highly skewed data, the performance gain is more prominent because the CPM allows excluding even more data from the replay.

We discuss the impact of CPM using six index selection strategies. Four of them (**All**, **Greedy**, **Optimal**, **Budget**) assume a uniform data distribution, while two additional strategies (**Greedy***, **Optimal***) utilize CPM to estimate primary index cost. CPM is computed from a histogram with 531 buckets.

Figure 3.13 (b) shows the average speedup of the selection strategies compared to **Replay** for every selected attribute $\mathcal{A}_1, \dots, \mathcal{A}_5$. **Budget** faces the same problems as for uniform data and performs only slightly better than **Replay** on average. **Greedy** and **Optimal**, which do not utilize CPM, perform similar independent of the data distribution. Their performance is slightly better for attributes with small σ -values (and hence higher temporal correlation), because results are clustered within a small temporal region of the stream. Hence, the actual primary I/O cost is smaller for those distributions. In contrast to their counterparts without CPM support, **Greedy*** and **Optimal*** provide a more accurate cost model, and thus a better selection of secondary indexes. This results in considerable runtime improvements compared to **Greedy** and **Optimal**, respectively. The positive effect of CPM vanishes as data approaches a uniform distribution. Similar to the experiments on uniform data, **Optimal** and **Optimal*** achieve an average improvement of about 20% compared to **Greedy** and **Greedy***, respec-

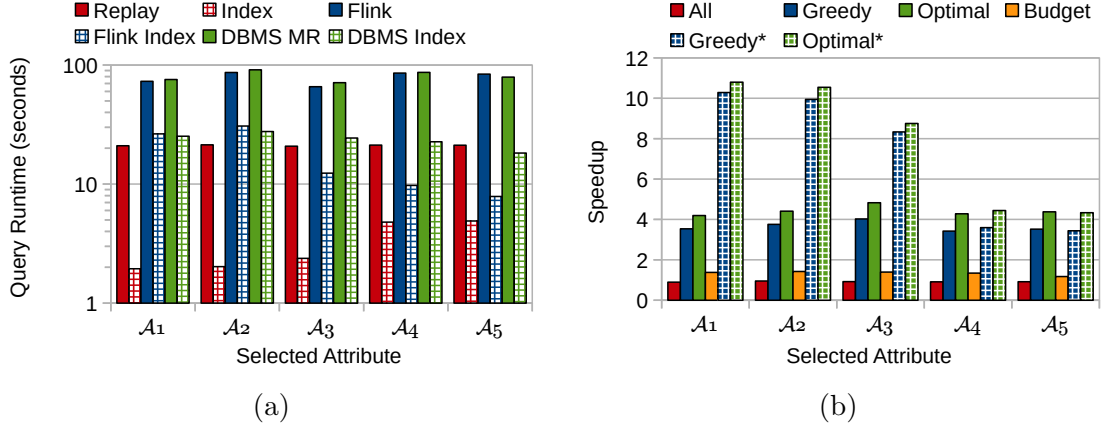


Figure 3.13: Query runtimes (a) and speedup of different index selection strategies compared to replay (b) for varying data distributions.

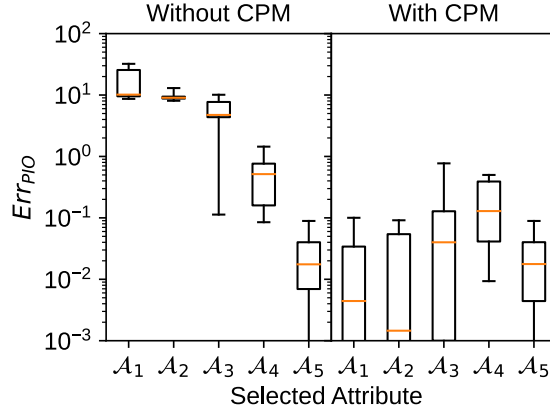


Figure 3.14: Err_{PIO} for varying data distributions without (left) and with (right) utilizing CPM.

tively.

Figure 3.14 shows Err_{PIO} for each selected attribute (using a log-scale). On the left-hand side, we plot the results without CPM, while on the right-hand side we use CPM for cost estimation. For attributes $\mathcal{A}_1 - \mathcal{A}_3$ the estimation without CPM leads to an extremely high overestimation. As expected, Err_{PIO} decreases for data that are more uniform. In contrast, the utilization of a CPM keeps the average estimation error low in all cases.

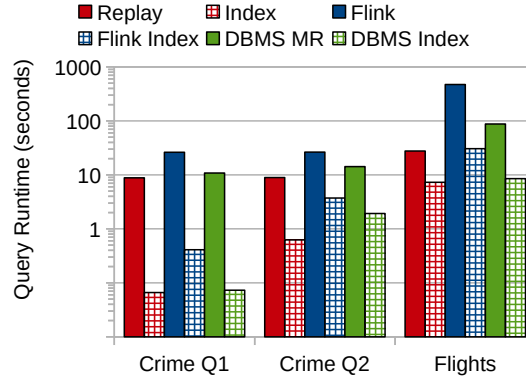


Figure 3.15: Query runtimes for real-world data sets.

```

1 SELECT * FROM CRIMES MATCH_RECOGNIZE (
2   MEASURES A.ID, B.ID, C.ID
3   PATTERN A T* B T* C WITHIN 30 MINUTES
4   DEFINE
5     A AS A.PRIMARY = 'ROBBERY' AND A.BEAT = 2232,
6   -- Variant 2: X1 <= A.LAT <= X2 AND Y1 <= A.LON <= Y2,
7     B AS B.PRIMARY = 'BATTERY' AND B.BEAT = A.BEAT,
8   -- Variant 2: X1 <= B.LAT <= X2 AND Y1 <= B.LON <= Y2,
9     C AS C.PRIMARY = 'VEHICLE THEFT' AND C.BEAT = A.BEAT
10  -- Variant 2: X1 <= C.LAT <= X2 AND Y1 <= C.LON <= Y2,
11 )

```

Listing 3.1: Possibly related crimes query; variant 2 replaces beat condition with a bounding box on lat/lon coordinates.

Results for Real-World Data

In this section, we analyze the accuracy of our cost model and the resulting query runtimes for both real-world data sets. Figure 3.15 shows that independent of the system, dataset, and query, the use of indexes significantly boosts query runtime. For Crime Q1, the improvement reaches two orders of magnitude, while for the more complex Q2, we achieve one order of magnitude. As expected, the performance of the replay-based variants does not vary with the queries. For the flight data, the improvements are similar to Crime Q2. Those results confirm that our approach gracefully handles complex data distributions, as found in real-world data sets.

In the following, we describe the queries for each data set and discuss the accuracy of our cost model in detail.

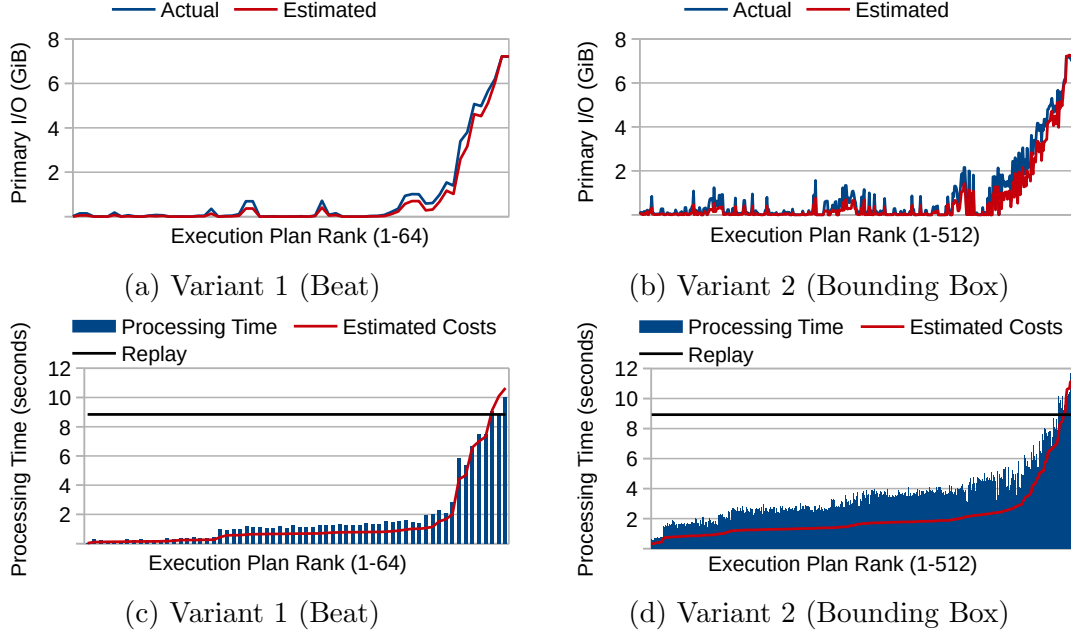


Figure 3.16: Estimated and actual primary I/O (a,b) and processing time with estimated total cost (c,d) for all execution plans of both crime query variants.

Chicago Crime Data. Based on the crime data, we first design a pattern matching query to detect crimes that are probably related. In particular, we search for a sequence of robbery, battery, and motor vehicle theft within a 30 minute time window and close spatial proximity. Every crime in this sequence possibly refers to the same suspect, who robs one person, hurts another person, and finally steals a car to escape. The full query is shown in Listing 3.1 using the `MATCH_RECOGNIZE` syntax. We express each situation in this sequence via a BS on the primary category attribute, pairwise separated with an always true KS (T^*) to allow for other reported crimes in between. The pattern query comes in two variants. The first requires each of the crimes to occur within a specific beat. The second constrains the region via a bounding box on the lat/lon coordinates. Thus, we receive a maximum of 6 and 9 possible secondary indexes for the first and the second variant, respectively. For both queries, we perform every possible execution plan and measure the runtime as well as the actual primary index I/O.

Figures 3.16 (a) and 3.16 (b) show the estimated and actual primary I/O cost for both query variants. Even though our approach tends to underestimate the cost in both cases, it clearly captures the trend. The underestimation is a result of temporal clustering within the source data. The crime cases are not temporally

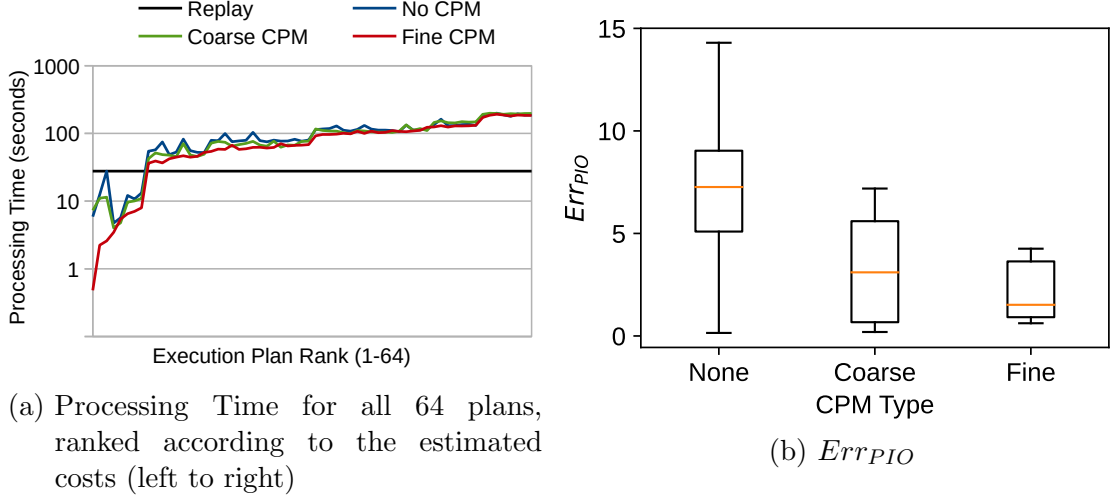


Figure 3.17: Processing Time (a) and Err_{PIO} (b) with varying CPM granularity for the landing pattern.

uniformly distributed but occur more frequently within specific times of the day (i.e., at night). The generated replay intervals are within such a bursty period, and thus we read more pages for a 30-minute window than expected. Note that CPM does not improve the accuracy of the cost estimation in this case because every substream contains at least one crime for the selected primary categories and regions. Hence, no substreams are excluded, and the parameters of the cost model are not adjusted. However, Figure 3.16 (c) and Figure 3.16 (d) show that our cost model is still able to determine a good ranking of the execution plans, and thus selects an efficient plan for processing the pattern query (best for Variant 1, 4th best for variant 2).

Open Sky Data Data. Given the trajectory of an airplane, we use pattern matching queries to detect landing maneuvers. The pattern identifies a gradual descent of an aircraft expressed as a continuous decrease in altitude and velocity. In total, the query contains six thresholds on attributes with secondary indexes resulting in $2^6 = 64$ possible query execution plans. The challenge of this event stream is that all the queried attributes are highly correlated and non-uniformly distributed. We perform all 64 execution plans three times, using (i) no CPM, a coarse-grained CPM (658 histogram buckets), and a very fine-grained CPM (1.1M histogram buckets) to adjust the parameters of the cost-model.

Figure 3.17 (a) shows the processing time for every execution plan ranked by the cost-model (left to right). Many of the plans incur a processing time far worse

than a simple replay. The reason is as follows. We require a starting symbol stating that the velocity and altitude values are above a certain threshold. Otherwise, every event indicating a decrease in altitude would create a new partial match (PM), which in turn leads to multiple matches for every landing phase. However, around 90% of all events satisfy the starting symbol's condition since they belong to the aircraft's cruise phase. Thus, any plan containing one of those conditions almost reads the entire secondary index, which exceeds the cost of a full replay due to the very high sorting cost. Nevertheless, each method selects a plan that is at least 3x faster than a pure replay. Only the method with a fine-grained CPM was able to select the optimal execution plan. Due to the high resolution, the selected plan did not make use of secondary indexes at all but directly replayed the valid regions of the CPM. This way, a sub-second response time was achieved. However, due to the massively increased cost for computing the CPM (5 ms coarse vs. 6.4 s fine), this advantage entirely disappears if we build CPM for every query.

Finally, Figure 3.17 (b) shows the relative error for the three variants. It reveals that in cases of highly correlated events, even a fine-grained CPM leads to cost estimation errors up to a factor of 5. Thus, estimation methods that are more accurate for these kinds of events are an interesting opportunity for future work.

3.6 Summary

In this chapter, we presented index-based methods to accelerate the processing of windowed aggregation and sequential pattern matching over persistent event streams. As a baseline for comparisons, we first presented a generic replay operator that reads the whole event stream from non-volatile storage and replays it into appropriate online algorithms.

Then, we detailed how to efficiently utilize *ChronicleDB*'s lightweight indexes to answer windowed aggregation queries. The presented method can handle time and count windows with arbitrary slide parameters. Since a replay is more efficient for small *slides*, we also developed a cost model that reliably chooses the best method for a given query.

Finally, we presented a comprehensive method to accelerate pattern matching queries via off-the-shelf secondary indexes, like LSM-trees or B⁺-trees. Based on this method, we developed a cost model that predicts the query execution costs for

an arbitrary index configuration and a selection algorithm that efficiently computes the optimal set of indexes for a given query.

We experimentally validated all presented approaches using a great variety of data sets and queries.

4

iGPU-Accelerated Online Processing

In order to offer both, high throughput and low latency, modern online SPEs like Apache Flink [Car+15] process event streams in parallel by scaling out the computation in a cluster environment (cf. Section 1.3). Unrelated streams can be handled independently by multiple computing units in parallel. Similarly, certain operations, such as filtering out irrelevant data items based on thresholds, offer opportunities for data-parallel processing. However, recent research [Zeu+19] has shown that orthogonal to improvements through *scaling out* processing over the network, there is still a huge untapped potential in *scaling up* a single computing resource. Especially by using recent advancements in modern hardware for streaming applications.

Among the most promising developments for *scaling up* a single node is the wide variety of GPUs. However, when dealing with latency-sensitive stream applications, the transfer time from main memory to the GPU is an ever-prominent limiting factor. In general, there are two ways to circumvent these limitations: (1) masking latency through software via scheduling algorithms [Kol+16] and (2) reducing the latency through hardware.

Modern hardware using recent advancements in the Heterogeneous System Architecture (HSA), a platform specification for heterogeneous computing, have made (2) more feasible [Hwu15]. In particular, the shared-memory approach of iGPUs seems like a natural fit for accelerating low latency stream processing applications since it avoids costly data transfers to and from the very size-limited GPU memory. In addition, the advent of so-called signals, a lightweight inter-kernel communication mechanism, further increases the potential for low latency processing in HSA.

In this chapter, we demonstrate that HSA-enabled iGPUs significantly improve the processing performance of online SPEs. Therefore, we design a prototypical HSA-enabled online SPE featuring all four major event stream operations (filter, windowed aggregation, windowed join, sequential pattern matching). Even though

current iGPU products target commodity hardware, those machines show promising results already. Furthermore, the presented techniques lay the groundwork for similar developments in the high-end server market.

4.1 Related Work

Using HSA for stream processing has not been widely researched. Due to the utilization of SIMD hardware characteristics, this work also shares similarities with previous research on iGPUs, dGPUs and FPGAs.

iGPU and HSA. Most similar to our work is the HELLS-Join [Kar+13], a join operator for windowed data streams. The authors use an early iGPU architecture called AMD Fusion Trinity and develop a join algorithm consisting of three phases: (P1) tuple comparison, (P2) producing join results, and (P3) updating windows. The algorithm is designed around outdated dedicated separate main memory areas for GPU and CPU, which makes inter-memory access more costly. As a result, only (P1) was handled by the GPU. Instead, we use recent HSA advancements to reduce overhead and process all three join stages on the iGPU. Furthermore, we provide HSA-enabled implementations for filtering, windowed aggregation, and sequential pattern matching and integrate them into a processing framework. This integration allows us to schedule complex queries across all available processing units. The CellJoin [GBY09] also focuses on windowed stream join algorithms for the cell processor, which is similar to iGPUs as it uses several specialized SIMD processing units with a more conventional processor on one die. However, separate memory areas between both processing units result in similar limitations for the HELLS-Join. Beyond iGPU join processing, there are efforts to explore the HSA architectures' capabilities to a wider spectrum of problems [Hwu15]. Hetero-Mark [Sun+16] introduces several design patterns for general CPU-GPU programming to create a diverse benchmark. Meanwhile, Mukherjee et al. [Muk+16] developed several micro-benchmarks to show the effectiveness of so-called persistent kernels, which significantly reduce the launch overhead of GPU kernels. We expand upon both of those enticing results by developing tailor-made HSA solutions for streaming problems and couple them with the potential of using iGPU hardware. Zhang et al. studied workload partitioning for heterogeneous architectures [Zha+17; Zha+21; Zha+20]. With FineStream [Zha+20], they propose an iGPU stream processing engine that partitions the query workload evenly across the available processing units (i.e., CPU and iGPU). While FineStream maximizes resource utilization across multiple processing units via adaptive scheduling, our approaches focus on the efficient

utilization of the iGPU hardware. Moreover, a combination of our algorithms with FineStream is a promising direction for future work.

dGPU. Stream Processing on dGPUs suffers from the overhead of transferring data to the dGPU via the PCIe bus. A common solution for this problem is to batch incoming data and schedule different tasks to either the dGPU or the CPU. Joselli et al. [Jos+08] automate task distribution through sampling each processing unit’s capabilities, while Verner et al. [VSS11] schedule tasks according to deadline constraints. Meanwhile, Pinnecke et al. [PBS15] and SABER [Kol+16] both investigate the sliding window semantics of streams and suggest fragmenting larger variable size windows into fixed-size batches in order to allow high throughput regardless of individual query peculiarities. Additionally, SABER introduces an adaptive scheduling strategy based on past processing statistics that maximizes both the CPU and dGPU workload. Even though all work above aims to hide latency, none explore the benefits of a shared memory architecture using HSA. Furthermore, none of those approaches support sequential pattern matching. Cugola and Margara [CM12] present a first solution for sequential pattern matching on dGPUs. However, their approach aims at minimizing costly data transfer via the PCIe bus and does not take advantage of shared main memory as present in iGPU architectures. Finally, GStream [ZM11] designs an extendable framework for streaming applications on GPU clusters. However, through the usage of network distribution in addition to the PCIe bottleneck, it also lacks the benefits of a tightly coupled CPU-GPU architecture.

FPGA. Due to their low-level programming interface, FPGAs offer predictable latency and high performance fixed-point and bit-wise operations [Che+08]. For stream processing, there are solutions for windows, filter, aggregation [MTA09], windowed joins [TM11] and sequential pattern matching [WTA11; Mou+15]. In contrast to specialized, highly tuned hardware solutions, HSA serves as a back-end for the widely used OpenCL programming language. Thus, our approach can benefit from and be integrated into a variety of existing projects. This interoperability and the focus on consumer-level hardware make our work more generally applicable while laying the groundwork for possible iGPU solutions in the high-performance computing area.

4.2 Preliminaries

CPUs and GPUs target fundamentally different workloads. While CPUs aim for the efficient execution of sequential programs with a small number of high-clocked cores, GPUs target massive data-parallel workloads. They consist of thousands

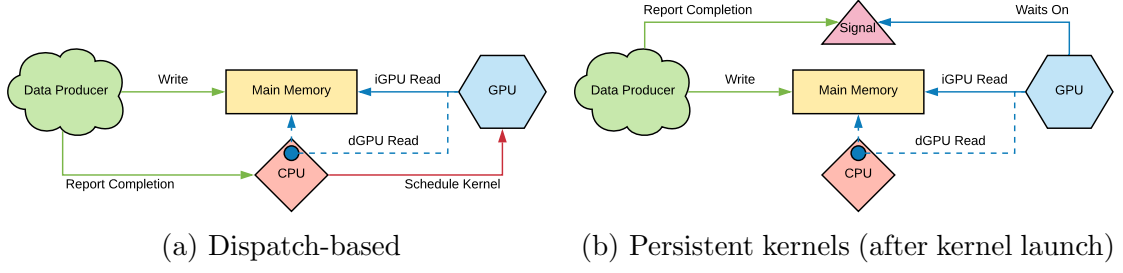


Figure 4.1: CPU kernel execution methods

of simple, lower clocked cores, each executing the same code on different data implementing the SIMD model. At first sight, GPUs seem to be a natural fit for high volume event stream processing. An event stream consists of homogeneous data items, and the processing pipeline is the same for all events of a stream. However, online SPEs are tailored towards low latency results, favoring a tuple-at-a-time or micro-batch processing scheme. For those small numbers of data items, GPU-based processing is typically doing more harm than good for the following reasons. First, the cores of a GPU operate at a lower clock speed than the CPU and do not implement features like out-of-order execution or speculative execution. Hence, they require a certain number of data items to be processed simultaneously to be effective. Second, classic GPU programming languages like OpenCL 1.2 treat the GPU as an accelerator that needs to be orchestrated by the CPU. In particular, the CPU manages data transfer to and from the GPU, schedules the execution of GPU programs (kernels), waits for their completion, and processes the results (e.g., passing them downstream the processing pipeline). We call this processing scheme dispatch-based execution and depict the general workflow in Figure 4.1 (a). Some data producer writes new data into main memory. Afterward, the producer invokes the CPU to perform kernel scheduling. Furthermore, for dGPUs, the CPU is also involved in data reads. This heavyweight synchronization results in noticeable processing delays, further increasing the minimum required batch size for GPU-based event processing to be beneficial.

While the design of the hardware induces the first limitation, the second is a software problem. In the following, we discuss HSA features allowing us to reduce the CPU/GPU synchronization overhead, and thus reduce the minimum batch size for effective GPU-based event processing.

4.2.1 Memory Management

In traditional GPU programming, memory management is the responsibility of the programmer. That is, memory regions holding relevant data are copied to/from the GPU's memory via corresponding function calls. In contrast, HSA supports fine-grained shared virtual memory (SVM). All computing devices share a unified address space, allowing the GPU to seamlessly access memory allocated by the CPU and vice versa. This feature is especially useful in combination with iGPUs as shown in Figure 4.1 (b). Due to a physically unified memory hierarchy, iGPUs do not require a memory copy operation to ship data to the GPU. SVM enhances this feature by also eliminating the need for mapping/unmapping memory regions. In comparison with a traditional dGPU work pattern, iGPUs in combination with SVM reduce both latency introduced by copying data back and forth and interaction with the CPU when new data is available.

Note that OpenCL 2.x¹ also features fine-grained SVM. However, we focus on HSA in this thesis and make use of more advanced features not available in OpenCL 2.0.

4.2.2 Signals

HSA provides a signaling mechanism, enabling lightweight communication between CPUs and GPUs. Depending on the platform, a signal is a 64- or 32-bit signed integer value. Runtime functions available on all processing units allow either unit to atomically manipulate the signal value (e.g., compare-and-swap, exchange). However, the most interesting feature of signals is that processing units can wait for updates of the signal value offering a lightweight synchronization mechanism across processing units. In particular, signals allow the implementation of persistent kernels [Muk+16]. Unlike dispatch-based execution (cf. Figure 4.1 (a)), a persistent kernel is launched once and stays active for the entire application lifetime. Figure 4.1 (b) depicts the associated workflow after launching a persistent kernel. The data producer changes the signal value if new data becomes available. Due to the signal value change, the kernel wakes up and immediately processes the new data. After processing the new data, the kernel signals completion and waits for more input data. This unique HSA feature reduces the overhead of a kernel launch and, for iGPUs, fully eliminates the need for CPU interaction upon the arrival of new data. The drawback of this approach is that the number of threads executing the kernel is fixed for the entire application lifetime since it is defined once during kernel launch. If the rate of the input data varies, this setting

¹https://www.khronos.org/registry/OpenCL/specs/2.2/html/OpenCL_C.html

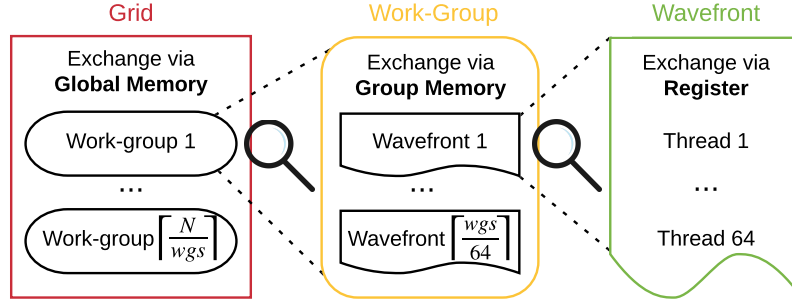


Figure 4.2: HSA thread group hierarchy

may not deliver the maximum possible performance. For example, if there are fewer data items than threads, some of them effectively execute a noop operation.

4.2.3 Data Exchange

Similar to traditional GPU programming languages, HSA organizes threads in a hierarchy of thread groups, which can communicate with each other via data exchanges on different memory layers. Figure 4.2 shows the overall hierarchy from left (largest group) to the right (smallest group). While every group is associated with a fixed memory layer, the grid and work-group size are set when scheduling a kernel. HSA schedules kernels with two parameters, the grid size N and the work-group size wgs . The grid size defines the global problem size given by the total number of data items to be processed. A grid is processed by multiple work-groups, each composed of wgs threads. Hence, the number of scheduled work-groups is $\lceil \frac{N}{wgs} \rceil$. Threads within a work-group can be synchronized using so-called barriers and share a small but fast local memory. Those threads are further grouped into so-called wavefronts (WFs), which are the smallest unit of execution on a GPU core. They are composed of 64 threads, resulting in $\lceil \frac{wgs}{64} \rceil$ WFs per work-group. All threads within a *wavefront* execute on a single instruction pointer. As a consequence, if threads within a WF need to follow different code branches (e.g., due to an if-else-statement), their execution has to be serialized. While threads fulfilling the branch condition continue processing, the other threads are stalled and vice versa.

Typically, threads exchange data via the group memory, if they belong to the same work-group, or via global memory otherwise. HSA provides an efficient mechanism for exchanging data between the threads of a WF without using auxiliary memory

by simply swapping register values. Furthermore, HSA provides information about the currently executing threads of a WF (i.e., threads not stalled due to branched execution).

Low-cost data exchange and the information about active threads enable us to store variable-length results densely inside an output buffer efficiently. In our case, variable-length results occur during filtering, join processing, and sequential pattern matching, since we cannot determine upfront which event satisfies the corresponding predicate. Without these operations, every thread that needs to write a result would execute an atomic operation on the slow global memory to reserve space in the output buffer. Instead, we can determine the number of results per WF by querying the number of active threads after predicate evaluation and reserve space in the output buffer for all results with one atomic operation on the global memory. This allows efficient result propagation on the GPU and eliminates the need for CPU interaction also for this task.

In summary, SVM, signals, and WF data exchange operations form the basis of our approaches, which we describe in the upcoming section.

4.3 Implementation

This section presents our processing framework featuring filtering, windowed aggregation, windowed joins, and sequential pattern matching. For each operator, we offer four different implementations. Single- and multi-threaded CPU versions, a dispatch-based GPU implementation, and a HSA-enabled implementation optimized for iGPUs. This gives a platform that allows (i) a fair comparison between all four processing techniques and (ii) an easy combination of GPU and CPU operator implementations in a single pipeline.

In the following, we first briefly describe event queues that establish the connection between event sources, operators, and sinks. Then, we discuss the implementation details of all four operators with an emphasis on the HSA variants.

4.3.1 Event Queues

An event queue decouples each pair of consecutive operators such that the upstream operator places its result events into the queue and the downstream operator consumes them from the queue. We implement the queues as array-backed

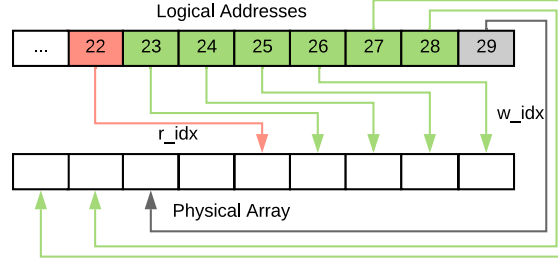


Figure 4.3: An example of logical queue addresses

ring buffers with a fixed maximum capacity in the number of elements (*size*). The backing array is contiguous in memory and, because of SVM, can be easily accessed by all computing devices. Every queue manages two indexes. One index points to the next write slot (idx_w) and the other one to the next read slot of the queue (idx_r). Instead of relying on the physical addresses in an array that may shrink/grow/loop around, we introduce logical addresses such that enqueueing events increment idx_w and dequeue operations increment idx_r . Consequently, the following invariants hold:

1. $idx_r \leq idx_w$
2. $idx_w - idx_r \leq size$

It is easy to see that all valid data lies between idx_r and idx_w and that a simple modulo computation obtains the physical array indexes. Figure 4.3 illustrates this with an example: The physical array has nine addresses, while the logical addresses continue to grow. Twenty-two points to the next data item to read, while 29 indicates the next write slot. Hence, slots 22 to 28 store valid data items. The indexes are implemented via HSA signals ensuring the proper propagation of updates across devices. This way, readers can wait for data by waiting for updates on the idx_w , and writers can wait for free space on the idx_r .

4.3.2 Filtering

The filter operator consumes a single input stream, applies a user-defined predicate to every incoming event, and forwards all events satisfying this predicate to the output stream (cf. Section 2.3.2). Unlike windowed operators, the filter processes every event exactly once. Hence, parallel processing requires the batching of input events at the cost of adding latency. However, even though batching enables predicate evaluation in parallel, the writing of results requires synchronization among threads to preserve the temporal order of output events. When done naively (e.g.,

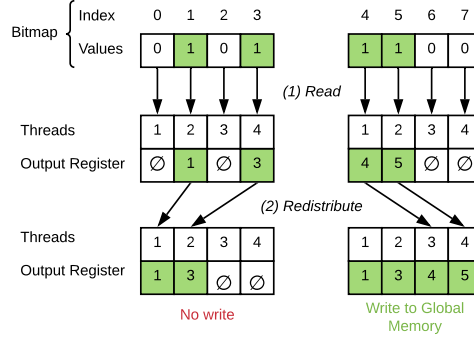


Figure 4.4: An example how results are written by the HSA-based filter implementation

by serializing threads), this synchronization entirely voids the savings achieved by parallelizing the predicate evaluation.

Our HSA-based implementation aims at minimizing this overhead. The basic idea is to use two separate kernels working in parallel. We call those kernels driver kernel and eval kernel. The driver kernel writes results to the output queue, and the eval kernel checks the filter condition. Both kernels communicate via signals and shared bitmaps. The eval kernel uses the signals to notify the driver kernel about completing the predicate evaluation for a batch of events. In contrast, the driver kernel uses them to signal that all results of a batch have been written to the output queue. We use double-buffering to keep both kernels busy. The driver kernel reads one bitmap to produce results, while the eval kernel fills the other bitmap. Therefore, the eval kernel sets the corresponding bit within the vector for every input event that fulfills the predicate. Every thread of the eval kernel processes 8 consecutive events so that its results fit into a single byte of the bitmap. Thus, the byte-offset within the bitmap is unique for every thread and it is unnecessary to synchronize the eval kernel threads.

When receiving the completion signal, the driver kernel writes the events that satisfied the predicate to the output queue. A naive solution for this would be the following. Every thread of the kernel inspects a single bit of the bit vector, and if the probe was successful, it writes the corresponding event to the output queue. However, for sparsely filled bit vectors, many threads are stalled during this write operation. Instead, we use an efficient WF data exchange to assign an output event to every thread before writing the results. Figure 4.4 shows four threads processing a vector of 8 bits in two cycles. In the first cycle, the bit vector contains 2 results at positions 1 and 3, detected by threads 2 and 4, respectively.

However, instead of writing them directly and causing a stall of the remaining threads, we redistribute the indexes to threads 1 and 2 and inspect the next four values. This time, the probes of threads 1 and 2 succeed, and we redistribute the corresponding indexes to threads 3 and 4. Finally, each of the four threads writes a result to the output queue in parallel and thus contributes to the writing process.

In our implementation, the driver kernel executes with 64 threads (i.e., a single WF), which allows us to inspect eight result bytes at once and batch up to 64 results in a single write operation. Furthermore, we also parallelize the reads from the bit vector, such that every thread reads eight bytes from a different location of the buffer. Again, we distribute those values among all threads via WF data exchange methods. Consequently, the batch size for this operator should be a multiple of 4,096 events (64 threads, each reading 64 bits) to fully utilize all threads in every step of the driver kernel.

Alternative Implementations

In the single-threaded CPU variant, a single CPU thread executes all tasks sequentially. It reads events from the input queue, applies the filter predicate, and writes results to the output queue. Input batches are evaluated in a tight loop with compiled predicate functions, allowing the compiler to optimize the resulting code.

In the multi-threaded case, a single CPU thread replaces the driver kernel, while a configurable number of additional threads replace the eval kernel. The task distribution among those threads is analogous to the HSA implementation. The data management thread waits for input data and materializes the result stream, while the worker-threads evaluate the filter predicate in parallel and write their results into a bitmap.

Our HSA implementations naturally generate result batches per WF and this way saves expensive atomic operations on the queue indices. Thus, for a fair comparison, we also implement output batching for the single- and multi-threaded CPU implementations. We collect up to 64 events in a dedicated buffer and flush them to the output queue in one go.

The dispatch-based GPU variant is similar to the multi-threaded variant. A single CPU thread is responsible for data management and kernel scheduling, while the GPU handles the data processing. In addition, we make use of SVM to avoid explicit memory management in the CPU thread.

4.3.3 Windowed Aggregation

Typically, windowed aggregates are computed incrementally by computing partial aggregates over *slide* events and composing the final result for the whole window from those partial aggregates (cf. Section 2.3.3). For ease of presentation, our implementation re-evaluates the entire window for every produced result. However, the proposed technique can easily be extended to support incremental aggregate computation. In addition, we limit the discussion to commutative and associative aggregation functions. The reason is that those functions take advantage of data-parallel computation and represent a wide range of common aggregates (e.g., sum, average, minimum, maximum).

Similar to filtering, windowed aggregation is composed of a driver kernel and an eval kernel synchronized via HSA signals. The driver kernel moves events from the input queue to the window and triggers aggregate evaluation immediately after inserting *slide* events. Note that this approach does not introduce any batch-related latency. The eval kernel performs aggregation by computing a parallel reduce operation on the window. For a standard GPU, a reduction runs on three levels: threads, work-groups, and grid. Each of them uses the fastest memory available. For instance, after summarizing n items, a thread stores its result in the local memory of its work-group. With HSA, we improve this reduction in two ways. First, instead of storing one value per thread in the local memory, we use WF data exchange operations to efficiently reduce the results of a whole WF (i.e., 64 threads) into a single value before storing it in the local work-group memory. This cuts down the data to process at the next level by a factor of 64. Second, the results of all work-groups are combined to obtain the final value of the reduction. Thus, this combination is a synchronization barrier for all work-groups. Because there exist no synchronization mechanisms across work-groups in languages like OpenCL, this step typically involves the CPU. With HSA signals, this is done exclusively on the GPU in the following way. Every work-group atomically increases a signal value by one upon completion. Thus, a signal value equal to the number of work-groups signals the completion of all computations. The driver kernel waits for all work-groups to complete, performs the final aggregation, and propagates the result event to the output queue. Since the result is a single event per window, this is done by a single thread of the eval-kernel.

Alternative Implementations

The alternative implementations are similar to the filter operator. In the single-threaded case, all work is done by a single CPU thread, while in the multi-threaded

case, both kernels are replaced by corresponding CPU threads. However, compared to the filter, windowed aggregation does not require result batching since it produces exactly one result per window. The dispatch-based adaption also follows the description of the filter implementation.

4.3.4 Windowed Joins

The join operator consumes two input streams (denoted as left and right in the following) and manages a window for each of them (cf. Section 2.3.4). Like aggregation, the join processes incoming events at a tuple-at-a-time basis and introduces no batch-related latency.

Different to the HELLS-Join [Kar+13], which manages window updates and result construction on the CPU, our HSA implementation runs solely on the GPU. Therefore, we again decompose the implementation into a driver kernel and an eval kernel. Like for aggregation, the driver kernel moves incoming events from the input queues to the windows and triggers the eval kernel via HSA signals. The main difference here is that the join consumes two input streams. Therefore, the join operator keeps track of the heads of both input queues to process incoming events in the correct temporal order.

The eval kernel processes a newly arrived event e_{new} by evaluating the join predicate on e_{new} and all events active in the other stream's window. To do this in parallel, we split the window's content into slices of 64 events such that each of them is processed by one of the WFs. Within a WF, every thread evaluates the join predicate on one of the slice's events. Then the threads that evaluate the predicate to false are stalled, while the remaining active threads write their results to the output queue. Note that all results carry the timestamp of e_{new} . Hence, the output order among those results does not matter, which simplifies synchronization among the WFs when writing results. In order to write in parallel, every WF directly reserves its contiguous slots in the output queue as described in Section 4.2.3 (i.e., using the number of results determined by counting the number of active threads). The WF's threads first write their results to the queue and then make them visible by increasing the queue's idx_w signal. This is the only synchronization point during the join process: A WF writing r results to slots $[s_1, s_r]$ must ensure that all WFs writing to slots smaller than s_1 completed their write operation, before increasing the idx_w to $s_r + 1$. Even though this method effectively serializes the *announcement* of new results (not the writing), it achieves good performance independent of the number of events to write as we will show in Section 4.4.

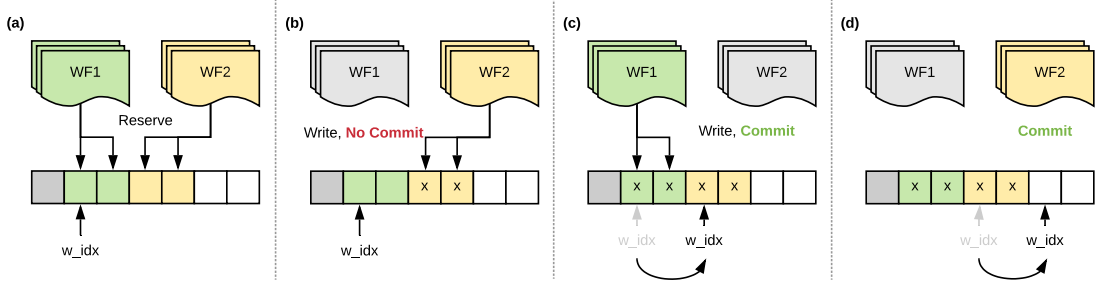


Figure 4.5: An example of concurrent writes in the join operator

Figure 4.5 shows this workflow for two simultaneously writing WFs. In the first step (see Figure 4.5 (a)), both WFs reserve two contiguous slots in the output queue. *WF2* finishes writing before *WF1* (see Figure 4.5 (b)) but is not allowed to increase the queue's idx_w , because otherwise consumers may read incomplete data. Instead, *WF2* waits for idx_w to reach its first write slot (via HSA signals). This ensures that all previous data is written. Then, *WF1* completes its write operations and immediately increases idx_w (see Figure 4.5 (c)). Finally, *WF2* is notified about the new value of idx_w and in turn commits its results (see Figure 4.5 (d)).

Alternative Implementations

As for the aggregation, the CPU-based implementations are similar to the filter. In the single-threaded case, all work is done by a single CPU thread, while in the multi-threaded case, both kernels are replaced by corresponding CPU threads. However, for the dispatch-based variant, we implement the HELLS-Join following the description in [Kar+13].

4.3.5 Sequential Pattern Matching

Our sequential pattern matching implementation adapts a nondeterministic finite automaton (NFA)-based evaluation approach, similar to SASE+ [DIG07]. For every symbol, there is a state in the NFA, and events that fulfill a symbol's condition trigger the corresponding transitions. Figure 4.6 shows an NFA for the example pattern $A B^* C$ with matching strategy $M = \text{contiguous}$. It consists of one state per symbol (S_A , S_B , S_C), and a starting state S_0 . When S_0 is active, the arrival of an event satisfying the condition of symbol A triggers a transition to S_A . From there, we reach the accepting state S_C with an arriving C or the Kleene-star state S_B upon an arriving B .

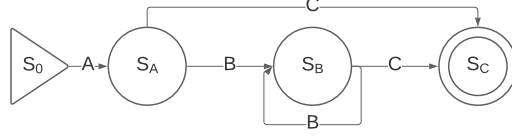


Figure 4.6: NFA representation of the example pattern $A B^* C$.

Every state is associated with a list of partial matches (PMs). A PM captures all events that contributed to reaching the current state. For instance, in state S_A , every PM consists of a single event that fulfills the condition of symbol A . In contrast, for S_C , a PM contains one event for A , one for C , and a variable amount of events for B (due to the Kleene-star quantifier).

Our iGPU-based sequential pattern matcher uses the iGPU and the CPU cooperatively by distributing tasks to the best-suited processing unit. The iGPU checks the window constraint and symbol conditions for incoming events, while the CPU performs compaction on the main memory to keep memory access efficient. In the following, we describe the tasks of both units in detail.

iGPU Tasks

Every state of the NFA is associated with a GPU kernel. Upon the arrival of a new event e_{new} , we run all those kernels with e_{new} as a parameter. Every kernel thread processes e_{new} together with one of the PMs of its state. This processing consists of two phases: check and propagate. The check phase first evaluates the window constraint on the examined partial match. Then, it evaluates the symbol condition associated with each of the state's outgoing transitions. The propagate phase takes the appropriate action depending on the outcome of the check phase and the applied matching strategy (cf. Section 2.3.5). Those actions are summarized in Table 4.1. Independent of the matching strategy, we drop a PM if it exceeds the window constraint. In contrast, if e_{new} fulfills none of the transition predicates, the action depends on the matching strategy. While, *skip-till-next* and *skip-till-any* simply ignore e_{new} , *contiguous* drops the PM, since no contiguous match is possible anymore. Finally, in case e_{new} triggers a transition, we add it to the PM. Additionally, if the transition was a forward transition, we move the PM to the next NFA state. Note that the *skip-till-any* strategy clones the PM before adding e_{new} in order to capture all possible matches.

Our implementation manages the PMs of a state in a dense array to ensure the iGPU has efficient access to contiguous memory regions. Furthermore, we perform

Check result	contiguous	skip-till-next	skip-till-any
Window exceeded	Drop	Drop	Drop
None	Drop	None	None
Self transition	Add e_{new} to PM	Add e_{new} to PM	Clone PM; Add e_{new} to PM
Forward transition	Move PM to next state; Add e_{new} to PM	Move PM to next state; Add e_{new} to PM	Clone PM; Move PM to next state; Add e_{new} to PM

Table 4.1: Result of the check phase and the corresponding action for each of the three matching strategies.

as much of the memory management as possible on the iGPU to take advantage of the massive parallelism.

We add e_{new} to a PM by copying the event data to the PM’s event array. Since every GPU thread processes exactly one PM, no synchronization among threads is required and all copying runs in parallel. To clone a PM, the corresponding GPU thread copies the PM data to the first free slot at the end of the state’s dense array. Therefore, the thread reserves a slot via an atomic operation to avoid data races. Likewise, for moving a PM, the thread first copies the data to a free slot in the dense array of the target NFA state before it drops the PM from the array of the current slot. Note that similar to adding an event to a PM, all copying runs in parallel; only reserving a slot in the destination state requires synchronization among GPU threads.

Dropping PMs due to the window constraint or a move to the next NFA state leaves gaps in the dense array. Hence, we require a compaction step to keep memory access efficient for the iGPU. However, compaction runs more efficiently on the CPU, and thus we only mark the corresponding array slot as dropped. This works similar to result propagation in our join implementation (cf. Section 4.3.4). Every kernel thread whose PM should be removed calls an atomic operation to reserve a slot in a dedicated global memory area. Then it writes the corresponding array index into this slot. After e_{new} is fully processed by the kernel, the CPU performs the compaction, which we detail in the upcoming subsection.

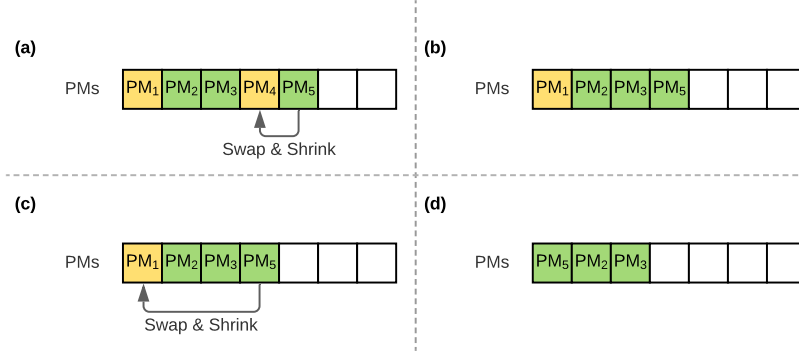


Figure 4.7: Example for the removal of two PMs (PM_1 , PM_4) in the dense array.

CPU Tasks

In total, the CPU executes three tasks. First, it serves as a dispatcher to schedule kernels for every state of the NFA. Unlike filtering, aggregation, and joins, we do not use persistent kernels here for the following reason. Persistent kernels fix the number of threads per kernel for the entire application lifetime. However, due to timeouts or transitions, the number of PMs per state (i.e., the number of required threads) varies with every arriving event. As a consequence, a fixed number of threads per kernel leads to over-/underutilization of the hardware. Thus, we use a dispatch-based scheduling approach to appropriately configure the number of kernel threads (i.e., the grid-size) for every arriving event.

Second, the CPU takes over the removal of dropped PMs. In order to keep access to PMs efficient, their removal should not create gaps in the dense array. Gaps are avoided by moving elements from the end of the array to the positions of the removed PMs. Since the removal of PMs happens in descending order of their position, our method guarantees to move only valid PMs (i.e., PMs not scheduled for removal). Figure 4.7 shows an example for a state with five PMs (PM_1, \dots, PM_5), whereof PM_1 and PM_4 are scheduled for removal by the GPU. First, PM_4 is deleted by moving the last valid element PM_5 to its position and releasing the last array slot (see Figure 4.7 a,b). Since PM_5 is again the last element within the array, it is moved one more time when removing PM_1 (see Figure 4.7 c,d).

The third task of the CPU is the creation of result events for PMs that reached a final state of the NFA. Therefore, it iterates all PMs of the corresponding state,

applies the *map* function specified with the pattern (cf. Section 2.3.5), and stores the results in the output queue.

Alternative Implementations

The CPU adoptions of our sequential pattern matcher are similar to those of the windowed join. In the single-threaded case, the CPU thread performs all the work (i.e., it iterates over all PMs in every NFA state to perform the corresponding checks). In contrast, the multi-threaded version works similar to our GPU approach, but instead of scheduling a kernel per state, we use a CPU thread.

In addition, we implement the dGPUs-optimized CDP algorithm proposed by Cugola and Margara [CM12]. Different from the original algorithm, we make use of HSA’s SVM feature instead of managing memory programmatically. The CDP algorithm takes a backward evaluation approach to avoid materializing partial matches (i.e., avoid memory transfers via the PCIe bus). It first buffers events for each of the NFA states. Then, if an event matching the final state arrives, it traverses the state buffers in reverse order to produce matches. CDP offers two so-called event selection policies that are somewhat comparable to our matching strategies. The multiple selection policy corresponds to the *skip-till-any* matching strategy and produces all possible matches. In contrast, the single selection policy selects only a single event per PM, which results in a similar computational complexity as for the *skip-till-next* strategy. However, due to the backward evaluation, the produced results differ. Thus, we only implement the multiple selection policy to ensure a fair comparison between our method and CDP. Moreover, CDP does not support Kleene quantifiers, which limits its practical applicability considerably.

4.4 Experimental Evaluation

This section presents a selection of important results from an extensive evaluation of our prototypic system. After a description of the experimental setup, we first discuss the effect of persistent kernels before we discuss the performance results of our HSA-enabled operator implementations.

³AMDGFX kernel repository <https://bit.ly/2vP3jAu>

⁴<https://github.com/RadeonOpenCompute/ROCm/tree/roc-4.0.x>

Component	Description	
CPU	AMD Ryzen 2400G@4x3.6GHz	
Memory	32 GB DDR4@2133MHz	
OS	Debian GNU/Linux buster/sid	
Kernel version	5.0.0-rc7-kfd+ ²	
HSA Implementation	ROCm 4.0.1 ³	
C/C++ Compiler	clang 12.0.0	
	iGPU	dGPU
Chip	AMD Vega11	AMD Vega56
Compute Units	11	56
Board Memory	-	8 GB HBM2
Clock-Speed	1250 MHz	1590 MHz

Table 4.2: Details of the evaluation platform

4.4.1 Setup

We execute all experiments on an iGPU as well as a dGPU. The system specification is shown in Table 4.2. The host code is written in C++ and compiled with clang 12 using optimization level `-O3`. The GPU code is written in OpenCL 2.0 with HSA specific extensions and compiled with the clang compiler provided by the ROCm platform⁴.

In order to examine a wide range of workloads, we use a synthetic event stream with the following characteristics. Every event consists of a timestamp (64-bit unsigned integer value) and a payload of 6-dimensional 32 bit floating point values (uniformly distributed in the unit interval).

We use our single-threaded operator implementations as a baseline for all experiments. We choose our own implementation over production-ready frameworks like SPEs like Apache Flink [Car+15] or Spark Streaming [Zah+16] for the following reason. Those engines are designed to scale out across many nodes to achieve high throughput and do not exploit the hardware resources of a single machine efficiently [Zeu+19], which leads to a very low single-node throughput. For example, we run experiments with a filter operation selecting 1/1000 events from the event stream on Apache Flink (single node, default local configuration) and compare it with all of our filter implementations. The results given in Figure 4.8 confirm that scale-out SPEs cannot compete with highly optimized single-node implementations. Our filter implementations outperform Apache Flink by up to two orders of magnitude. Thus, we do not include those engines in the upcoming evaluation.

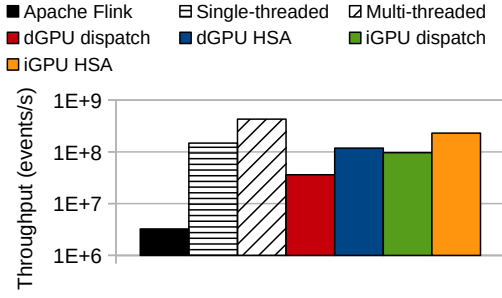


Figure 4.8: Throughput-comparison of the filter operation between Apache Flink and the variants of our processing framework.

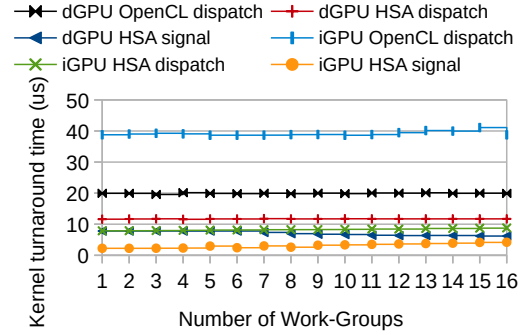


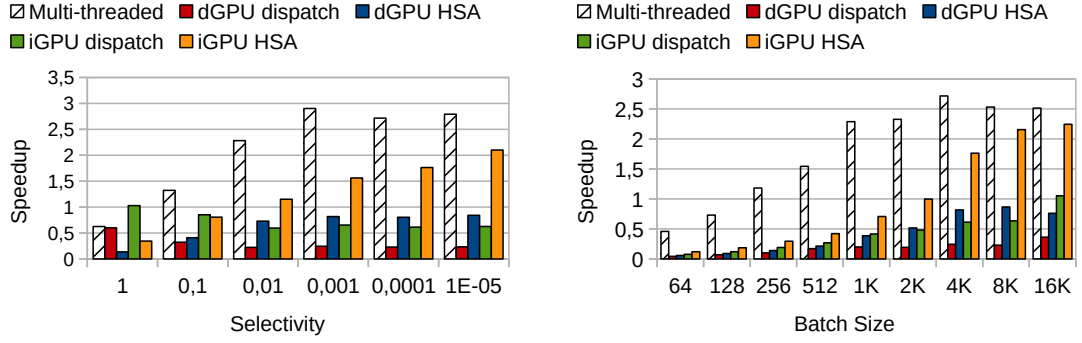
Figure 4.9: Comparison of kernel turnaround time

4.4.2 Persistent Kernels

We start with analyzing the savings in kernel launch time when using persistent kernels. For that, we implement a noop-kernel, dispatch it with (i) the OpenCL runtime, (ii) the HSA runtime, and (iii) HSA signals and measure the time from dispatching until the kernel completes execution. We schedule 1024 threads per work-group and vary the number of groups. Figure 4.9 shows that signals outperform the dispatch-based approaches independent of the GPU-type. Furthermore, dispatching via HSA is cheaper for the iGPU than for the dGPU. Interestingly, the OpenCL dispatch performs better for dGPUs. Since this behaviour can not be observed with HSA, we believe the OpenCL code base for dGPUs is more mature (i.e., optimized) than for iGPUs.

4.4.3 Operator Evaluation.

We evaluate every operator in a single operator pipeline, varying its parameters (e.g., window size, selectivity). In all experiments, the operators consume events from preloaded input queues and write their results to an output queue with sufficient space to avoid wait times during measurements. The multi-threaded CPU implementations use four worker-threads. This setting matches the number of physical CPU cores and results in the best throughput for all operators. For the GPU variants, the optimal work-group size and number of scheduled work-groups depend on multiple factors (implementation variant, operator type, window size,



(a) Fixed batch size (4,096), varying selectivity

(b) Fixed selectivity (0.001), varying batch size

Figure 4.10: Speedup of filter implementations compared to the single-threaded implementation

device type). We experimentally determine the optimal setting upfront for every combination and use this setting during the evaluation. As an example, the iGPU HSA join uses 11 work-groups (i.e., the number of computing units) with 64 threads per group.

Filter

We evaluate the filter implementations with batch sizes ranging from 64 to 16K elements. The predicate computes the Euclidean distance from the origin to the point given by the 6-dimensional event payload and filters all events with a distance above a user-defined threshold. We vary the condition's selectivity from 1 (all events pass) to 10^{-5} and measure the throughput achieved by each of the implementations.

Figure 4.10 shows two representative results from this experiment. In Figure 4.10 (a) we fix the batch size to 4K elements and vary the selectivity of the filter condition (x-axis). Only multi-threaded and iGPU HSA are able to outperform the single-threaded implementation. In both cases, the speedup increases as the selectivity decreases. The reason is that the lower the selectivity is, the fewer events are written to the output queues, and hence the less synchronization is required between the processing threads. However, maintaining the temporal ordering of results is more efficient on the CPU than on the GPU since a single thread is responsible for this instead of a whole WF. Thus, iGPU HSA is not able to outperform the multi-threaded implementation. Interestingly, iGPU dispatch performs better than

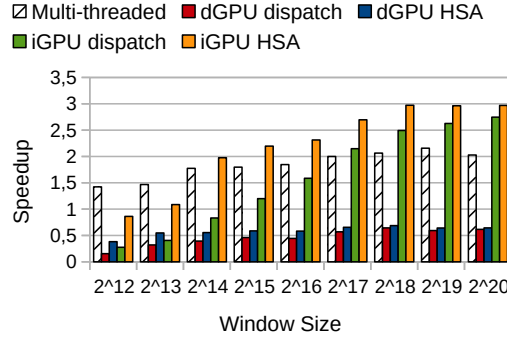


Figure 4.11: Speedup of aggregation implementations compared to the single-threaded implementation with varying window sizes

iGPU HSA for high selectivities (1.0, 0.1). The reason for this behavior is that iGPU dispatch uses the same result writing mechanism as the multi-threaded variant. For high selectivities, the advantages of single-threaded writing outweigh the performance penalties of the dispatch overhead. Finally, the dGPU variants are not able to compete with the other implementations because the overhead for moving data back and forth voids the benefits of parallel processing. However, the results clearly show that HSA substantially improves performance for dGPUs compared to traditional *dispatch-based* processing.

In Figure 4.10 (b), we fix the selectivity to 10^{-3} and vary the batch size from 64 to 16K elements (x-axis). The results show that the multi-threaded implementation requires only 64 elements per thread (256 element batch, four threads) to outperform the single-threaded variant. Furthermore, its speedup stabilizes with batch sizes above 4K elements. All the GPU-based implementations benefit from larger batch sizes, but only iGPU HSA is able to outperform the single-threaded variant. Additionally, its performance improves beyond batch sizes of 4K elements and approaches the speed of the multi-threaded implementation.

Aggregation

We evaluate aggregation by computing the sum of a single attribute while varying the window size from 4K (2^{12}) to 1M (2^{20}) events and keeping the slide constantly at 1 element. Hence, we re-evaluate the window for every incoming event. Again, we measure the system's throughput and compare our implementations to the single-threaded baseline.

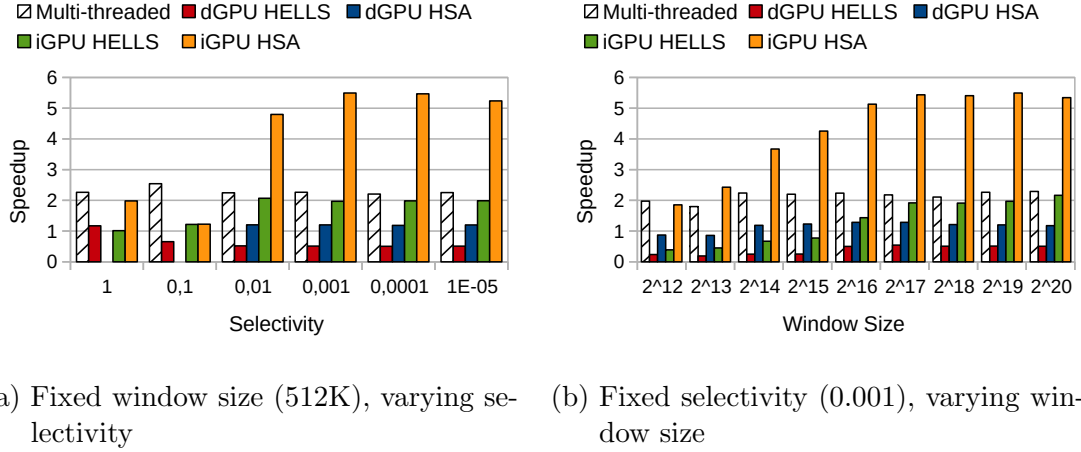


Figure 4.12: Speedup of join implementations compared to the single-threaded implementation

Figure 4.11 shows the results of this experiment. Except for very small windows (4K and 8K elements), iGPU HSA constantly outperforms the other implementations. Moreover, multi-threaded and both iGPU variants scale with increasing window size, while the dGPU variants barely reach half the throughput of the single-threaded implementation. Again, the data-shipping overhead is responsible for the bad dGPU performance. However, unlike the filter experiments, the performance of the dispatch-based versions is close to the HSA implementations (on the respective hardware). The reason is that the time for collecting the result on the CPU is short because aggregation produces a single result event read from a fixed memory location. Hence, the time between two consecutive kernel invocations is almost negligible.

Join

Similar to the filter, the performance of the join operator depends on two parameters, the window size and the selectivity of the join predicate. Again, we compare the throughput of our implementations with the single-threaded baseline while varying the window size from 4K (2^{12}) to 1M (2^{20}) elements and the condition's selectivity from 1 (all events pass) to 10^{-5} . Two events fulfill the join condition if the Euclidean distance of their payloads is below a user-defined threshold.

Figure 4.12 shows two representative results from this experiment. In Figure 4.12 (a) we fix the window size to 512K elements and vary the selectivity of the filter

condition (x-axis). While iGPU HSA constantly outperforms the single-threaded implementation, it requires a selectivity of about 0.01 to be superior to the multi-threaded version. Again, this is because of the efficient result propagation on the CPU. This effect is not very noticeable for a selectivity value of 1.0 because all GPU threads are involved in writing results. For a selectivity of 0.1, however, the majority of threads are stalled during result writing. Lower selectivities relax this problem for the following reason. If no thread of a WF needs to write a result, the next window slice is processed immediately, and iGPU HSA clearly outperforms all other implementations. The iGPU HELLS-Join performs similar to the multi-threaded variant for low selectivities and falls behind for larger ones. dGPU HSA was not able to answer the queries with high selectivities because it transfers many results to main memory, which the SVM implementation could not handle. As the development of the ROCm platform is in an early stage, we expect this issue to be resolved in the future.

In Figure 4.12 (b), we fix the selectivity to 0.001 and varied the window size from 4K to 1M elements (x-axis). The results show that iGPU HSA performs similar to the multi-threaded implementation for small windows and clearly outperforms all other approaches for larger windows. The reason is that we only synchronize WFs that actually write results to the output queue and thus do not suffer from global synchronization needed during filtering. The HELLS-Join was designed for huge windows, which is reflected in this experiment. It requires at least 128K elements to compete with the multi-threaded implementation. Even though it uses a double buffering technique to keep CPU and GPU busy, the performance suffers from synchronizing both processing units. Similar to the other experiments, the dGPU severely suffers from data-shipping overheads but also benefits from our HSA-enabled processing techniques.

Sequential Pattern Matching

As shown in Section 3.5.3 the computational complexity (i.e., the number of active PMs per time unit) of sequential pattern matching is low for the *contiguous* matching strategy. Thus, the additional synchronization overhead fully eliminates the benefits of parallelization in this case. For this reason, we focus on the *skip-till-next* and *skip-till-any* strategies in the evaluation of our parallelization approaches.

In order to control the size of the NFA states (i.e., the number of PMs), we use the predicate ϕ as a condition for every symbol. ϕ is parameterized by δ and defined as follows

$$\phi_\delta = (|e_{new}.a_1 - e_{prev}.a_1| \leq \delta).$$

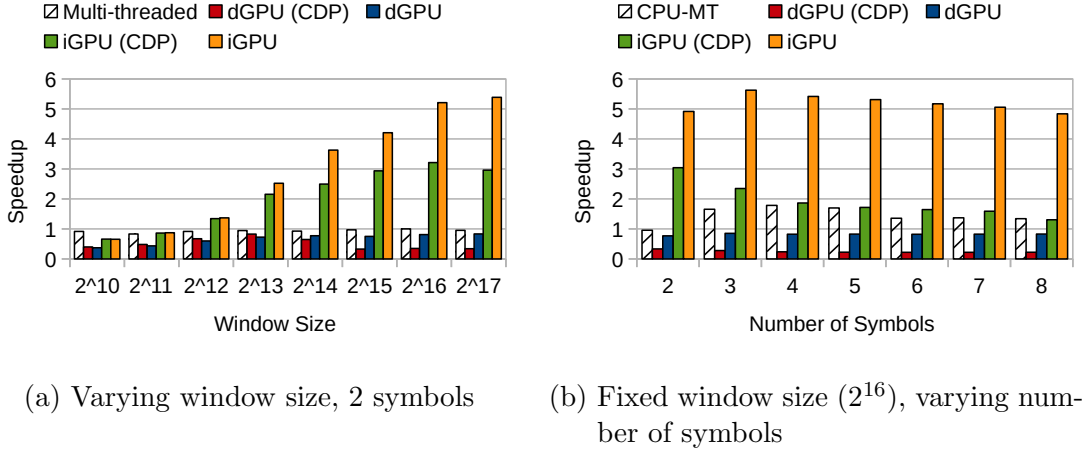


Figure 4.13: Speedup of pattern matching approaches compared to the single-threaded implementation using the *skip-till-any* matching strategy.

Here, e_{new} refers to the newly arrived event, while e_{prev} refers to the last event within the currently processed PM. The predicate evaluates to true if the difference of both events' a_1 attribute values is below the threshold δ .

In the first experiment we examine, how the presented approaches scale with the bare number of PMs. Therefore, we define a simple pattern (A, B) with matching strategy $M = \text{skip-till-any}$ and vary the window size from $2^{10} = 1024$ to $2^{17} = 131072$ events. The total number of PMs is kept equal to the window size by setting the predicate of A to *true* (i.e., every new event starts a PM) and the predicate of B to ϕ_{10-7} (i.e., a very unlikely case).

Figure 4.13 (a) shows the results of this experiment as a function of the window size. Independent of the window size, CDP as well as our approach perform poorly on dGPUs because the data transfer cost is too high and eliminates the benefits of parallelizing the predicate evaluation. Also the multi-threaded implementation is inferior to the single-threaded. The reason is that the multi-threaded approach uses one thread per NFA state. However, in this experiment with two symbols (i.e., one NFA state), there is only one extra processing thread. In this case, the additional synchronization costs eliminate the improvements from using one additional thread. In contrast, both iGPU variants scale nicely with the window size. However, CDP reaches its maximum speedup (3x) at a window size of 2^{16} , while our optimized approach scales to even larger windows and achieves a more than 5-fold speedup. The performance of CDP is limited because it does not materialize PMs and requires additional memory accesses for predicate evaluation. In contrast, our approach stores PMs densely in memory, which results in less wait

time.

In the second experiment, we study the efficiency of the presented approaches if the PMs are distributed across multiple states. Therefore, we fix the window size to 2^{16} and vary the number of symbols in the pattern. Like previously, we use the matching strategy $M = \text{skip-till-any}$ and set the predicate of the first symbol to *true*. For the remaining symbols, we use $\phi_{10^{-5}}$. By using a fixed δ for all conditions, the number of PMs gradually decreases from one NFA state to the next. In our setting, the first state always holds 2^{16} PMs, the seventh state only contains approx. 40 PMs.

Figure 4.13 (b) shows the results of this experiment. Similar to the previous experiment, both dGPU variants suffer from slow memory transfers and perform poorly. For more than two symbols, the extra threads allow the multi-threaded implementation to outperform the single-threaded. However, since the PMs are not evenly distributed among the NFA states, the multi-threaded implementation does not scale with additional symbols. The performance of CDP on the iGPU also suffers from additional symbols because PMs are not materialized and re-created with every arriving event. In contrast, our approach provides a 5-fold speedup, independent of the number of states, because it requires no re-computations and allows the flexible distribution of kernel-threads to NFA states.

In order to achieve an even distribution of the PMs over the NFA states, we repeat the previous experiment using the $M = \text{skip-till-next}$ matching strategy and an experimentally tuned δ for each of the symbol predicates. Figure 4.14 (a) shows the results for this experiment. Due to the missing support for $M = \text{skip-till-next}$, we could not use the CDP algorithm in this experiment. Both GPU variants behave almost the same as in the previous experiment. However, the iGPU performs slightly worse, since $M = \text{skip-till-next}$ incurs more memory-management tasks on the CPU due to moving (i.e., copy/remove) of PMs to the subsequent NFA states, instead of just copying them on the GPU. However, this time the multi-threaded implementation scales nicely with the number of symbols. As the machine has four physical cores, the speedup experiences a dip at five symbols. Nevertheless, due to hyper-threading, the performance improves up to eight symbols.

The last experiment analyzes the performance of the presented approaches when using a Kleene symbol. The setup is similar to the first experiment. The only two differences are that we insert a Kleene-plus symbol in between (i.e., the pattern is (A, B^+, C)) and use the matching strategy $M = \text{skip-till-next}$ to prevent exponential growth of the Kleene state. For the Kleene-plus symbol, we set $\delta = 10^{-5}$.

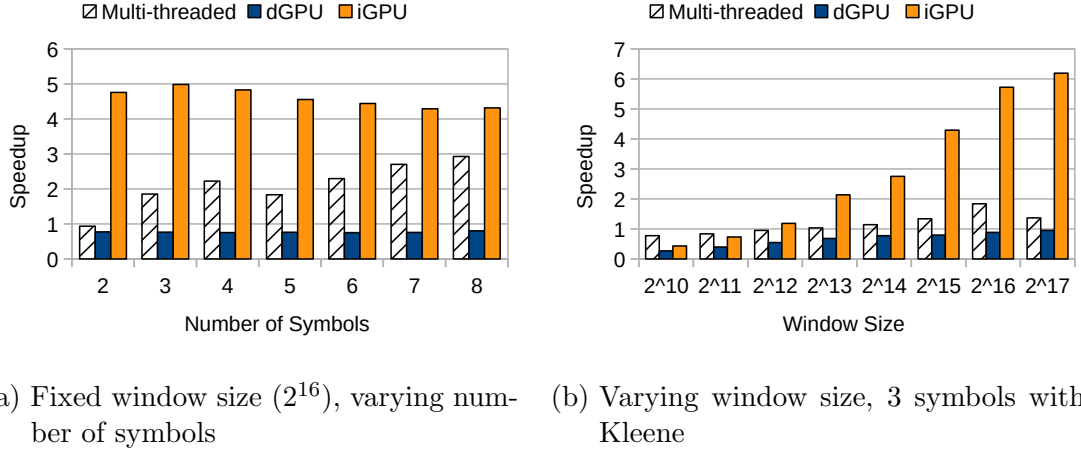


Figure 4.14: Speedup of pattern matching approaches compared to the single-threaded implementation using the *skip-till-next* matching strategy.

Figure 4.14 (b) shows the results of this experiment. Again, we do not report results for the CDP algorithm due to the lack of support for Kleene quantifiers and the *skip-till-next* matching strategy. The results are similar to those of the first experiment. Our iGPU approach nicely scales with the number of PMs, even though we only use three states. In contrast, the multi-threaded implementation suffers from the few states and provides only a very little speedup, independent of the window size. Finally, the dGPU again suffers from costly memory transfers and performs poorly for all tested window sizes.

4.5 Summary

Event processing systems evaluate queries that continuously perform the same operations over incoming events. The SIMD model is a natural fit to accelerate this type of processing, but dGPUs introduce latency that is often unacceptable for event scenarios. In order to bridge the gap between both worlds, we have presented a prototypical event processing system including filter, windowed aggregation, windowed join, and sequential pattern matching operations based on recent advancements in the HSA platform. We have shown that an iGPU, which shares its memory with the CPU, in combination with persistent kernels, can enable processing of small batches at low latency for the operations examined in our study. Moreover, we described a framework in which the iGPU and CPU can co-exist for more complex operations.

5

Pattern Matching on Temporal Intervals

The sequential nature of regular expression-based patterns has two significant deficiencies. First, the expressible temporal relationships are limited to before/after/at the same time. Conditions lasting for periods and their temporal relationships (e.g., *A during B*) are not, or only hardly, expressible in this approach. Second, due to the sequential nature of this process, efficient parallel execution strategies are scarce. Nevertheless, efficient parallel and distributed execution is a crucial aspect when dealing with ever-increasing data rates.

To overcome these deficiencies, we use the concept of situations (cf. Section 2.4) that capture real-world phenomena lasting for periods. Situations can be derived from event streams on the fly, and a temporal pattern (TP) can be matched using the situations' time intervals. Furthermore, situations reduce the input data by (i) summarizing relevant sub-sequences of the stream and (ii) filtering out events not relevant for matching the pattern, which, as we will show, allows for efficient parallel and distributed query processing.

We illustrate our idea with the following example. A traffic monitoring system continuously receives sensor data from connected cars (i.e., position, speed, acceleration). One of the system goals is to notify drivers about potential threats around their locations, such as an aggressively driving car. Among others, the American Automobile Association has identified the following two actions being indicators for aggressive driving¹: “*Operating the vehicle in an erratic, reckless, careless, or negligent manner or suddenly changing speeds*” and “*Driving too fast for conditions or in excess of posted speed limit*”. From these definitions, a pattern to detect aggressive drivers is: “A sharp acceleration followed by hard braking, both accompanied by a period of speeding.”

¹<http://www.iii.org/fact-statistic/aggressive-driving>

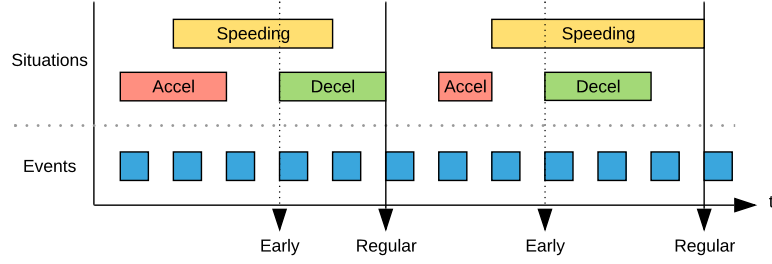


Figure 5.1: Detecting aggressive driving with situations.

Figure 5.1 illustrates this example. The stream of raw sensor readings is transformed into three situation streams, one per pattern component. Every situation consists of a time interval representing its temporal validity and a meaningful summarization of the event sequence it was derived from (e.g., the average speed during the speeding phase). The TP can then be matched by joining streams of situations with appropriate conditions on their interval relationships. Figure 5.1 also showcases two more desirable features for temporal pattern matching. At first, the way the situations are temporally related differs slightly among the two sketched matches. In the first match, they *overlap*, while in the second match, deceleration happens *during* the speeding situation. The query language should be flexible enough to cover these cases within a single query. Second, the pattern should be detected with the lowest possible latency. As depicted above, we can conclude both matches at the beginning of the deceleration situation since speeding holds at this point, and the pattern allows any combination of their endpoints. Technically, this means the system should conclude a successful match without exact knowledge about the validity of all situations. Additionally, when applied at a large scale with millions of cars, efficient parallel and distributed processing is inevitable to guarantee query responses in a reasonable amount of time.

This chapter presents *TPStream*, a holistic operator for complex temporal pattern matching on point event streams. *TPStream* detects matches at the earliest possible point in time by closely coupling derivation of situations with pattern matching. Unlike previous research, the operator and our low latency optimizations can easily be implemented in commonly available point-based systems. The reason is that we use time intervals only internally and produce point event streams as results. Moreover, *TPStream* is easily parallelizable and continuously adapts its processing strategy to deal with fluctuating data rates and changes in the data

distribution of incoming streams.

The remainder of this chapter is organized as follows. The next section discusses alternative solutions to temporal pattern matching in detail. Section 5.2 reviews related work, before we introduce *TPStream*'s query language in Section 5.3. In Section 5.4 we model all aspects of *TPStream* in an algebra. Efficient evaluation strategies, the algorithm for low-latency matching and our optimization techniques are presented in Section 5.5. Solutions for parallel and distributed execution of *TPStream* are introduced in Section 5.6. We evaluate the performance of *TPStream* in Section 5.7 and briefly summarize this chapter in Section 5.8.

5.1 State-of-the-Art

To the best of our knowledge, the only work on complex temporal relations (TRs) in event stream pattern matching is the *ISEQ* operator [Li+11]. However, *ISEQ* has several shortcomings concerning the desired features: First, the operator requires interval-events (i.e., situations) as input, leaving all aspects of deriving situations to an unknown external entity. Being unaware of the origin of interval events severely limits the operator in processing power (in terms of plan optimization). Moreover, it renders detection with the lowest possible latency impossible since there is no way to access an incomplete situation or indirectly manipulate the building of a situation through constraints. Second, a TP is specified using a conjunction of endpoint relationships (i.e., an ordering on start (ts) and end (te) of intervals). This way, alternatives are expressed by omitting one or more endpoints. For example, the pattern $A.ts < B.ts < A.te \leq B.te \vee A.ts < B.ts < B.te < A.te$ on two situations A and B is expressed as $A.ts < B.ts < A.te$. Hence, disjunctions like $A.ts < B.ts < A.te < B.te \vee B.ts < A.ts < B.te < A.te$ are not expressible in a single query. Instead, they require multiple queries in an approach without any specified optimization component to detect shared processing opportunities. Third, *ISEQ* does not provide any solution to parallelizing the process of matching complex TRs. Finally, *ISEQ* relies on auxiliary index structures and punctuation mechanisms for efficient query execution, complicating the integration into existing systems.

5.1.1 Straw Man's Approach:

Besides *ISEQ*, we identified two approaches to perform temporal pattern matching with point event streams. Thus, we can provide a point of comparison to

ESP systems featuring sequential pattern matching. The first approach works in two phases: In the first phase, a pattern matcher is deployed for every defined situation, computing its duration (start/end timestamp) and the desired summarizations. Technically, this means matching patterns of the form $\neg S S^+ \neg S$ with S being the situation's condition (e.g., *speed* > 70 *mph*). This pattern identifies the longest contiguous subsequence of events fulfilling S by surrounding the sequence S^+ with events not fulfilling the condition ($\neg S$). The result of this approach is a dedicated stream per defined situation, each of which is ordered by the end timestamp. Thus we can map the TP to a sequence of situations (reflecting the order of end timestamps, possibly containing alternatives). Finally, in the second phase, a dedicated pattern matching operator is used to find all matching sequences, whereby additional predicates ensure the proper ordering of start timestamps. While this approach derives situations including the desired summarizations, it fails to produce early results because, just like in *ISEQ*, situations are entirely derived before they are available for pattern matching.

The second approach uses a single pattern matching operator and expresses the TP as a single sequence of point events. In order to express temporal overlaps, we connect the conditions of all involved situations via a logical *AND*. For example, *Acceleration overlaps Speeding* is expressed as AB^+C with the following conditions: $A = \text{accel} > 8 \text{ m/s}^2$, $C = \text{speed} > 70 \text{ mph}$ and $B = A \wedge C$. Since this approach expresses patterns at the granularity of events, early results are achieved by simply omitting the last portion of the pattern. At the same time, this approach leaves summarizations of single situations to a post-processing step since it disassembles situations to express temporal overlaps.

5.2 Related Work

Besides sequential pattern matching and interval-based ESP systems, our approaches relate to Context/State in CEP, Stream Reasoning, and query processing in Spatio-Temporal Database Systems. We already discussed work related to sequential pattern matching and in interval-based ESP systems in Chapter 2. Thus, we focus on the remaining related concepts in this section.

Context/State in CEP. There has been recent work on introducing contexts into a CEP environment. CAESAR [Pop+16] associates queries to long-lasting context windows, detects them from incoming events as soon as they start, and suspends queries of inactive contexts. Similarly, Etzion et al. [Etz+16] use contexts to group up event types and process them together. While contexts and

situations are related concepts, the key difference is that contexts are purposefully decoupled from events. Therefore, it is not possible to query the relation of different contexts to each other. In contrast, *TPStream* focuses on efficient, adaptive, and low-latency implementations of those TRs. Likewise, work dealing with time periods such as states [HV15] or windowed aggregation [GAE06; Gro+16], cover the deriving aspect of situations, but lack interval relations [All83] or pattern matching.

Stream Reasoning. The semantic web community has worked on extending RDF triples with a time dimension, introducing the concept of temporally limited information, and thus allowing for a variety of graph-based temporal queries such as temporal joins [GGZ16]. C-SPARQL [Bar+09] deals with streams in particular. It introduces window semantics and corresponding aggregation capabilities but does not feature temporal relationships necessary for formulating sequential pattern queries. EP-SPARQL [Ani+11] and its implementation ETALIS assume that every data item is associated with a fixed time interval and support the usage of Allen’s temporal operators when formulating queries. Similarly, TEF-SPARQL [Kie+13] supports all temporal relationships in a language and algebra build around always valid triples and so-called facts. Analogous to situations, facts feature a start time and a potentially undetermined expiration time which may be set in the future. Like *TPStream*, EP-SPARQL and TEF-SPARQL aim to solve the combination of fleeting events with longer-lasting situations. However, they are ultimately complementary to the work presented here due to a variety of reasons. First, the approaches aim to empower SPARQL with event semantics. In contrast, we look at the problem from the perspective of event processing in order to facilitate easy integration into existing ESP languages and systems. Second, both works focus on establishing the language in particular while *TPStream* also deals with the implementation side to improve latency and parallelization aspects. Finally, a more recent study [Gao+15] showed performance benefits of TEF-SPARQL’s algebraic notion of ongoing facts by compiling them into Esper queries. We go a step further by not only supporting ongoing situations but also developing optimization mechanisms unique to the pattern matching process.

Spatio-Temporal Database Systems. The spatial database community studied the problem of spatio-temporal pattern queries (STPQ) in trajectory databases, e.g., in [Erw04]. In general, these approaches cannot be directly applied to an event processing environment because they are built on top of a persistent trajectory database model, where movement histories are already stored and indexed in the database. However, the design of [SG11] in particular served as a foundation for our proposed *TPStream* operator as *TPStream* adapts similar concepts of temporal predicates and constraints. Similar to the spatial community, Helmer et al. [HP16]

introduce an event query language for high-level event detection for temporal databases. The approach is based on introducing Allen’s interval algebra into PostgreSQL and is also applicable for video stream surveillance [PBH17]. While those efforts focus on languages and databases, *TPStream* targets event streams and efficient low latency and parallel processing methods.

Due to relating multiple types of situations to each other based on their point of occurrence, our evaluation method also relates to temporal joins (see [Gao+05] for an excellent survey). Even though most research is not based in stream processing, [PHD16] present a sweep-line algorithm for joins based on temporal *overlap* predicates. In order to evaluate interval relations, they use a specialized hash-index on intervals’ start and end. [DBG14] also target joins with *overlap* predicates, but focus on long-lasting intervals. The authors adaptively divide intervals into temporal partitions of different granularities while reducing the join to temporally overlapping partitions. Both works are orthogonal to *TPStream* since they optimize for overlap predicates on stored interval data. In contrast, *TPStream* derives intervals on the fly, allows for multiple arbitrary temporal relations, and combines both aspects for low latency pattern detection. In comparison to join algorithms on streams [Bab+04; GÖ03] as well as adaptive approaches [AH00], *TPStream* combines both, the derivation of situations and the detection of patterns. Thus, the operator can offer new techniques for early result detection unique to pattern matching in event streams.

5.3 Query Language

This section presents *TPStream*’s query language for specifying temporal pattern queries over point event streams. Therefore, we first present the basic structure of a *TPStream*-query. Then, we discuss every component in detail before we formulate the aggressive drivers query from the introduction as an example. Finally, this section is closed with a discussion on the expressiveness of the proposed language.

5.3.1 Syntax

The basic structure of the language is based on MATCH_RECOGNIZE [16] (items in square brackets are optional).

```
FROM          <input stream>
[PARTITION BY <attributes>]
```

DEFINE	<situation definitions>
PATTERN	<pattern definition>
WITHIN	<duration>
RETURN	<output definition>

The FROM clause selects the input stream for the query. If the physical input stream carries multiple logical partitions, where each of them should be evaluated separately, this is specified in the optional PARTITION BY clause. The partitioning scheme is specified via one or more <attributes> of the input stream, whereby every unique combination of attribute values defines a logical partition. To limit the search space for matches of the TP, a time window in which the pattern must occur completely is defined via the WITHIN clause. The duration is specified as <number> <time unit>, with time unit being one of second(s), minute(s), hour(s).

The DEFINE clause specifies which situation streams should be derived from the raw events and how to derive them.

```

<situation definitions> :=
    <situation definition> |
    <situation definition>, <situation definitions>
<situation definition> :=
    <name> AS <condition> [<duration constraint>]

```

A situation stream definition consists of a unique name and a condition. Conditions are constructed using boolean expressions (true, false, and, or, not), predicates (<, >, ≤, ≥, =, ≠) and arithmetic expressions composed of constant values and references to attributes of the input stream. The situations (time-intervals) of a situation stream are derived by applying those conditions to every incoming event. Optionally, a duration constraint can be set, allowing to restrict the length of derived situations.

```

<duration constraint> := AT LEAST <duration> |
                        AT MOST <duration> |
                        BETWEEN <duration> AND <duration>

```

If a temporal constraint is defined, only situations fulfilling this constraint are considered for matching the TP.

The PATTERN clause specifies the TP as a set of temporal constraints (TCs) between two situation types.



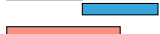
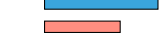


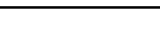
Relation (R)	Equivalent (R)	Visualization	Definition (δ_R)
A before B	B after A		$A.ts < A.te < B.ts < B.te$
A starts B	B started by A		$A.ts = B.ts < A.te < B.te$
A meets B	B met by A		$A.ts < A.te = B.ts < B.te$
A overlaps B	B overlapped by A		$A.ts < B.ts < A.te < B.te$
A during B	B contains A		$B.ts < A.ts < A.te < B.te$
A finishes B	B finished by A		$A.ts < B.ts < A.te = B.te$
A equals B			$A.ts = B.ts < A.te = B.te$

Table 5.1: Allen's Interval Algebra

```

<pattern definition> := <constraint> |
                        <constraint> AND <pattern definition>
<constraint>       := <name> <relations> <name>
<relations>        := <relation> |
                        <relation>;<relations>
    
```

The meaning here is as follows. A set of situations matches the TP if all defined constraints are satisfied. A constraint between situations from two streams is satisfied if their TR is among the specified relations. In other words, a constraint is a disjunction of TRs (delimited by ; in the query language). In order to express TRs between situations, we adopt Allen's Interval Algebra (see [All83]) depicted in Table 5.1 for two intervals A and B .

Every interval has a starting point (ts) and an ending point (te), resulting in four points. te is the first point in time when the interval is not valid, i.e., the interval is half-open. The relation between these four points (δ_R) defines the relation (R) between two situations A and B . As an example depicted in Table 5.1, A before B means the interval A ends before the interval B begins. Similarly, A during B means that $A.ts$ and $A.te$ are both within the interval B .

The last part of a *TPStream* query is the RETURN clause defining the shape of result events.

```

<output definition> := <output variable> |
                        <output variable>, <output definition>
<output variable>   := <aggregate> as <name>
<aggregate>         := FIRST(<ref>) | LAST(<ref>) |
                        COUNT(<ref>) | SUM(<ref>) | ...
    
```

Every attribute of the output event stream is an aggregation over an attribute of the input-stream. It summarizes all events that occurred during the time interval of a situation. The attribute reference (`<ref>`) is composed of the name of a situation stream and the name of an input attribute. Besides the standard aggregation functions like COUNT or SUM, *TPStream* allows to refer to values of the FIRST and LAST event participating the specified situation.

```

1 FROM      CarSensors PARTITION BY car_id
2 DEFINE    A AS accel > 8m/s2 AT LEAST 5s,
3           B AS speed > 70 mph BETWEEN 4s AND 30s,
4           C AS accel < -9m/s2 AT LEAST 3s
5 PATTERN    A meets;overlaps;starts;during B
6           AND B contains;finishes;overlaps;meets C
7           AND A before C
8 WITHIN    5 minutes
9 RETURN    first(B.car_id) AS id,
10          avg(B.speed) AS avg_speed;

```

Listing 5.1: Aggressive drivers query

Listing 5.1 shows the query definition for detecting aggressively driving cars from the introductory example. The query processes point events from the *CarSensors* stream which is partitioned by the *car_id* to evaluate every driver individually. We define the three situations for sharp acceleration (*A*), speeding (*B*) and hard braking (*C*), by referring to the acceleration and speed attributes of the input stream. All three definitions make use of duration constraints. The pattern allows for different combinations among the derived situation types. For instance, acceleration may *meet*, *overlap*, *start*, or *occur during* a phase of speeding. A match must occur completely within 5 minutes, and result events contain the unique *car_id* and the average speed during the speeding situation.

5.3.2 Expressiveness

As discussed in Section 2.3.5, sequential patterns are defined via regular expressions over symbols. Specific extensions, like aggregation, put the expressiveness of those languages between regular and context-free grammars [ZDI14]. However, only *ISEQ* provides a native way to process patterns based on TRs. This deficit is also reflected in the respective languages.

By design, *TPStream* can express all TRs (and unlike *ISEQ* alternatives among them) in a single query. In contrast, a single sequential pattern matching query detects a sequence, i.e., a *before* relation. Nevertheless, as shown by both straw

man’s approaches in Section 5.1, it is possible to express other TRs in systems supporting Kleene-closure. Thus, our language does not express more than the language of other systems.

Instead, we focus on enabling the user to express complex TPs in a single, readable and maintainable query via the widely-known interval algebra of Allen (Table 5.1). For this purpose, we made two notable design choices that differ from sequence-based approaches. First, in sequential pattern matching, events of the input stream may be ignored depending on the selected matching strategy. In contrast, *TPStream* derives the longest possible contiguous sequence of events because this aligns well with the idea of long-lasting situations and avoids ambiguity whether a situation is still ongoing during other events. Second, in some languages [DIG07] symbols can access aggregates of other symbols. Due to ambiguity in the expected results when dealing with situations, we do not allow this. For example, consider modifying the definition of symbol *B* in Listing 5.1 to “*B AS speed > max(A.speed)*”. Then, for *A overlaps B*, it is unclear whether to access $\max(A.speed)$ when *A* finishes, when *B* starts, or continuously monitor it for every *B*. We would also like to sketch that, apart from those concessions, it is possible to express a purely sequence-based pattern with *TPStream*: A sequence is expressed with a *before* relation, and the implicit ongoing nature of situations is eliminated with a duration constraint. However, for those cases highly optimized sequential pattern matching implementations [ZDI14] are preferable. In conclusion, this means that we do not change the expressiveness of other approaches. However, by extending a query language with Allen’s Interval Algebra, our benefits can be almost universally adopted.

5.4 Algebra

The goal in designing *TPStream* is to develop an operator capable of continuously deriving situations from a stream of events and relate those situations to each other. For this purpose, we first formally model those aspects in an algebra.

5.4.1 Derivation

Situations (cf. Definition 2.19) are derived from event streams through aggregation and predicate evaluation. We first define aggregation on continuous event subsequences before deliberating on predicates and how to derive situation streams.

Definition 5.1 (Aggregated Event Subsequence). An aggregate γ_{agg} is applied to an event stream subsequence $E_{[i,j]}$ by applying *agg* to the events of this subsequence.

$$\gamma_{agg}(E_{[i,j]}) := (agg(e_i, \dots, e_j), e_i.t, e_{j+1}.t)$$

When obvious from context, we abbreviate γ_{agg} with γ .

The result in Definition 5.1 technically already is a situation. However, for the derivation process as a whole, we want to discover situations for which a set of circumstances holds. In order to provide an unambiguous process to identify these situations, we are looking for the longest possible sequences for which these circumstances apply.

Definition 5.2 (Derived Situation). Situations are derived from event streams with the function $derive_{\phi, \eta, \gamma}$. It aggregates information of a contiguous subsequence of events $E_{[i,j]}$ by applying γ iff the events in $E_{[i,j]}$ are the longest possible sequence of events to fulfill a given predicate ϕ and the covered timespan is within the given duration constraint $\eta := [d_{min}, d_{max}]$:

$$derive_{\phi, \gamma, \eta}(E_{[i,j]}) = \begin{cases} \gamma(E_{[i,j]}) & \text{if } \forall l \in [i, j] : \phi(e_l) \wedge \\ & \neg(\phi(e_{i-1}) \vee \phi(e_{j+1})) \wedge \\ & (e_{j+1}.ts - e_i.ts) \in \eta \\ \langle \rangle & \text{otherwise} \end{cases}$$

Example. Assume the query in Listing 5.1 derives a speeding situation for a car with the time interval $[2, 10)$. This means $speed \leq 70 \text{ mph}$ at $t = 1$ and $t = 10$ and in between those timestamps $speed > 70 \text{ mph}$. From an algebraic standpoint, assuming knowledge about the whole event stream, this aligns well with a natural interpretation: There are not multiple situations (e.g. $[2, 3), [2, 4), \dots$) but rather one continuous speeding phase which fulfills the duration constraint ($d_{min} = 4s$ and $d_{max} = 30s$). For that reason and because it results in unique situations, we choose to derive the longest possible subsequence in Definition 5.2.

Definition 5.3 (Derived Situation Stream). The $deriveStream_{\phi, \gamma, \eta}$ function derives a stream of situations from a given event stream E by applying the function

$derive_{\phi,\gamma,\eta}$ to all possible subsequences and unifying the results:

$$deriveStream_{\phi,\gamma,\eta}(E) = \biguplus_j \biguplus_{i=1}^j derive_{\phi,\gamma,\eta}(E_{[i,j]})$$

Note that, because $derive_{\phi,\gamma,\eta}$ derives the longest situations possible, it is easy to show that $deriveStream_{\phi,\gamma,\eta}$ produces a stream of situations with disjoint time intervals. This important property means that the order of situations using start timestamps is the same as the order using end timestamps, resulting in a beneficial pattern for query processing as stated in [GÖ05].

5.4.2 Pattern Matching

TPStream matches multiple situation streams to a TP and produces a result event stream according to the given definitions. A TP is composed of TCs between situation streams, which in turn comprise multiple TRs between exactly two streams. This section presents formal definitions of these terms, the output of a successful match, and ultimately the *TPStream* operator.

Example. Consider the example query of Listing 5.1 and let s^A be an acceleration situation as defined by A and s^B, s^C be a speeding (B) and deceleration (C) situation, respectively. The pattern describes how pairs of situations can relate to each other via TCs. For s^A and s^B the TR can be either A meets B , A overlaps B , A starts B , or A during B . It does not matter if acceleration overlaps speeding or if speeding contains acceleration. Both cases may lead to the result of detecting aggressive drivers. The TP on the other hand, is a conjunction of TCs. In order to match the pattern, every TC must be fulfilled.

Definition 5.4 (Temporal Relation). Given two situation streams S^A, S^B , a TR $R^{A,B}$, defines a valid relationship between two situations $s^A \in S^A$ and $s^B \in S^B$ according to Allen's Interval Algebra (cf. Table 5.1). s^A and s^B fulfill $R^{A,B}$, iff they satisfy the corresponding algebraic definition (δ_R).

Definition 5.5 (Temporal Constraint). A TC $C^{A,B}$ between two situation streams S^A, S^B is a set of TRs $\{R_1^{A,B}, \dots, R_m^{A,B}\}$. Two situations $s^A \in S^A$ and $s^B \in S^B$ fulfill $C^{A,B}$, iff they at least fulfill one of the TRs.

In other words, TCs allow specifying multiple valid relations between two situation streams, providing the desired flexibility in expressing alternatives.

Definition 5.6 (Temporal Pattern). For a set of situation streams (S^1, \dots, S^m) , a TP $P = \{C^{i,j} | 1 \leq i < j \leq m\}$ is a set of TCs. A TP is matched by a *temporal configuration* $\bar{s} = (s^1 \in S^1, \dots, s^m \in S^m)$, iff \bar{s} satisfies every TC:

$$match_P(\bar{s}) :\Leftrightarrow \forall C^{i,j} \in P : \exists R^{i,j} \in C^{i,j} : \delta_{R^{i,j}}(s^i, s^j)$$

Definition 5.7 (Pattern Matching Output). A temporal pattern matching operator $PM_{w,\hat{\gamma},P}$ matches a *temporal configuration* $\bar{s} = (s^1, s^2, \dots, s^m)$ to a TP P . It aggregates the information of \bar{s} with some suitable aggregate $\hat{\gamma}$ and checks the *window* condition (cf. WITHIN clause).

$$window(\bar{s}, w) = w \geq \max_{s \in \bar{s}}(s.te) - \min_{s \in \bar{s}}(s.ts)$$

The operator produces an output if the *temporal configuration* matches the pattern during the specified window, i.e.:

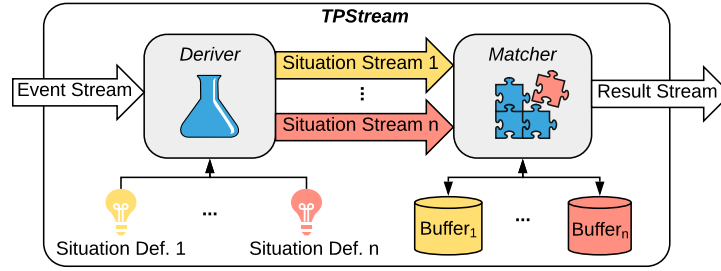
$$PM_{w,\hat{\gamma},P}(\bar{s}) := \begin{cases} \langle (\hat{\gamma}(\bar{s}), \max_{s \in \bar{s}} s.te) \rangle & \text{if } match_P(\bar{s}) \wedge window(\bar{s}, w) \\ \langle \rangle & \text{otherwise} \end{cases}$$

Similarly to how we extended derived situations to derived situation streams (Definition 5.2 to 5.3), we can extend Definition 5.7 to situation streams.

Definition 5.8 (TPStream). $TPStream_{w,\hat{\gamma},P}$ matches multiple situation streams S^1, \dots, S^m to a TP P by applying the corresponding pattern matching operator $PM_{w,\hat{\gamma}}$ to the cross product of the situation streams and unifying the results:

$$TPStream_{w,\hat{\gamma},P}(S^1, \dots, S^m) := \bigsqcup_{\bar{s} \in \times_{i=1}^m S^i} PM_{w,\hat{\gamma},P}(\bar{s})$$

Note that $TPStream_{w,\hat{\gamma},P}$ results in an event stream and is thus easily integrable into existing point-based ESP systems.

Figure 5.2: *TPStream* Architecture

5.5 Algorithms & Implementation

In this section, we present our algorithms and implementation details for detecting TPs among streams of point events. Following the definitions from the previous section, the general architecture consists of two main components, as depicted in Figure 5.2. First, the deriver component consumes events from the input stream and derives the defined situation streams. Then, it passes those streams to the matcher component, which performs the actual pattern matching. In the following two subsections, we will explain both components in detail. For the sake of simplicity, we defer the discussion on low latency detection to Section 5.5.3 and wait for the end timestamp of derived situations before invoking the matcher. The last part of this section describes how *TPStream* computes efficient execution plans and dynamically adapts to changing workloads.

5.5.1 Deriving Situations

Definition 5.3 introduces derived situation streams, using knowledge about the whole input-stream. To compute situation streams incrementally as new events arrive, the deriver component manages a buffer for ongoing situations (*Buf*) and the situation stream definitions (*Def*). Algorithm 5.1 shows how they are used to derive situations on-the-fly. For every defined situation, the deriver handles 3 cases. First, if there is no started situation in the buffer, but the predicate holds, it starts a new situation. Therefore, it computes initial values for all defined aggregates (e.g., $p.speed$ for an $\max(speed)$ aggregate) and stores them in the buffer together with the event's timestamp (Lines 4,5). Second, if the buffer contains a started situation and the current event fulfills the predicate, the deriver prolongs the temporal validity of the situation. In addition, it updates the

Algorithm 5.1: DeriveSituations

Input: (p, t) : event
Data: $Buf := [(p', ts)_i]$: active situation buffer
Data: $Def := [(\phi, \gamma, \eta)_i]$: situation definitions

```

1  $Res \leftarrow \emptyset$ ;
2 foreach  $i \in |Def|$  do
3    $(\phi, \gamma, \eta) \leftarrow Def[i]$ ;
4   if  $Buf[i] = \emptyset \wedge \phi(p)$  then
5      $Buf[i] \leftarrow (initAgg(p, \gamma), t)$ ;
6   else if  $\phi(p)$  then
7      $updateAgg(p, Buf[i], \gamma)$ ;
8   else if  $Buf[i] \neq \emptyset$  then
9     if  $(t - Buf[i].ts) \in \eta$  then
10       $Res \leftarrow Res \cup \{(Buf[i].p', Buf[i].ts, t)\}$ ;
11       $Buf[i] \leftarrow \emptyset$ ;
12 if  $Res \neq \emptyset$  then
13    $updateMatcher(Res, t)$ ;
    
```

buffered aggregates using the event's payload (p) (Lines 6,7). Finally, the deriver finishes a buffered situation on the first event not satisfying the defined predicate. In this case, it fixes the situation's end timestamp to the current time, adds it to the result set Res (provided it satisfies the duration constraint η), and clears the corresponding buffer slot (Lines 8-11). After updating the state of every situation stream, the deriver passes the result set to the matcher component (Lines 12,13).

5.5.2 Matching the Pattern

The matcher implements an incremental version of $TPStream_{w, \hat{\gamma}}$ (Definition 5.8). In other words, it detects matches on the fly as the deriver passes in new situations. The general idea is to manage a buffer for every situation stream and perform the pattern detection via a multi-way join between those buffers, using the TCs as join-conditions. Recap that all situations within a stream are disjoint and thus imply the same order on both the start and end timestamps (Definition 5.3). We will use this fact to ensure the efficient execution of the matcher component.

Algorithm 5.2: UpdateMatcher

Input: \mathcal{S}_F : set of finished situations
Input: t : the current time
 1 `purgeBuffers(t);`
 2 **foreach** $s \in \mathcal{S}_F$ **do**
 3 `addToBuffer(s);`
 4 $(\mathcal{C}, Buf, nextStep) \leftarrow \text{getEvaluationOrder}(s)$;
 5 `performMatch($\{s\}, (\mathcal{C}, Buf, nextStep)$);`

Algorithm 5.3: PerformMatch

Input: ws : partial result
Input: $step : (\mathcal{C} : \text{Temporal Constraints}, Buf : \text{Situation Buffer}, nextStep)$
 1 **if** $step = \emptyset$ **then**
 2 `publishResult(ws);`
 3 **else if** $\text{containsSituationForStep}(ws, step) \wedge \text{checkConstraints}(ws, \mathcal{C})$ **then**
 4 `performMatch($ws, nextStep$);`
 5 **else if** $\neg \text{containsSituationForStep}(ws, step)$ **then**
 6 **foreach** $(p, t_s, t_e) \in \text{findMatches}(\mathcal{C}, Buf, ws)$ **do**
 7 `performMatch($ws \cup \{(p, t_s, t_e)\}, nextStep$);`

Every time the deriver returns a non-empty set of finished situations, we invoke Algorithm 5.2. At first, the algorithm purges all expired situations from the buffers (Line 1) by removing all situations s with $s.ts < t - window$. Because of the mentioned ordering, this effectively means finding the first situation s' with $s'.ts \geq t - window$ and discarding all previous situations. We implement the buffers via array-backed ring buffers to efficiently support these operations. After purging outdated situations, the matcher executes the following steps for every received situation. First, it adds the situation to the corresponding buffer (Line 3). Then, it looks up the evaluation order for the given situation (Line 4). That is the order in which the matcher joins the situation buffers. Then, for every join-step, the matcher stores the corresponding TCs (\mathcal{C}), the situation buffer to join (Buf), and a reference to the next step. Finally, it calls the recursive matching algorithm (Algorithm 5.3). Note that the currently processed situation s is fixed in the parameter to the recursive algorithm. This ensures unique results because it forces s to be part of every generated result and s changes with every invocation of the

algorithm.

Algorithm 5.3 matches the pattern as follows. Every recursive step joins one situation buffer with the current partial result (ws). Here, two cases must be distinguished. The first case handles situations given as a parameter from Algorithm 5.2. In this case, the algorithm omits to join the entire buffer, checks the TCs of the current step (\mathcal{C}), and proceeds recursively if the constraints are fulfilled (Lines 3,4). The second case joins ws with the situation buffer (Buf) associated with the current step. Then, the algorithm recursively proceeds to the next step for every join result (i.e., every matching situation) (Lines 5-7). After processing all steps, ws contains one situation from every buffer, and all TCs are satisfied for this set of situations. Hence, the algorithm calls *publishResult* to materialize a result event and push it into the output stream (Lines 1,2).

Obviously, the evaluation performance of Algorithm 5.3 mainly depends on the efficiency of the function *findMatches*. A naïve approach would be, to scan the entire situation buffer and check the TCs for every situation separately (i.e., perform a nested-loop join). Let n denote the number of recursive steps, Res_i the i -th intermediate result, and Buf_i the situation buffer traversed in step i . With $|Res_1| = |Buf_1|$, the costs of the naïve approach (C_{naive}) can be estimated with:

$$C_{naive} = |Res_n| + \sum_{i=1}^{n-1} |Res_i| \cdot |B_{i+1}| \quad (5.1)$$

To speed up the computation, we again utilize the order of situation streams. Because the buffers reflect the temporal order, we can find all matching situations using binary searches. We first discuss this for a single TR before extending it to (multiple) TCs. Recall that a TR explicitly defines a relationship between all four endpoints of two situations. For instance, this is $A.ts < B.ts < A.te < B.te$ for A overlaps B . Now, given an instance of situation A , we obtain matching instances of B by (i) issuing two range-queries on the buffer of B , using the timestamps of A as boundaries and (ii) intersecting the results of those queries. For the example relation, these queries are:

1. $A.ts < ts < A.te$ for the start-timestamp and
2. $A.te < te < \infty$ for the end timestamp.

It is easy to see that every situation falling into both ranges fulfills the given TR. Figure 5.3 illustrates this using three situations. Situation A_1 in combination with the TR is used to build the two search ranges. After intersecting the results ($\{B_1\}$

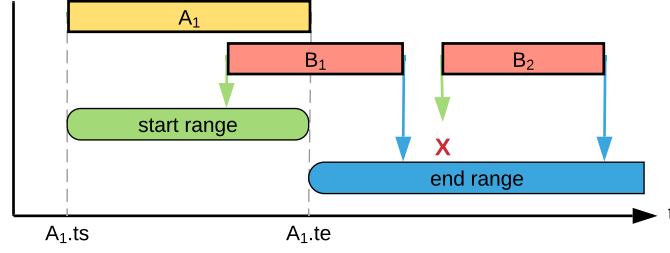


Figure 5.3: Temporal Matching via Range Queries

for the start range and $\{B_1, B_2\}$ for the end range), we receive our final result B_1 . Note that for TRs allowing more than one result (e.g., A before B), this strategy additionally eliminates the need for checking every combination individually.

In general, a TC contains more than one TR, stating each of them as a valid relationship between two situations. Multiple TRs can easily be integrated by executing the search separately for each of them and subsequently building the union of the obtained results. Similarly, the conjunction of multiple TCs is implemented as an intersection of the results from the respective individual queries. Because a contiguous array backs the buffers, we can represent the search results as index ranges and thus efficiently compute the required unifications and intersections. This approach reduces the estimated costs of Algorithm 5.3 to:

$$C_{binary} = \sum_{i=2}^n (|Res_{i-1}| \cdot |Res_i| + C_{findMatches}(Buf_i)), \text{ with} \quad (5.2)$$

$$C_{findMatches}(Buf_i) = \sum_{C \in \mathcal{C}_i} \sum_{R \in C} 4 \cdot \log_2(|Buf_i|) \quad (5.3)$$

$C_{findMatches}(Buf_i)$ is composed as follows. For a single TR, we need to execute 4 binary searches on Buf_i . Thus, the cost for one TR is $4 \cdot \log_2(|Buf_i|)$. In every recursive step, we consider $|\mathcal{C}_i|$ TCs, each composed of $|C|$ TRs, which leads to the sum expression in Equation 5.3.

Relation (R)	Definition (δ_R)	$tr_{min}(R)$	Prefix-Group (G)	$tg_{min}(G)$
A before B	$A.ts < A.te < B.ts < B.te$	$B.ts$	$A.ts < A.te \leq B.ts$	$B.ts$
A meets B	$A.ts < A.te = B.ts < B.te$	$B.ts$		
A starts B	$A.ts = B.ts < A.te < B.te$	$A.te$		
A equals B	$A.ts = B.ts < A.te = B.te$	$A.te = B.te$	$A.ts = B.ts$	$B.ts$
A started by B	$A.ts = B.ts < B.te < A.te$	$B.te$		
A overlaps B	$A.ts < B.ts < A.te < B.te$	$A.te$		
A finishes B	$A.ts < B.ts < A.te = B.te$	$A.te = B.te$	$A.ts < B.ts$	$B.ts$
A contains B	$A.ts < B.ts < B.te < A.te$	$B.te$		

Table 5.2: Temporal relations R and their prefix groups G with their earliest detection times $tr_{min}(R)$ and $tg_{min}(G)$

5.5.3 Low-Latency Matching

In this section, we determine the earliest points in time $tr_{min}(R)$, $tc_{min}(C)$, $tp_{min}(P)$ to detect a TR R , a TC C , and a TP P , respectively. Then, we illustrate cases in which the algorithms of Section 5.5.2 in combination with low-latency matching fail to deliver correct results. In the last part of this section, we present adjusted algorithms for low-latency matching.

Analysis

Two situations A and B can only be related once we know they exist, making $\max(A.ts, B.ts) \leq tr_{min}(R)$ a trivial lower bound for all relations R . For an exact computation of $tr_{min}(R)$ we consider the definition δ_R of relations given in Table 5.2. Let $t_1 \leq t_2 \leq t_3 \leq t_4$ be the timestamps in the order they appear in δ_R . It is easy to see that the ordering of t_4 is already available at t_3 , because $t_3 \leq t_4$ and there are no timestamps after t_4 . Furthermore, multiple relations are sharing the same definitions up to t_2 , i.e., it is not possible to distinguish those relations from each other. We group these relations with a common prefix of two timestamps into so-called prefix groups as shown in Table 5.2 for relations starting at situation A (an analogous definition exists for relations starting with situation B). Thus, we conclude that the following holds for any TR R

$$tr_{min}(R) = t_3.$$

A TC $C = (R_1, \dots, R_n)$ for two situations A, B matches if one of the contained relations is fulfilled. Thus, C has multiple earliest detection times given by a

set

$$tc_{min}(C) = \{tr_{min}(R_1), \dots, tr_{min}(R_n)\}.$$

Furthermore, if C contains all relations of a prefix group G (cf. Table 5.2), the detection time of these relations is shifted to the trivial lower bound of that group (denoted $tg_{min}(G)$).

Finally, for a pattern $P = (C_1, \dots, C_m)$, every constraint $C_i = (R_1^i, \dots, R_{n_i}^i)$ must be matched. However, a single *temporal configuration* matching P fulfills exactly one TR ($R_{j_i}^i \in C_i, i = 1, \dots, m$) from every constraint, making

$$tp_{min}(P) = \{max(tr_{min}(R_{j_1}^1), \dots, tr_{min}(R_{j_m}^m)) \mid 1 \leq j_i \leq n_i, 1 \leq i \leq m\}.$$

Thus, $tp_{min}(P)$ is among all the constraint detection points, i.e., $tp_{min}(P) \subseteq \bigcup_{i=1 \dots m} tc_{min}(C_i)$.

Problem Statement

For the ease of presentation, we first postpone discussing optional duration constraints on situations and prefix groups to the end of this section. In general, our low-latency analysis provides two insights for the matching algorithm. First, new matches only occur if a new situation starts or a situation ends. Second, only a subset of the situations in a pattern P can produce a match at a point in $tp_{min}(P)$. Thus, we delay the matching process until a situation with at least one endpoint in $tp_{min}(P)$ occurs without affecting the latency. We call those situations trigger situations since only they trigger a call to Algorithm 5.3. These insights affect our algorithms in the following ways. First, situations must be available for matching from the start. Therefore, we adjust the deriver component. Additionally, we need to determine for each situation stream if the derived situations are trigger situations. For trigger situations, we need to compute the point in time to execute Algorithm 5.3 (at its start, end, or both).

In contrast to our algorithms so far, the following two cases must be considered during the matching process to ensure the correctness of the produced results while delivering them as early as possible. First, we ensured unique results by examining a new situation on every invocation of Algorithm 5.3. However, because for low latency matching both endpoints of a situation must be considered, the algorithm might be called twice with the same pre-set situation. Hence, additional steps are required to guarantee unique results. Second, situations whose endpoint is unknown always carry the current time as a temporary end timestamp. This affects the matching of TRs requiring both situations to end at the

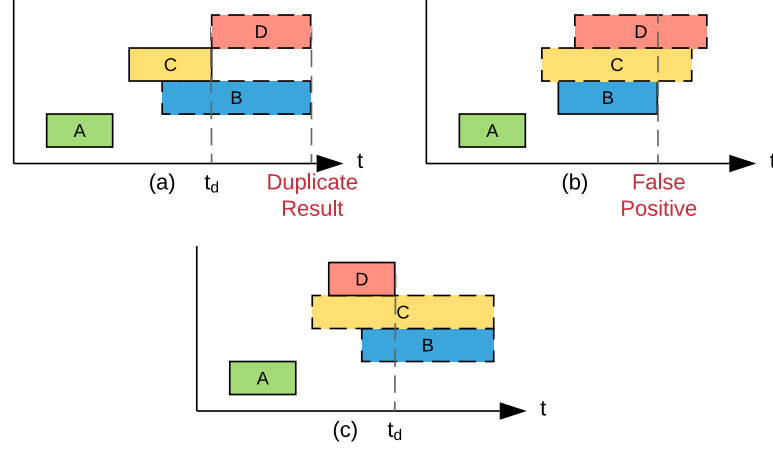


Figure 5.4: Earliest detection time ($tp_{min}(P)$) of different *temporal configurations* for the sample pattern P

same point in time (i.e., *finishes*, *equals*). If the matcher inspects two ongoing situations, they might fulfill the respective TR even though their actual end timestamps differ. Hence, in such cases, the matcher produces false positive matches. Note that this phenomenon does not affect other relations since they require their end timestamps to differ, which never holds for two ongoing situations. We illustrate both cases using the following example pattern P on four situations (A, B, C, D) .

*A before B AND A before C AND A before D AND
C contains; finishes; meets D*

It defines A as the starting point of every match. B is not explicitly related to either C or D and is required to happen after A . Consequently, B is a *trigger* situation and $B.ts \in tp_{min}(P)$. For D , both timestamps, $D.ts$ (via *meets*) and $D.te$ (via *contains*, *finishes*), are in $tp_{min}(P)$.

Figure 5.4 (a), show a *temporal configuration* with $D.ts$ as the earliest point of detection for P . However, since $D.te \in tp_{min}(P)$, the same match would be detected again at $D.te$. Moreover, Figure 5.4 (b) showcases the false positive detection if validating C *finishes* D would succeed at $B.te$. Finally, Figure 5.4 (c) illustrates that two ongoing situations like B and C may be part of a match, even if they are not explicitly related via a TC.

Low-Latency Matching Algorithm

Instead of handling the above cases explicitly, our low latency algorithm avoids them by ensuring a unique combination of situations in the partial result before passing it to the matching algorithm. In particular, this means the algorithm manages started situations in a separate buffer that is inaccessible to the matching algorithm. It uses this extra buffer to build all valid situation combinations upfront (i.e., all combinations of started situations, not explicitly related to the current one). Furthermore, to avoid duplicate results, we exploit the following fact. TRs that enforce matching on a situation's start require the second situation to be finished in the past. On the other hand, TRs triggering matching on a situation's end require the second situation to be either started (and not yet finished) or finished at the same time (cf. Table 5.2). Consequently, manually adding the started counterpart to the partial result before executing the matching algorithm on a situation's end ensures the uniqueness of the produced results.

The details are presented in Algorithm 5.4. After purging outdated situations from the buffers (Line 1), the algorithm adds each started situation (s) to the extra buffer. Then, if $s.ts \in tp_{min}(P)$, the algorithm pre-sets s in the partial result and performs a regular match (Lines 2-6). Furthermore, if there are *started and unrelated* situations, it additionally performs matches with s and every combination of them (Lines 7-9). This accounts for configurations as seen in Figure 5.4 (a). In the next step, Algorithm 5.4 migrates all finished situations from the separate buffer to the regular situation buffer (Lines 10-12) and triggers the matching process if $s.te \in tp_{min}(P)$ (Lines 13,14). This time with combinations of s and all *started and related* situations (Lines 15-17), further combined with all *started and unrelated* situations (Lines 18-20), which fuses the avoidance of duplicate results and false positives. Figure 5.4 (c) shows an example for this case. Note that the actual constraint-checking among the created combinations is performed by the call to *performMatch* (Algorithm 5.3) since it is aware of pre-set situations in the partial result. As we will show in Section 5.7, the extensive building of combinations has only minimal impact on the runtime performance because it shifts load from joining to the update algorithm and does not introduce additional computation steps.

Duration Constraints. Only a few modifications are required to incorporate duration constraints on situations into low latency-matching. First, if a maximum duration constraint is defined (regardless of a possibly specified minimum duration), the corresponding situation must not be included in the matching process

Algorithm 5.4: Low-Latency MatcherUpdate

Input: S_f, S_s : sets of finished/started situations
Input: t : current time

```

1  purgeBuffers( $t$ );
2  foreach  $s \in S_s$  do
3      startedBuffer.add( $s$ );
4      if  $matchOnStart(s)$  then
5           $order \leftarrow getEvaluationOrder(s)$ ;
6          performMatch(  $\{s\}, order$  );
7           $U \leftarrow getUnrelatedStarted(s)$ ;
8          foreach  $u \in powerset(U) \setminus \emptyset$  do
9              | performMatch(  $u \cup \{s\}, order$  );
10 foreach  $s \in S_f$  do
11     startedBuffer.remove( $s$ );
12     addToBuffer( $s$ );
13     if  $matchOnEnd(s)$  then
14          $order \leftarrow getEvaluationOrder(s)$ ;
15          $R \leftarrow getRelatedStarted(s)$ ;
16         foreach  $r \in powerset(R) \setminus \emptyset$  do
17             | performMatch(  $r \cup \{s\}, order$  );
18             |  $U \leftarrow getUnrelatedStarted(s)$ ;
19             | foreach  $u \in powerset(U) \setminus \emptyset$  do
20                 | | performMatch(  $r \cup u \cup \{s\}, order$  );
    
```

until its end is known. Hence, we exclude these situations from the set of started situations (S_s). Furthermore, if their start timestamp is in $tp_{min}(P)$, we defer the matching to their end timestamp. Second, if a minimum but no maximum duration is defined, we defer the inclusion into the set of started situations until the constraint is satisfied. This possibly implies the inclusion of its deferred start timestamp (\overline{ts}) into $tp_{min}(P)$. As an example, consider A during B and the following order of timestamps: $B.ts < A.ts < A.te < B.\overline{ts} < B.te$. This match can not be detected at $A.te$, because B 's duration does not exceed the lower bound at this point. Hence $B.\overline{ts}$ requires a matcher invocation.

Prefix Groups. In order to handle prefix groups, we relax the restriction that two started and explicitly related situations must not be matched. In particular, we

perform the matching if the corresponding TC contains one or more prefix groups. However, to still omit false positives, the matcher must distinguish between prefix group and regular detection. In our algorithm, this means splitting the TC into two disjoint sets, one containing all TRs forming a prefix group and another one for the remaining relations. We use the first set to match a situation's start while we use the second on its end.

5.5.4 Computing the Evaluation Order

The matcher component maps the problem of temporal pattern matching to a multi-way join between situation buffers. Like join processing in traditional DBMS, the performance of joining depends heavily on the order in which we execute the join operations. This section discusses how the matcher's evaluation order is computed and presents the cost model used during this process.

Analogous to classical join processing, our optimizer enumerates possible execution plans, computes their expected computational costs, and determines the plan with minimum cost. We do not provide multiple implementations of the join operator so that enumerating possible plans reduces to the enumeration of possible evaluation orders. To further reduce the number of plans to consider, we exclude orderings joining a situation buffer without an applicable TC. In other words, we omit plans involving the calculation of a cross-product.

According to Equation 5.2, estimating the costs for a given plan boils down to estimating the size of intermediate results.

$$|Res_i| := \begin{cases} |Buf_1| & \text{if } i = 1 \\ |Res_{i-1}| \cdot |Buf_i| \cdot \varphi_i & \text{otherwise} \end{cases} \quad (5.4)$$

φ_i denotes the selectivity of the applicable TCs in step i (\mathcal{C}_i), which can be composed from the selectivities of the contained TRs as follows.

$$\varphi_i := \prod_{C \in \mathcal{C}_i} \left(\sum_{R \in C} \varphi_R \right) \quad (5.5)$$

When initially deploying a query into the system, the situation buffers are empty, and we have no estimation on the selectivity of the TCs. Hence, we assume the selectivities depicted in Table 5.3, which are based on the following back-of-the-envelope calculation. The combined selectivity of all possible relations should be

Relation	Selectivity
before	0.445
during	0.03
overlaps	0.01
starts, finishes, meets	0.0049
equals	0.0006

Table 5.3: Initial estimates for the selectivity of TRs (φ_R)

100%. When assuming equal-sized buffers and temporally uniformly distributed situations, the selectivity of a *before* relation will be around 50%. For *during*, the number of results is limited by the maximum of both buffer sizes because a situation A can happen during at most one other situation B , but B may contain multiple A situations. All other TRs define a 1:1 relationship, which limits the worst case to the minimum of both buffer sizes. As seen in Table 5.3, we additionally separate the last case by the number of stated equalities in δ_R . Note that even though this is an initial estimate, the resulting plans prove to work well in most cases (cf. Section 5.7.4).

Adaptivity

A query in an ESP system is typically active for a long time. Hence, more important than the quality of an initial execution plan is to tune this plan and adapt it to changing workloads. To do so, we keep track of the buffer sizes and selectivities imposed by TCs during execution. The buffer sizes are available at any time and at no cost since the underlying data structure tracks them. However, to smooth out potential spikes, we monitor the buffer size using an exponential moving average (EMA), which is adjusted after each call to the matcher’s update method as follows.

$$EMA_i = \alpha \cdot |Buf_i| + (1 - \alpha) \cdot EMA_{i-1} \quad (5.6)$$

EMA_i holds after the i -th update. $|Buf_i|$ denotes the size of the considered buffer at update i and the smoothing factor $\alpha \in (0, 1)$ determines how much weight is given to previous values. For example, a value close to 1 assigns almost no weight to older values, while a value close to 0 decreases the influence of new values. We also manage the selectivities of the TCs with EMAs using one EMA-value per constraint.

The active plan stores a snapshot of the statistics it is based on. Then, after every update, we compare those stored statistics to the current values. If any of

them differs by more than a defined threshold, we trigger a re-computation of the evaluation order.

Finally, if a plan migration is required, we can migrate to the new plan between any two invocations of the matcher component. Because the matcher does not store any intermediate results but solely relies on the situation buffers, this switch comes without any additional migration costs. As we will show in Section 5.7.4, the total costs for adaptivity are negligible.

5.6 Parallel TPStream

This section presents a parallel version of *TPStream* to improve throughput by leveraging the parallel processing power of today's multicore systems without introducing any form of latency degradation. The nature of *TPStream* makes its parallelization a non-trivial task for the following two reasons: First, *TPStream* works with continuous sequences of situations derived from continuously arriving events. Thus, to identify a pattern P , either a single thread must process all events leading to the detection of a match for P , or the active threads need to share the state information. Second, a key component of *TPStream* to achieve low latency is its situation buffer. However, in a parallel version, the effort involved in synchronizing the accesses of different threads to the buffer can become a severe performance bottleneck.

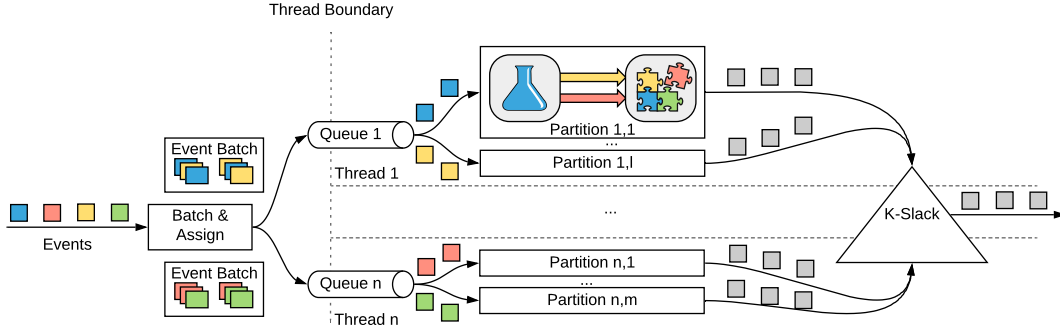
In the following, we present the essential concepts for introducing parallelism to *TPStream*. First, we start with the static integration of *TPStream* into the application context from which we receive incoming event streams and to which we send the results of continuous pattern queries. We then develop two multi-threaded versions of the *TPStream* query processing algorithm. The first version targets partitioned input streams, while the second version applies to non-partitioned ones. In fact, parallelizing the latter case is more complex and requires a finely tuned processing pipeline, which we describe in detail in separate subsections. Next, we address the problem of the fluctuating behavior of event streams (e.g., changes in data rates) and present an auto-tuning component that overcomes the limitations of our static approaches presented so far. Finally, at the end of this section, we present an extension of parallel *TPStream* for a distributed streaming environment and show how to integrate *TPStream* into a distributed streaming platform.

5.6.1 Integration with Application Context

Before we go into the details of parallel processing, we discuss the import of event streams and the export of result streams. In a parallel environment, the processing threads must be decoupled from both the (external) event producers and the result consumers.

For the decoupling from event producers, we deploy queues on the thread boundaries in which the producer is writing events while processors are reading from the queues. Because the synchronization of producers and processors introduces a reasonable overhead, we decided to access the queues in event-batches of fixed size rather than single events. The batch size is initially constant, but the auto-tuning component (see details in Section 5.6.4) can adjust it if changes occur in the application context and system load. Furthermore, the size of a queue is constant per active thread, resulting in a stable and predictable memory footprint. In our experiments, we learned that the capacity of a queue should be 2^{16} events per active thread. Overload is propagated upstream via back pressure allowing for countermeasures on the producer side (e.g., writing events into logs). In general, the goal is to set the batch size in was that minimizes the waiting time on the producer and consumer sides. We experimentally examined various settings and found that a batch should be a power of two, at least 32 and at most 2^{15} (i.e., half of the number of events in a queue). In the extreme case of two batches, the producer writes in its batch, while a processor reads from the other batch. Depending on the parallelization approach, we either use a shared queue for all threads or a dedicated queue per thread.

Additionally, the processing threads need to be decoupled from the result consumers. In particular, we have to treat out-of-order results which are likely to occur during parallel processing. Consider for example the following (simplified) scenario of two processing threads ($pt_1; pt_2$) that process two consecutive event-batches ($b_1 \rightarrow pt_1; b_2 \rightarrow pt_2$) in parallel. Because all events in b_1 have smaller timestamps than events in b_2 , the results of pt_2 will have timestamps greater than those of pt_1 . However, if pt_2 finishes first and sends its results downstream before pt_1 , the merged result stream does not have a monotonically increasing time order anymore. A naïve solution to this problem is to wait for a result from every processing thread and only forward the result with the smallest timestamp at a time. However, this may lead to unpredictable delays and block all processing threads if one of them is not producing any results. Therefore, we use a K -slack [BSW04] working as follows. A min-heap of fixed size (K events) collects the results. The first K events are simply stored in the heap. From event $K + 1$, the new result is put into the heap, while the top of the heap (i.e., the event with the smallest

Figure 5.5: Overview partition parallel *TPStream*

timestamp) is removed and sent downstream. This method avoids the blocking of threads during result propagation at the expense of possible out-of-order results. If a result with a timestamp smaller than the top of the heap arrives, we are unable to decide whether it is out-of-order or not. However, depending on the downstream operators, we can either publish it because the downstream operator can handle it or discard it and report a warning. The probability of this case occurring depends on the parameter K , which needs to be chosen according to the (latency) demands of the concrete application scenario.

5.6.2 Partitioned Data

TPStream's query language features a `PARTITION BY` clause (cf. Section 5.3), which allows to divide the input stream into several logical partitions. Those partitions are evaluated independently by applying the *TPStream* operator to each of the partitions. For example, in the aggressive drivers query, every car is analyzed separately. Thus, the input stream gets partitioned by the `car_id` attribute. Therefore, even in the single-threaded implementation, *TPStream* maintains a processing pipeline including a dedicated matcher and deriver for every partition. Given this, parallel processing of a partitioned stream is straightforward, as shown in Figure 5.5. We assign partitions to processing threads in a round-robin fashion. Every thread has its input queue fed with batches containing only the relevant events for the assigned partitions. Therefore, the producer thread maintains a *working-batch* per processing thread and assigns incoming events accordingly. Once the *working-batch* reaches the configured batch size, the producer puts it into the corresponding queue and starts a new *working-batch*. The working threads consume the batches from their queues and process them event-by-event analogous to the

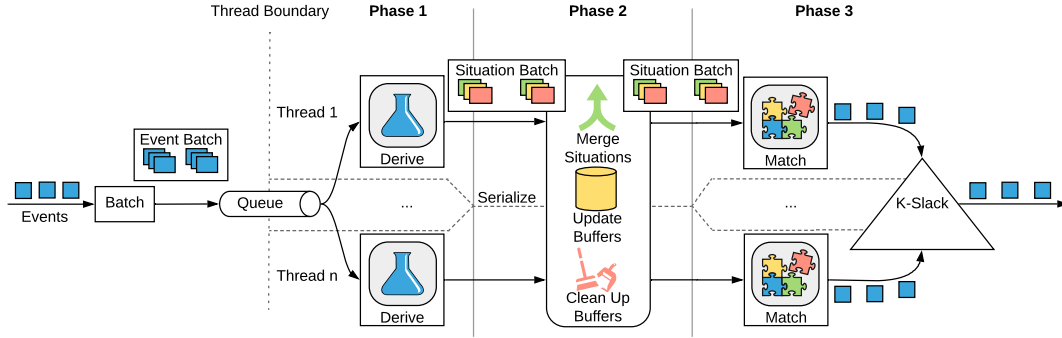


Figure 5.6: Overview: parallel processing of unpartitioned data

single-threaded case. The only difference here is the output handling. Instead of sending results downstream directly, they go through the K -slack deployed at the end of the pipeline.

To illustrate partition parallel processing, consider the following event trace.

$$\langle e_1[t = 1, part = 1], e_2[t = 2, part = 2], e_3[t = 3, part = 3], e_4[t = 4, part = 2], \\ e_5[t = 5, part = 4], e_6[t = 6, part = 1], e_7[t = 7, part = 3], e_8[t = 8, part = 4] \rangle$$

It shows eight events, each with a timestamp t and a partition attribute $part$. Two processing threads handle the four logical partitions with a batch size of four events. This results in two batches created by the producer: $b_1 = (e_1, e_3, e_6, e_7)$ and $b_2 = (e_2, e_4, e_5, e_8)$. The producer assigns b_1 to the first and b_2 to the second processing thread. In turn, each of them maintains two *TPStream* instances, one per assigned partition. The processing threads then route the events of their batches to the corresponding partition and process them analogous to the single-threaded case.

5.6.3 Unpartitioned Data

In the following, we consider that all processing threads need to work cooperatively on all input data. This case occurs if there is no `PARTITION BY` clause. Figure 5.6 gives a brief overview of how multiple threads process unpartitioned data. Similar to partition-based processing, we collect the incoming events into batches of fixed size. The difference here is that there is only one shared queue serving all processing threads. Thus, the producer simply slices the incoming event stream into fixed-size batches by maintaining a single working batch instead of multiple ones. Every

processing thread handles one event batch at a time, each passing three phases: deriving, synchronization, and matching. Here, the second phase is the only synchronization point between the processing threads.

In the first phase, the deriver generates situations from the current event batch. Phase two first analyzes the derived situations of the previous batch and merges situations spanning the batch boundary. Then, the situation buffers are updated by inserting new situations and purging outdated ones. Finally, phase three performs the actual pattern matching. Like in the partitioned case, we employ a K -slack at the end of the pipeline to produce an ordered result stream.

Before detailing each of these phases, we illustrate this method with another example. Consider $TPStream$ with two situation definitions (A, B) , a batch size of 4 events, two processing threads (pt_1, pt_2) and the following event trace of eight events:

$$\langle e_1[t = 1], \dots, e_8[t = 8] \rangle$$

The producer generates two event batches $b_1 = (e_1, \dots, e_4)$ and $b_2 = (e_5, \dots, e_8)$. Let thread pt_1 processes b_1 and pt_2 processes b_2 in parallel. Furthermore, assume that both of them finish the deriving phase at the same time returning the following situation batches:

$$\begin{aligned} sb_1 &= \{ A_1 = [2, 4), B_1 = [4, 5) \} \\ sb_2 &= \{ A_2 = [7, 8), B_2 = [5, 6) \} \end{aligned}$$

For ease of presentation, we represent derived situations by their validity intervals only. Phase two synchronizes the execution because $TPStream$ relies on the temporal ordering of the situation buffers. Hence, pt_1 is permitted first because all situations in sb_1 happen before the situations in sb_2 . While A_1 can be added to the buffer directly, B_1 is a cross-batch candidate: the last valid timestamp of B_1 is equal the timestamp of e_4 – the last event in b_1 . Thus, it is possible that the situation continues with the next batch and is put aside for further checking. Then, pt_1 removes outdated situations from the buffers and moves on to the third phase. Next, pt_2 enters phase two and adds A_2 to the buffer. It then checks the set-aside B_1 and merges it with B_2 because B_2 starts with the first event of b_2 , and hence the situation would span e_4 and e_5 in sequential processing. After buffering the merged situation $B_{1,2} = [4, 6)$ and purging outdated situations, pt_2 continues with phase three.

The following describes the three phases in detail and shows how to keep the syn-

chronization overhead between processing threads at a minimum.

Deriving

In the deriving phase, a processing thread fetches an event batch from the shared queue and applies all situation definitions to the events of that batch (cf. Algorithm 5.1). Then, it stores the resulting situations in a situation batch. Besides the derived situations, a situation batch carries a sequence number, head, and tail information. The sequence number is required to update the situation buffers (i.e., maintain the temporal ordering of situation streams). The head and tail information is required to merge situations spanning more than one batch. They contain all ongoing/partial situations after evaluating the current batch's first and last events, respectively.

Synchronization

After a processing thread creates a situation batch, it enters the synchronization phase. As mentioned above, this phase serializes the execution to guarantee the temporal ordering among situation buffers. In order to enforce the correct execution order among all threads, we use the sequence number of situation batches. For example, a processing thread holding a batch s may only enter this phase if the thread responsible for batch $(s-1)$ moved to phase three.

Once entered, the following steps are executed in this phase. At first, we merge batch-crossing situations from the previous batch. Algorithm 5.5 shows the details of this step. We loop over all situation streams and consider three different cases.

- i) If the tail of the previous batch does not contain a partial situation, nothing needs to be done (lines 3-4).
- ii) If there was a situation ongoing in the previous batch, but we have no information in the head of the current batch, this situation ends with the first event of the current batch. Thus, we set the end timestamp accordingly and add the merged situation to a new update (lines 5-7). At the end of merging, we add this new update to the batch (lines 16,17).
- iii) If we have information in the previous tail and the current head, this situation ends either within the current batch or in a future batch. To handle this case, we loop the situation updates of this batch. If we find a matching situation, we merge it by merging the aggregates of both situations and calculating

Algorithm 5.5: MergeSituations

Input: *previous*: SituationBatch
Input: *current*: SituationBatch
Input: *n*: Number of situation streams

```

1  newUpdate  $\leftarrow \emptyset$ ;
2  foreach  $i \in 1 \dots n$  do
3      if previous.tail[i] =  $\emptyset$  then
4           $\lfloor$  continue;
5      else if current.head[i] =  $\emptyset$  then
6          previous.tail[i].end = current.head.timestamp;
7          newUpdate  $\leftarrow$  newUpdate  $\cup$  {previous.tail[i]};
8      else
9          merged  $\leftarrow$  false;
10         foreach update  $\in$  current.updates do
11             if update[i]  $\neq \emptyset$  then
12                  $\lfloor$  merge(update[i], previous.tail[i]);
13                  $\lfloor$  merged  $\leftarrow$  true;
14         if  $\neg$ merged then
15              $\lfloor$  merge(current.tail[i], previous.tail[i]);
16 if newUpdate  $\neq \emptyset$  then
17      $\lfloor$  current.addUpdate(newUpdate);
    
```

the time interval of the merged situation (Lines 9-13). In case we find no appropriate update, the situation covers the whole batch and ends within a future batch. Hence, we merge the information from the previous tail into the current tail (Lines 14,15).

After completing the merge, we add the situations to the corresponding buffers in proper order.

Finally, we purge outdated events from the buffers. Therefore, we compute the minimum timestamp of relevant situations (i.e., situations that may be part of a successful match of any active thread) by subtracting the defined window size from the smallest timestamp of the current situation batch. Every processing thread maintains an instance of this timestamp, and we use the global minimum among all threads as a safe lower bound for removing outdated situations.

Matching

The matching process works similar to the single-threaded version, but instead of processing a single situation update at a time, we process all batch updates in one go. However, there are two challenges here: First, threads must be able to execute the matching phase while another thread is in the synchronization phase, resulting in concurrent buffer reads and writes. Second, in contrast to the single-threaded version, the clean-up process for the buffer has to consider the state of all working threads.

We solve the first challenge with a specialized buffer implementation. As stated in Section 5.5.2, the situation buffers are array-backed ring buffers. These buffers store data in arrays of fixed size and manage two indexes. One index points to the next write slot (idx_w) while the other one points to the first element of the buffer (idx_r). Instead of relying on the physical addresses in an array that may shrink/grow/loop around, we introduce logical addresses such that add operations increase idx_w and clean-up operations also increase idx_r . Consequently, the following invariants hold at any time:

1. $idx_r \leq idx_w$
2. $idx_w - idx_r \leq size$

It is easy to see that (i) all valid data lies between idx_r and idx_w , (ii) references remain valid even if the buffer is modified (as long as they still fall into this range), and (iii) a simple modulo computation obtains the physical array indexes. Processing threads working on a situation batch can now receive a logical address range, thus resolving problems with concurrent buffer reads and writes. In Figure 5.7 we illustrate this with an example. The physical array has 10 addresses, while the logical addresses continue to grow to 28. Every thread has a range of logical addresses it currently processes.

We solve the second challenge with a slightly modified variant of the range-query on the buffers (cf. Section 5.5.2). During the synchronization phase, the processing thread retrieves two index values for every buffer. The first one is the upper bound of relevant situations. This bound initially equals the value of idx_w before adding any situation of the currently processed batch. Then, when processing the situations of the current batch, the upper bound is increased accordingly. The second is the lower bound of relevant situations, which corresponds to the timestamp computed for purging outdated situations.

We use those indexes to restrict the range-queries appropriately and prevent other threads from cleaning required data from the buffers. The current processing

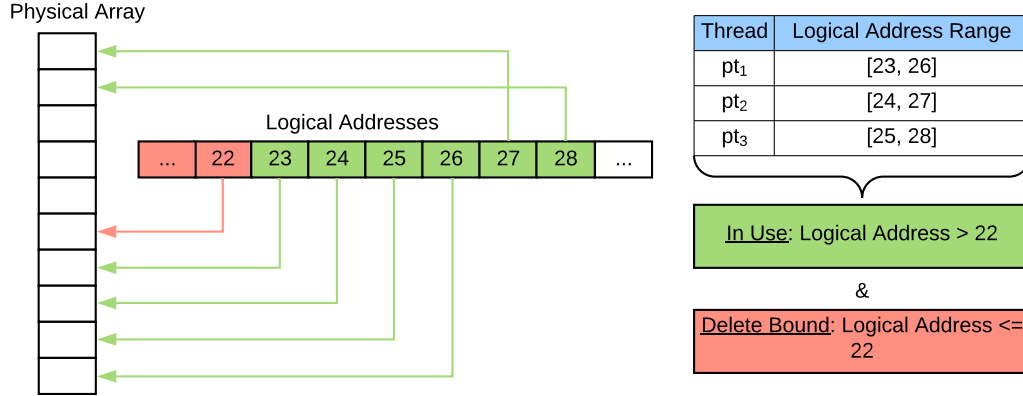


Figure 5.7: Example of the buffer implementation with logical addressing.

thread can purge all situations older than the globally lowest bound. For example, in Figure 5.7, the lowest address in use by threads pt_1 , pt_2 , and pt_3 is 23. Thus, it is safe to purge everything below or equal to 22.

5.6.4 Auto-Tuning

In a streaming scenario with fluctuating data rates, fixed batch sizes and a fixed number of processing threads is not an option. Large values allow for high throughput and the graceful handling of peak loads. However, the latency suffers because event batches take a long time to fill. Furthermore, we waste resources because the processing threads are possibly starving. On the other hand, small batch sizes and few processing threads generate low latency results for moderate event rates but lead to congestion when the load increases. In the following, we develop an auto-tune component, which continuously monitors the workload and tunes the batch size and the number of threads to handle the current load with the lowest latency and lowest resource consumption possible. We first describe the approach for partitioned data before we discuss the necessary adjustments to the approach for the unpartitioned case.

Independent of the parallelization approach, the auto-tune component uses a small set of parameters and sensors, which we briefly describe upfront (Table 5.4 gives a complete overview). Note that all sensors exclude times for thread synchronization, as they are unpredictable, and our model relies on predicting processing times. Instead, the model handles synchronization times implicitly.

Name	Description
Parameters	
r_{sched}	The scheduling rate of the auto-tune component
r_{batch}	The target batch rate in batches/second.
pt_{max}	The maximum number of processing threads to use.
β	Factor for over-provisioning the number of threads.
Sensors	
r_{in}	The input event rate.
bs	The current batch size.
t_{proc}	The processing time of a single event batch (of size bs) without the time required to deque event batches.
t_{wait}	The time spent on waiting to enter the synchronization phase. (unpartitioned approach only).
t_{sync}	The time spent in the synchronization-phase, without the wait time to enter it (unpartitioned approach only).
t_{buf}	The time spent updating the situation buffers during the synchronization phase (unpartitioned approach only). We have: $t_{buf} \leq t_{sync}$
Results	
bs^*	The target batch-size, i.e. the new batch-size after auto-tuning.
t_{proc}^*	The predicted processing time based on bs^* .
t_{sync}^*	The predicted synchronization time based on bs^* .
pt^*	The target number of processing threads, i.e. the number of threads after auto-tuning.

Table 5.4: Overview of parameters and sensors for the auto-tune component.

Auto tuning is scheduled at a fixed rate (r_{sched}). The sensors maintain average values of their respective measures and are reset after every auto-tune execution. r_{in} is the input event rate, measured in events per second, bs the currently used batch size, and t_{proc} is the time required to process a batch of bs events. pt_{max} is the maximum number of processing threads, and r_{batch} describes the target batch rate. With r_{batch} , we control the number of batches that should be created per second, effectively limiting the synchronization overhead introduced by en- and dequeuing event batches into/out of the queues. With the input rate and the target batch rate, the system is able to compute the target batch size bs^* as follows.

$$bs^* = \arg \min_{b \in \mathcal{B}} \left(\left| r_{batch} - \frac{r_{in}}{b} \right| \right) \quad (5.7)$$

That is, we choose bs^* from a set of batch sizes (\mathcal{B}), such that the resulting batch rate is as close as possible to r_{batch} . We define this set as $\mathcal{B} = \{32, 64, \dots, 32768\}$.

Powers of two have the following advantages: i) The number of possible values is small, ii) tiny to huge batches are possible, and iii) it allows fine-grained adjustments for smaller batch sizes.

Then, we predict the processing time per batch when using bs^* as the batch size and, based on this computation, the optimal number of threads to use. The predicted processing time (t_{proc}^*) increases linearly with the batch size for the following reason. First, deriving situations causes constant costs per event. Second, assuming that the number of derived situations increases linearly with the number of events, the number of matcher invocations also increases linearly. Hence, t_{proc}^* can be predicted as follows.

$$t_{proc}^* = t_{proc} \cdot \frac{bs^*}{bs} \quad (5.8)$$

We then use t_{proc}^* to compute the number of threads required.

$$pt^* = \min \left(\left\lceil \frac{r_{in}}{bs^*} \cdot t_{proc}^* \cdot \beta \right\rceil, pt_{max} \right) \quad (5.9)$$

Essentially this is the estimated number of batches per second times the estimated processing time per batch, rounded up. We additionally use the parameter β to over-provision the number of threads and leave some space for queue operations.

Handling of Skewed Input

Typically, the number of events to process varies among partitions. Assuming that the number of partitions is much larger than the number of processing threads, the proposed round-robin strategy achieves a good load-balancing between threads in most cases. However, in case of heavily skewed data, this might not be sufficient to balance the load between threads. Consider for example two processing threads (pt_1, pt_2) and four partitions with varying event rates ($part_1, \dots, part_4$). $part_1$ and $part_3$ receive 10^6 events/s while $part_2$ and $part_4$ receive only 1000 events/s. According to the round-robin strategy, $part_1$ and $part_3$ are assigned to pt_1 , while $part_2$ and $part_4$ are assigned to pt_2 . Obviously, pt_2 will be idle most of the time, while pt_1 is overloaded.

In order to solve this problem, the auto-tune component continuously monitors the current load per thread and migrates partitions from overloaded threads to idle threads. In the following, we describe how we monitor the load and when we trigger a migration. As stated in Section 5.6.2 the producer thread creates batches of events and assigns them to processing threads according to the current partitioning scheme. During this process, we collect additional statistics. Namely,

the total number of batches created (n) and for every active thread (pt_i) the batches assigned to it (n_{pt_i}). Let m be the number of active threads. Then, an optimally balanced system would yield $n_{pt_i} = \frac{n}{m}, i = 1, \dots, m$. However, in real world scenarios, such an optimal distribution can barely be achieved. Nevertheless, a distribution within a 10% range around the optimum is achievable leading to the following condition for partition migration:

$$\exists i \in [1, m] : \frac{n}{10 \cdot m} \leq \left| n_{pt_i} - \frac{n}{m} \right| \quad (5.10)$$

Note that we consider migration only if the auto-tune component did not decide to change the number of threads since this already changes the partition assignment.

Adjustments of Unpartitioned Data

For unpartitioned data, t_{proc} contains the time spent waiting to enter the synchronization phase (t_{wait}), which we need to ignore to obtain a reliable prediction of t_{proc}^* . Furthermore, the computation time of the synchronization phase does not scale linearly with the number of processed events for the following reason. While updating the situation buffers (t_{buf}) is linear in the number of events, the time for merging partial situations and cleaning the buffers is constant. The reason is that the complexity of merging depends only on the number of defined situation streams and the window size. Thus it is independent of the batch size. That said, we can compute t_{proc}^* for unpartitioned data as follows:

$$t_{proc}^* = \left(t_{buf} + \underbrace{t_{proc} - (t_{wait} + t_{sync})}_{\text{Time phases 1 \& 2}} \right) \cdot \frac{bs^*}{bs} + \underbrace{(t_{sync} - t_{buf})}_{\text{Sync. phase constant part}} \quad (5.11)$$

Furthermore, we need to assure that the processing threads are not blocking each other when entering the synchronization phase. To achieve this, we compute the expected processing time for the synchronization phase t_{sync}^* :

$$t_{sync}^* = (t_{sync} - t_{buf}) + t_{buf} \cdot \frac{r_{in}}{bs^*} \quad (5.12)$$

Since every thread needs to execute the synchronization phase once per event batch, t_{proc}^* should be longer than $(pt^* - 1) \cdot t_{sync}^*$, so that all remaining threads could theoretically execute the synchronization phase while the current thread is processing phases 1 and 3. We experimentally determined that if this ratio is

greater than 2, we encounter almost no wait times. Consequently, we scale up the batch size until the following inequation holds:

$$\frac{t_{proc}^*}{t_{sync}^*} \geq 2 \cdot (pt^* - 1) \quad (5.13)$$

Since upscaling the batch size does not change the required number of threads to manage the faced load, the auto-tune component can apply the computed values without touching the previously computed required number of threads (pt^*).

5.6.5 Distributed TPStream

The techniques for partitioned and unpartitioned data are also applicable in a distributed computing environment. We base our discussion of these adjustments on the architecture of Apache Kafka² because most streaming applications usually receive their data through some message queue and perform computation in a shared-nothing cluster. By integrating our work into a full-fledged framework like Kafka, we can use its fault tolerance and load distribution features and adjust them towards optimizing *TPStream*.

Apache Kafka's core is a distributed log allowing producers to write into and consumers to read from the log. Messages are sent to a so-called topic, and topics are further divided into partitions. Kafka stores every topic partition in a separate commit log on a Kafka broker, which is essentially one process in a cluster designed for managing the input and output of a partition. In general, Kafka stores an offset per consumer, i.e., consumers work and commit their offset to the log independently. To allow multiple consumers to work on a single task in parallel, they must belong to the same consumer group. The unit for parallel computations in Kafka is the number of partitions of a topic. Therefore, every partition in a topic is processed by exactly one consumer until a failure or load balancing reschedule the partition to another consumer of the same group.

Redistribution of data between consumers, such as exchanging situations between deriving and matching, is performed through additional topics, incurring overhead through writing to the topic logs and sending data over the network. Therefore, and due to not being able to rely on the highly optimized ring buffer implementation of *TPStream* in all processing stages, some adjustments are necessary when applying the techniques described above in a Kafka cluster.

²<https://kafka.apache.org/>

The partitioned data approach naturally translates into a Kafka architecture because incoming events can be partitioned based on the partition-by clause into m partitions. In a cluster with n consumer nodes, the input stream of *TPStream* is configured as a topic with $l = \min(m, n)$ partitions. If there are more logical partitions than physical processing nodes, we choose the lower number since every partition is a separate log file, thus incurring additional file system overhead. A group of l Kafka Streams³ consumer applications executes the *TPStream* operator utilizing exactly-once processing semantics. Results of each partition are written into a single, separate result topic monitored by a single consumer implementing the K -slack operation.

For unpartitioned data, we adjust the batching and merging phase. First, we divide the input topic into n topic partitions and distribute event batches to those partitions, effectively writing in a broker's log. This replaces the manual batching and queue logic of our single-machine approach. Situation batches are derived in parallel by a Kafka Streams consumer group that writes the resulting batches into a single result topic of unsorted runs. Unlike the single machine approach, matching is done by one Kafka consumer rather than the same deriving thread because synchronization of sequence numbers and merging those batches introduces significant overhead in a network environment. Due to this, we can perform a K -slack operation on the sequence numbers before the matching process to further improve processing speed.

So far, our single machine solutions have been primarily data-parallel. However, due to the modifications for a cluster environment, the strategies can be plugged into a wide variety of existing research. The second approach, in particular, uses typical pipeline parallel patterns since the matcher is an independent process waiting for the input of the deriver. This pattern opens the gate to adjust the work for GPU architectures similar to traditional pattern matching [CM12]. In addition, persisting intermediate results into Kafka topics allows for task-parallel solutions of sharing sub-patterns [RLR16] since multiple matchers with shared situation definitions can work with the same Kafka topic.

5.7 Experimental Evaluation

In this section, we present the results from our experimental evaluation of *TPStream*.

³<https://kafka.apache.org/documentation/streams/>

5.7.1 Setup

We run all single machine experiments on a workstation equipped with an AMD Ryzen7 2700X CPU (8 cores, 16 threads) and 16GB of memory, running an Ubuntu Linux (18.04, kernel version 4.16). The results for each experiment are averaged values from 10 runs, whereby every run is preceded by a warm-up phase of evaluating at least 100,000 events.

The main goal of this section is to compare *TPStream*'s processing performance and our low latency approach to the state-of-the-art solution for temporal pattern matching (*ISEQ*). Unfortunately, there is no publicly available implementation of *ISEQ*, so we implement it based on the description in [Li+11]. As required by the design of *ISEQ*, the input consists of interval streams ordered by the endpoint. We generate those streams with our *deriver* component.

In order to provide a comparison with point-based systems, we also include CEP-solutions from the open-source community (Esper 6.0.1 [esp20]) and academia (SASE+ ⁴) where applicable. While Esper is a production-ready online ESP system, highly optimized for efficient query execution, SASE+ is one of the most popular CEP languages in the research community and serves as the foundation for the *ISEQ* operator. The rich query language of Esper allows us to express both straw man's approaches as sketched in the introduction. We refer to the first approach (2 phase pattern matching) with *Esper-1* and the low latency approach as *Esper-2*. Because the SASE+ implementation does not feature chaining of queries, we only implement the low-latency approach. We implement *TPStream* and all its competitors in Java, whereby we integrate *TPStream* and *ISEQ* into Java Event Processing Connectivity (JEPC) – an event processing middleware developed at the University of Marburg [Hoß+13]. We use Oracle's JVM 1.8.0_181 with 16GB of heap space to run all experiments.

During the evaluation, we use two data sources. The first source comprises trip data generated with the Linear Road Benchmark [Ara+04]. Besides other attributes, every event consists of a unique car id, its location, current speed, and acceleration. We generate data simulating 5 hours of traffic on a single expressway with 1,000 active cars per hour. Every active car reports its state once per second, leading to 887 million events (36 GB of data). The second source is a random event generator tuned to pose a high load on the system. It generates event streams with a configurable number of boolean attributes, each representing a single situation stream. The generated situations are valid for 10 to 100 seconds, while the gaps between two consecutive situations span 10 to 50 seconds (both

⁴<https://github.com/haopeng/sase>

uniformly distributed). The generator produces events with a frequency of 1Hz so that for a situation lasting n seconds, the corresponding attribute value is `true` for exactly n consecutive events.

Independent of the data source and except for the parallel section, we use a single thread for both reading/generating the data and evaluating the query. In every experiment, we measure the reading/generation time upfront and remove it from the presented results. The most important parameters throughout all experiments are as follows.

Event Rate: The rate (events/s) with which events enter the system.

Window Size: The size of the time window (seconds) during which a pattern must occur completely.

Event Count: The total number of events to process.

5.7.2 Processing Time

This set of experiments compares the processing performance of *TPStream* with its competitors using various queries and parameter settings. We ingest the events at the maximum possible rate and use the processing time as the primary measure.

Aggressive Drivers

We ingest different fractions (1M to 100M events) of the Linear Road dataset into the system and execute the example query of Listing 5.1 (without duration constraints). The thresholds for speeding, acceleration and deceleration are the 99th, 90th and 90th percentiles for the speed and positive/negative acceleration values of a 50M event sample, respectively. Besides chaining of queries, the SASE+ implementation also lacks support for disjunctions. Nevertheless, to include SASE+ in this experiment, we also evaluate a simplified query version which restricts the used TRs to *meets* and *overlaps*.

Figure 5.8 shows the results of this experiment. The x-axis shows the number of processed events, while the y-axis depicts the processing time. *TPStream* and *ISEQ* are head to head, and their processing times increase linearly with the number of processed events. Furthermore, they are insensitive to alternatives, resulting in almost identical processing times for both query variants. *TPStream* can not outperform *ISEQ* in this experiment because all situations overlap in the given

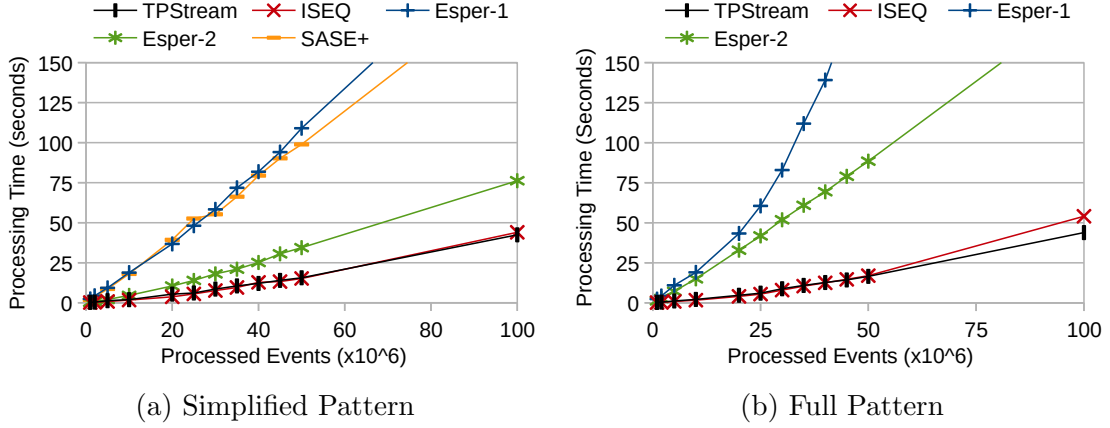


Figure 5.8: Processing time for aggressive driver detection as a function of the input size.

pattern, which allows breaking the buffer scan early. However, *TPStream* reduces the detection latency (time between the start of the first situation and the detection time) up to 70% (13% on average) compared to *ISEQ*.

Esper benefits from the simplified pattern, but its evaluation performance is inferior to *TPStream* and *ISEQ*. Esper-1 performs worse than Esper-2 in both cases. *TPStream* and *ISEQ* are about 8 (simplified pattern) and 20 (full pattern) times faster. However, Esper-2 performs well on the simplified pattern and requires only twice the processing time of *TPStream* and *ISEQ*. When looking at the full pattern, its performance drops (factor 4.3) since the automaton maintains a lot more states. Finally, the performance of SASE+ is equal to Esper-2.

Disconnected Pattern

The second experiment compares processing time and memory consumption of the systems using a more complex pattern: *A starts B before C overlaps D*. The difference to the first experiment is that every *A starts B* sub-match may be related to many *B overlaps C* sub-matches instead of contributing to at most one match. Hence, we expect the processing time/memory consumption to depend on the size of the configured time window. We inject 100M synthetic events into the systems and execute the query with window sizes varying from 500s (8:20 minutes) to 100,000s (slightly more than one day).

Figure 5.9 shows the processing time of all systems as a function of the window size. In this experiment, *TPStream* outperforms *ISEQ* by more than a factor

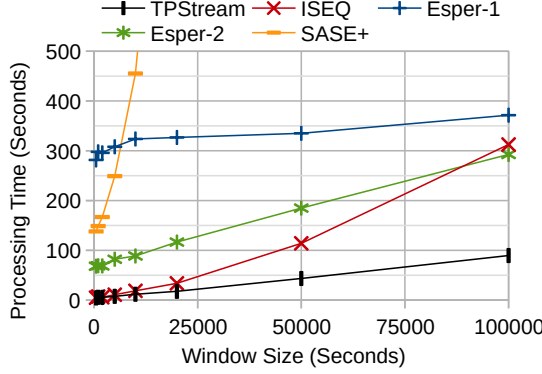


Figure 5.9: Processing time for disconnected pattern detection as a function of the window size

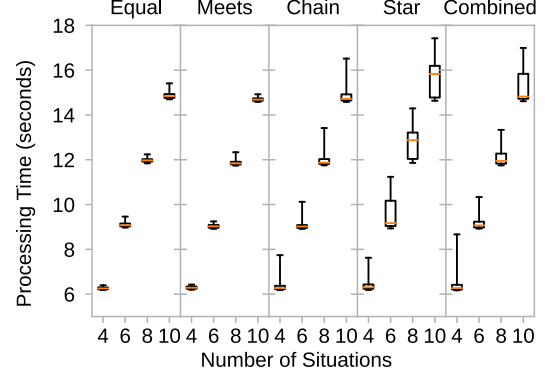


Figure 5.10: Processing time for various query patterns

of 3 using a window of 100,000s. The reason is that *ISEQ* does not exploit the order on the situations' start timestamp and requires additional computational steps during result construction and buffer pruning. *SASE+* is not able to manage the many intermediate automaton states efficiently and required approx. 5:20h to finish this experiment with the largest window size. *Esper* behaves similarly to the previous experiment, except that *Esper-2* outperforms *ISEQ* for 100,000-second windows.

To measure the average memory consumption, we monitor the used heap space with a frequency of 20Hz during each run and average these values. Here, *ISEQ* requires the least memory by far (452 - 465 MB), independent of the window size. *TPStream*'s memory consumption behaves similarly for window sizes up to 10,000 (466-507 MB) but increases for larger windows (up to 3,6GB). The reason is that we create many small objects during the buffer search, which are not immediately garbage collected. *Esper* uses around 4GB, independent of the window size, and *SASE+* consumes 3 - 11 GB of memory.

Query Patterns

To give a comprehensive overview of *TPStream*'s processing performance, we evaluate five different query patterns and vary the number of situation streams from 4 to 10. Query-Patterns 1-3 (**Equal**, **Meets**, **Chain**) are of the form:

$$Q_n = S_1 \oplus_1 S_2 \wedge S_2 \oplus_2 S_3 \wedge \cdots \wedge S_{n-1} \oplus_{n-1} S_n$$

Here, n denotes the number of situation streams and \oplus_i the TR connecting S_i and S_{i+1} . \oplus_i is set to *equals*, *meets* and a randomly drawn TR for query patterns 1,2 and 3 respectively. For Query pattern 4 (**Star**), S_1 is connected with every other situation:

$$Q_n = S_1 \oplus_1 S_2 \wedge S_1 \oplus_2 S_3 \wedge \cdots \wedge S_1 \oplus_{n-1} S_n$$

Again, n denotes the number of situation streams and \oplus_i the TR connecting S_1 and S_i . Like for the **Chain** pattern \oplus_i is a randomly drawn TR. Query 5 (**Combined**) combines the **Chain** and **Star** patterns by connecting the first $n/2$ situations via the **Chain** pattern and the remaining situations according to the **Star** pattern. Every query-type is executed 100 times, using 50M synthetic events and a window size of 2,000s.

The box plots in Figure 5.10 provide the median as well as the 25th and 75th percentiles of the processing time. For all query types, the median processing time increases linearly with the number of situations. The generic **Chain** pattern incurs higher maximum values than **Equal** and **Meets**, because the possible TRs include *before*, which selects many situations. This forces the matcher to build many partial results – especially if three or more consecutive situations are in a *before* relationship. **Star** queries are more sensitive to the concrete pattern instance because in the worst case, every situation triggers the matching process. We also observe this effect for the **Combined** pattern – albeit to a smaller degree because it connects only half of the situations via a **Star** pattern.

5.7.3 Low Latency

This set of experiments compares the latency of our approach with the state-of-the-art solution for temporal pattern matching, *ISEQ*.

Application Time

At first, we measure the latency improvement of *TPStream* compared to *ISEQ* in terms of application time. Therefore, we evaluate each TR independently using two synthetic situation streams (A, B). We vary the average duration ratio from 2:1 to 1:2 while fixing A 's average duration to 55 seconds. In this experiment, we set the window size to 1,000s.

Figure 5.11 reports the relative detection time of *TPStream* compared to *ISEQ* for the tested TRs and duration ratios. We define the detection time as $t_d(P) - \min(A.ts, B.ts)$. That is, the total time from the beginning of the first situation

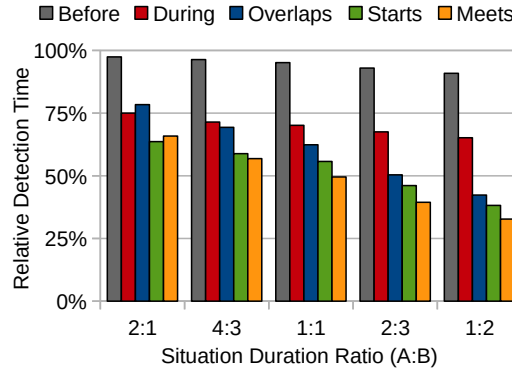


Figure 5.11: Relative detection latency per TR compared to *ISEQ*.

participating in the match until the detection time. The results show that the latency gain increases with increasing length of B -situations, independent of the TR. The reason is that the longer B becomes, the longer *ISEQ* has to wait for its end timestamp. Among all TRs, *before* shows the smallest and *meets* the largest latency improvements. However, in absolute numbers, we reduce the detection time by the length of the B situation for both TRs since the detection time is $B.ts$ in either case. The difference is that *meets* allows no gap between the situations, which restricts the detection time to the sum of both durations. In contrast, *before*-matches can span the entire window (1,000s in this case). For the remaining relations, the detection time is $A.te$, and the average improvement depends on the concrete TR. In the worst case (*during*) this is $\frac{B.te - B.ts}{2}$. Note that we exclude *equals* and *finishes* because no latency improvements can be achieved for those TRs.

Wall Clock Latency

We conduct two experiments showing that *TPStream*'s processing techniques significantly reduce the result latency in terms of wall clock time, a critical aspect in a streaming scenario. Therefore, we repeat the experiment from Section 5.7.2 twice. In the first experiment, we measure the time passed between the arrival of the first event that could produce a result and the receipt of that result. We vary the window size and push events with the maximum possible rate. For the second experiment, we fix the window size at 100,000s and vary the event rate from 1 Mhz to 1 Hz. This time, we split the measured latency into (i) processing latency and (ii) event latency. Processing latency is the time passed between the arrival of the event that triggered the result and the actual receipt of that result. In contrast, event latency is the time passed between the arrival of the first event

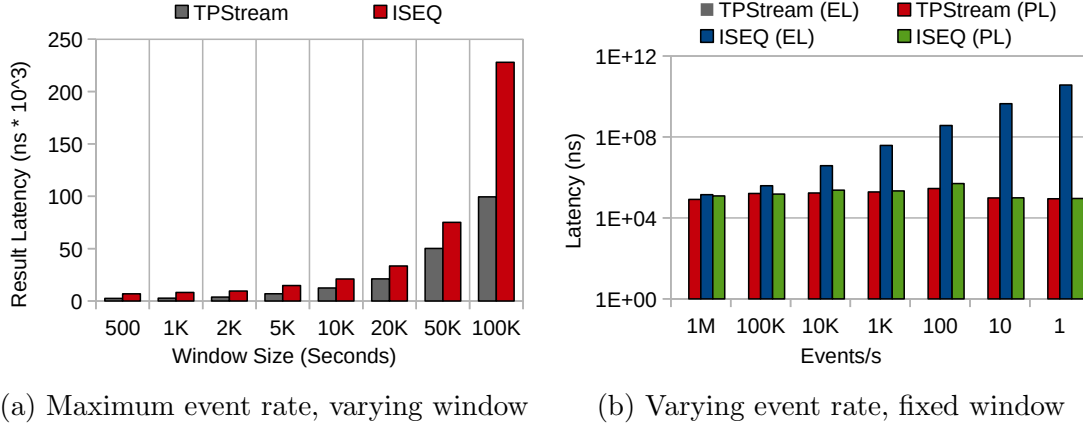


Figure 5.12: Comparison of result latency (a) under maximum possible throughput as a function of the window size, (b) under varying event rates with a fixed size window (EL = event latency, PL = processing latency).

that *could* trigger the result and the arrival of the event that *actually* triggered that result.

The results are shown in Figure 5.12. Both figures show the average latency per result (y-axis, note the log-scale for b). While Figure 5.12 (a) shows, that *TPStream*'s evaluation techniques provide latency savings through reduced processing time, Figure 5.12 (b) highlights the savings achieved with our low-latency matcher. Especially when the rate is in sync with application time (1 event/s), the event latency (EL) of *ISEQ* dominates the processing latency (PL) and almost reaches the application time savings (approx. 35s, cf. Figure 5.11, 1:1, *overlaps*), while *TPStream* introduces no event latency at all.

5.7.4 Plan Quality & Adaption

In this set of experiments, we evaluate the optimization techniques presented in Section 5.5.4. Analogous to Section 5.7.2, we ingest events with the maximum possible rate.

Initial Plan Quality

To evaluate the quality of the generated initial plans, we used the following queries on three situation streams:

Q1: *A overlaps B AND A overlaps C AND B starts C*

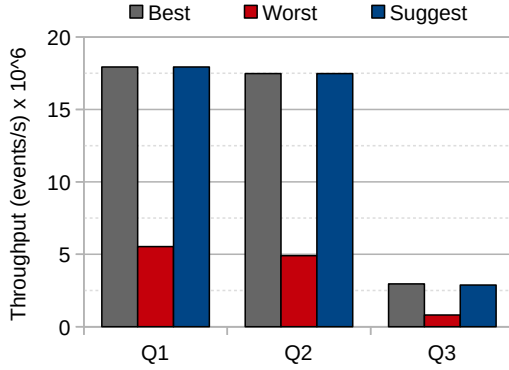


Figure 5.13: Quality of the initial plans for Q1 – Q3

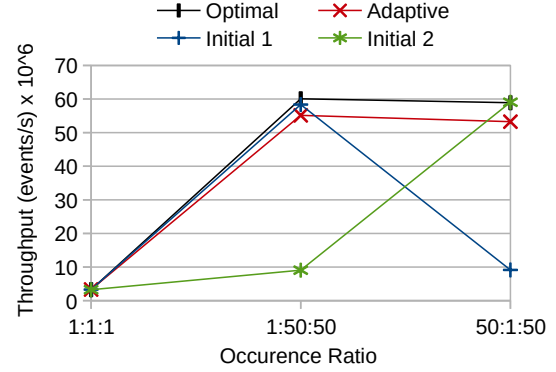


Figure 5.14: Throughput comparison: dynamic plan adaption vs. best initial plans

Q2: *A overlaps B AND A before C AND B overlaps C*

Q3: *A before B AND A before C AND B before C*

We generate all six valid plans for each query and measure the throughput (processed events/s) by evaluating synthetic events with a window size of 5,000s.

Figure 5.13 shows the results for the best, worst, and suggested plans and clearly confirms our approach. For queries **Q1** and **Q2**, the best plan is suggested. The initial plan for **Q3** is $C \rightarrow B \rightarrow A$ even though the estimated costs for $C \rightarrow A \rightarrow B$ are identical. The experiments show that $C \rightarrow A \rightarrow B$ is a slightly better choice, but the difference is negligible.

Dynamic Plan Adaption

To analyze the plan adaption capabilities of *TPStream*, we execute **Q3** again and process 300M events. The occurrence ratio of situations *A*, *B* and *C* change from 1:1:1 to 1:50:50 after 100M events and finally to 50:1:50 after 200M events. We set the window size, the EMA smoothing factor (α), and the threshold for plan migration to 10,000s, 0.01, and 0.2, respectively. Besides the adaptive implementation (Adaptive), we run the experiment with both best initial plans $C \rightarrow B \rightarrow A$ (Initial-1), $C \rightarrow A \rightarrow B$ (Initial-2), and an implementation, doing a hardcoded switch to the best plan exactly when the characteristics of the stream changes (Optimal).

Figure 5.14 shows the throughput for all four configurations and the three different stream characteristics. Initial-1 and Initial-2 have drawbacks in either one of the

skewed phases, while our adaptive approach is very close to the optimal solution (suffering slightly from dynamic adaption). However, the total runtime of Optimal (33,959ms) compared to Adaptive (34,106ms) reveals only a negligible overhead of 147ms (less than 1%) for plan adaption.

5.7.5 Parallel Approaches

Finally, we evaluate the parallel approaches presented in Section 5.6. We first analyze the speedup achieved for partitioned and unpartitioned data before showing our adaptive variants' efficiency. For every experiment, we use a dedicated producer thread. This thread reads/generates the input events and pushes them into *TPStream*. When processing the data, we vary the number of processing threads and measure the resulting speedup compared to single-threaded execution. We close this section with an evaluation of our distributed strategies using Apache Kafka.

Partition Parallelism

To evaluate our approach for partitioned data, we reuse the query for detecting aggressive drivers (`PARTITION BY car_id`). We read 50M events from the event file and push them into *TPStream* as fast as possible. We vary the number of processing threads from 1 to 16 and the batch size between 128, 1024, 8192 events. The size of the input queues is set according to Section 5.6.1, summing up to 2.5 MiB per thread (40 bytes per event). With a *K*-slack size of 20 events, we do not face out-of-order results in any of the tested configurations. Furthermore, we execute the experiment twice. Once with events regularly read from the data file, and once with preloading the events into memory since we expected the disk I/O to become a bottleneck. In both cases, a dedicated producer thread is responsible for ingesting events into *TPStream*. For the first case, this thread is also responsible for the disk I/O.

Figure 5.15 shows the results for this experiment. The x-axis shows the number of processing threads, while the y-axis depicts the speedup compared to single-threaded execution. With events read from the data file, *TPStream* scales nicely up to four threads, independent of the batch size (Figure 5.15 (a)). For more processing threads, the producer thread becomes the bottleneck of the pipeline since it has to handle both the disk I/O and the assignment of event batches to processing threads. Consequently, larger batch sizes perform slightly better because fewer batches need to be assigned (i.e., fewer queue operations are needed).

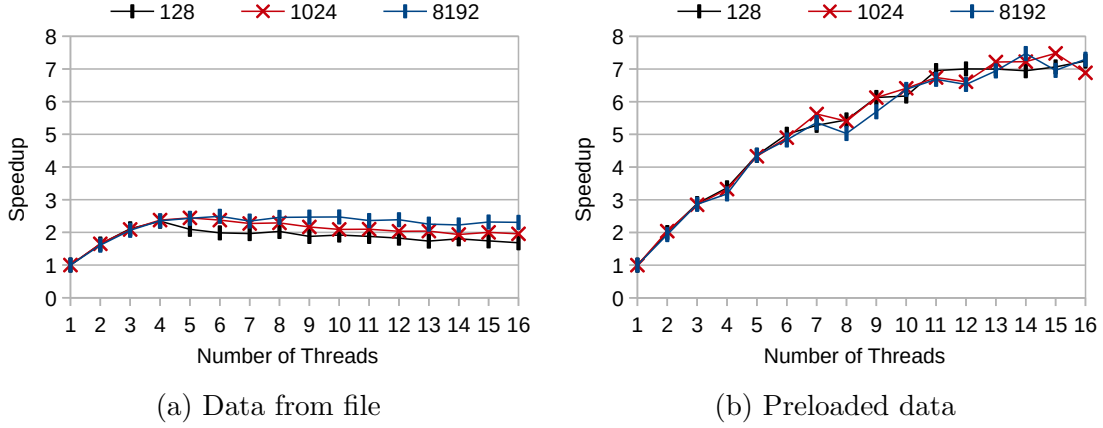


Figure 5.15: SpeedUp compared to single-threaded mode for partition parallel execution of the aggressive drivers query with different batch sizes and (a) data loaded from file, (b) preloaded data

With preloaded data (Figure 5.15 (b)), our approach scales nicely up to 7 processing threads because the producer is not blocked due to I/O operations and can provide sufficient data to the input queues of the workers. With 8 processing threads, we expect a slight dip because the CPU has 8 physical cores, and we use a dedicated producer thread. Afterward, hyper-threading comes into play. With more than 8 processing threads, the speedup achieved per thread reduces significantly. Note that the batch size has no impact on the processing performance since the producer can provide data sufficiently fast, and processing threads are independent. Due to this independence, the window size also does not affect the achieved speedup.

Unpartitioned Data

For this experiment, we reuse the query from Section 5.7.2 because its computational complexity increases with the configured window size. We ingest 100M synthetic events from our event generator using a separate producer thread. We vary the number of processing threads and the batch size from 1 to 16 and between 128, 1024, 8192, respectively. We execute the query using small (500), medium (5,000, 20,000), and large (100,000) windows and measure the speedup compared to single-threaded execution. The size of the input queue is again set according to Section 5.6.1 (2.5 MiB, 40 bytes per event). In this experiment, the number of results is much higher than for the aggressive drivers query. The reason is that once we find a combination that satisfies $A \text{ starts } B$, almost all subsequent batches carry at least one match of $C \text{ overlaps } D$. Hence, they produce results until $A \text{ starts } B$

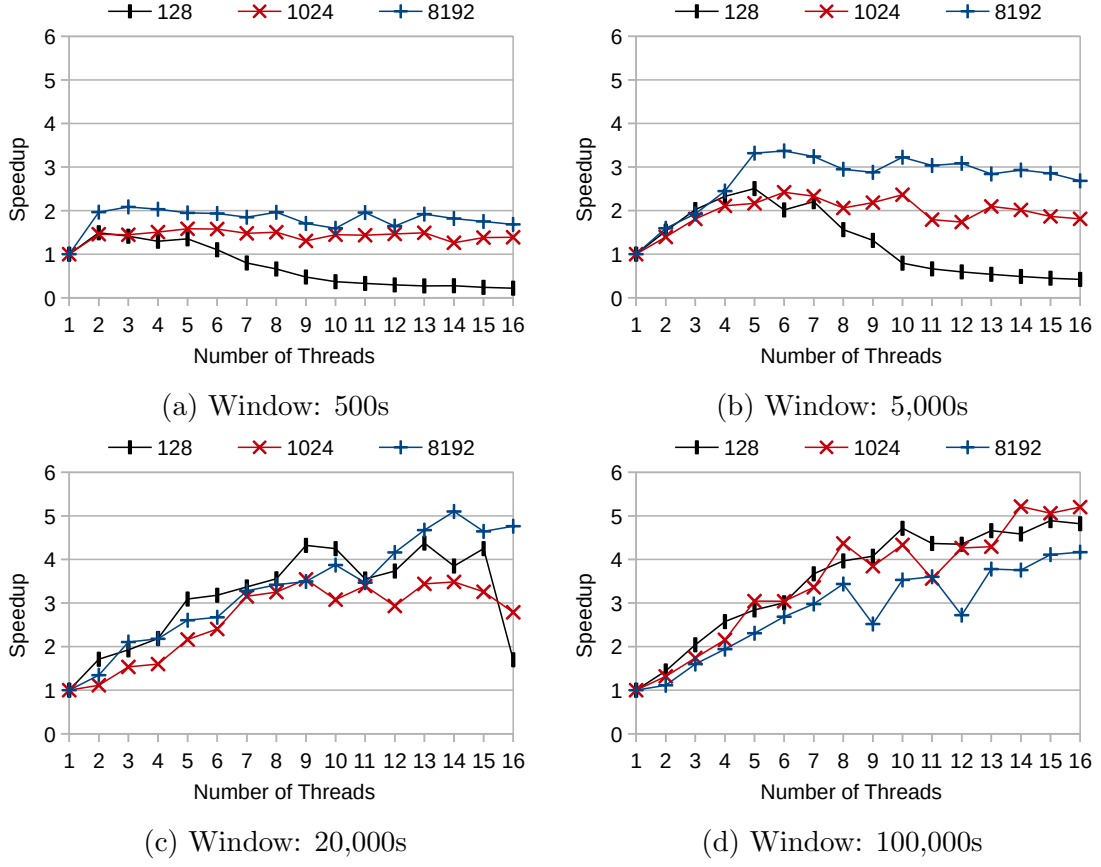


Figure 5.16: Processing time for parallel disconnected pattern detection for varying batch and window sizes.

leaves the window. Consequently, the size of the K -slack depends on the batch- and window-sizes as well as on the number of threads. For this experiment, we choose $K = 2 \cdot \#threads \cdot \text{avg}(res_{batch})$. That is, twice the number of active threads times the average number of results per batch ($\text{avg}(res_{batch})$), resulting in an out-of-order rate of less than 0.5%. For example, we set $K = 600$ for a window size of 100,000, a batch size of 1024, and 8 threads.

Figure 5.16 shows the results for this experiment. In case of the small batch size (128), Figure 5.16 (a) - (c) show a sudden drop at the respective tail of each experiment. This drop is a result of too small batches incurring congestion at the synchronization phase. For the less complex queries with window sizes 500 and 5,000, the producer can hardly saturate 4 and 6 threads, respectively. However, different from the partition-based parallel approach, the results show that the performance depends on the configured batch size. The lower the computational

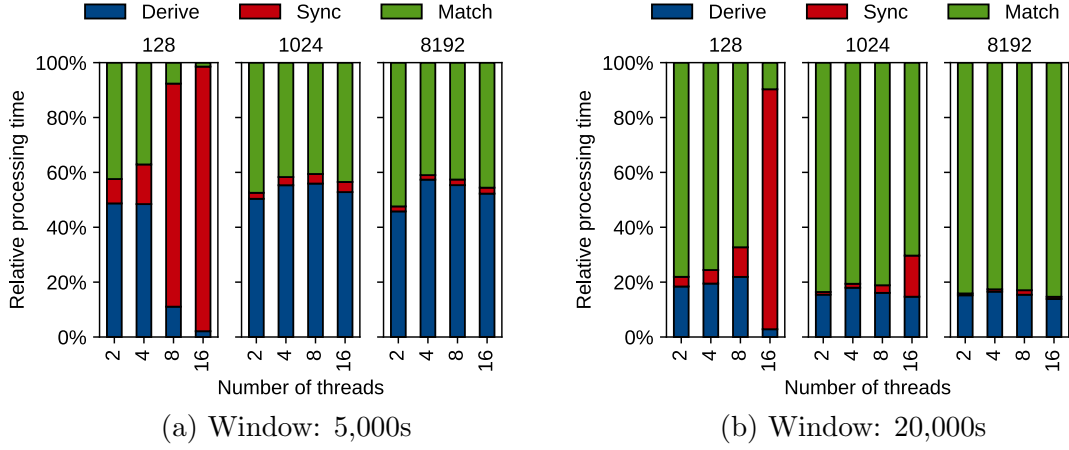


Figure 5.17: Relative processing times spent in the different phases of unpartitioned parallel processing for varying batch and window sizes.

complexity (i.e., the smaller the window size), the larger is the required batch size for scaling. The reason is that with less complexity, the time spent in the matching phase (i.e., the time of independent processing) reduces, and thus the probability of contention in the sync-phase increases. For the more complex queries, our approach can take advantage of all available resources and utilize all available threads.

Furthermore, Figure 5.17 shows the relative time spent in the different phases (y-axis) for varying batch-sizes (128, 1024, 8192), processing threads (x-axis), and windows of size 5,000 (a) and 20,000 (b). Following Figure 5.16, the results confirm that a sufficiently large batch size is essential for scaling. With the batch size being too small (128), the synchronization phase quickly becomes a bottleneck, especially for small windows. Additionally, this experiment showcases that the time spent in the matching phase grows proportional to the configured window size.

Auto-Tuning

To prove the validity of our auto-tuning approach, we use the 4 scenarios found in Table 5.5 and generate a changing workload with the following progression: 100M events at the maximum possible rate, 100M events at 10M events/s, 100M events at 5M events/s, 30M events at 1M events/s, 100M events at 5M events/s, 100M events at 10M events/s and 100M events at the maximum possible rate, summing up to 630M events to process. We additionally executed the queries

id	Query	Parallel Approach	Data	Window	Initial Threads	Initial Batch Size
a	Aggr. Drivers	Partitioned	Disk load	5 m	8	256
b	Aggr. Drivers	Partitioned	Preloaded	5 m	8	256
c	Disconnected	Unpartitioned	Generated	20,000 s	8	256
d	Disconnected	Unpartitioned	Generated	500 s	8	512

Table 5.5: Specification of the queries executed for the auto-tuning experiment

with batch size and number of processing threads fixed to the initial values given in Table 5.5.

We configured the auto-tune component with the following configuration: $r_{sched} = 1$, $r_{batch} = 100,000$, $\beta = 1.2$, $pt_{max} = 8$. Every time the auto-tune component changes a parameter, we track this change and create the timelines shown in Figure 5.18. The x-axis shows the elapsed processing time, highlighting the workload changes. The batch size is aligned on the left and the number of threads on the right y-axis.

As a first result, the processing times of the auto-tuned runs and the runs with fixed parameters were almost identical (varying by less than 1 second), showing that our approach introduces very low overhead. However, in all four cases, the timelines show that the initially configured parameters are way too high most of the time, thus wasting resources and introducing unnecessary latency. Furthermore, the adjustments always immediately follow the changes in the workload. We observe only a few back-and-forths in the parameter configuration, confirming the validity and robustness of our model.

For scenario (a), we see that the configured parameters stay stable until we throttle the event rate to 1M events/s. The reason is that when loading data from disk, the maximum reachable event rate is around 3M events/s. However, the parameters tend to exceed the previous values during the last period and quickly change between 32-64 and 2-4 for the batch size and number of threads, respectively. We attribute this to the operating system’s page cache, which seems to keep a fraction of the data file in memory. For preloaded data (scenario b), the workload changes are clearly reflected by the batch size and the used number of threads. However, compared to the other measurements, the changes in the number of threads are noisier. The reason is that the pattern is highly selective, and matcher invocations rarely occur, which in turn causes the average processing time per batch to vary.

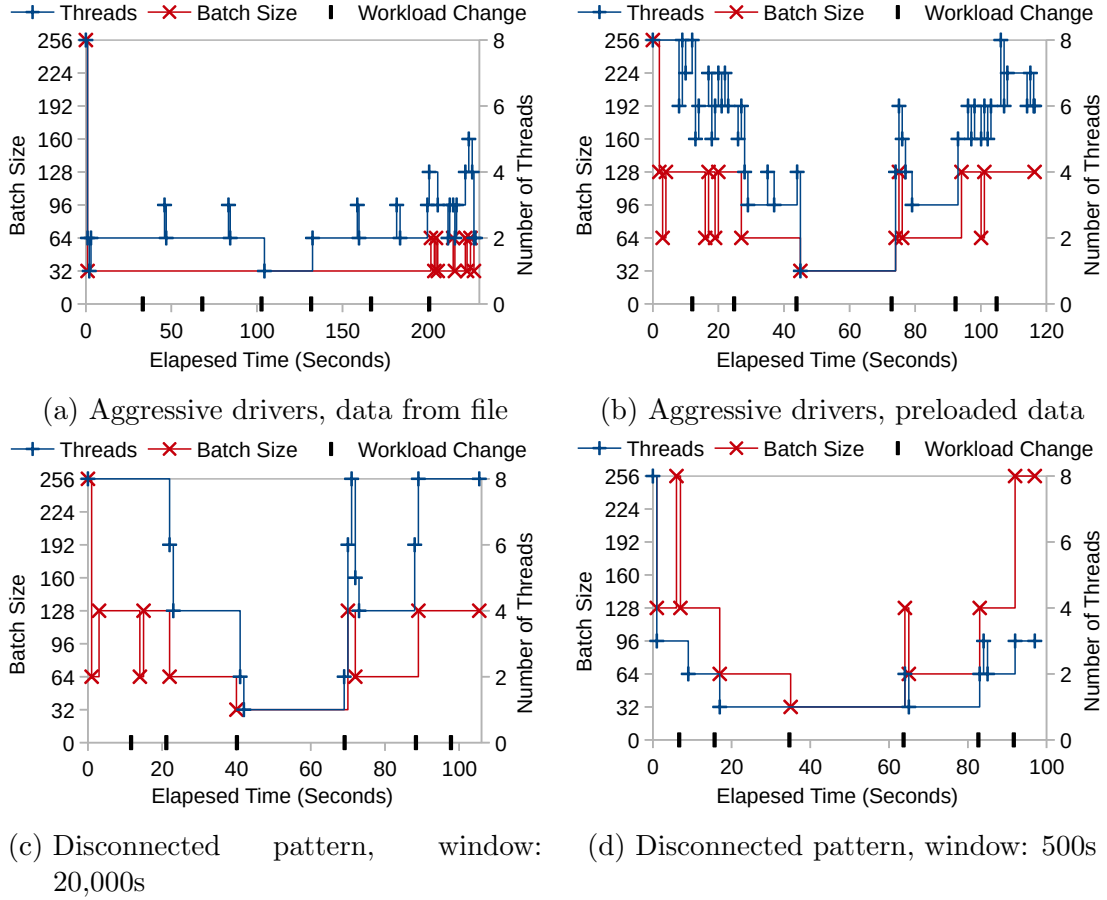


Figure 5.18: Evaluation of the configured batch size and number of threads over time for different queries.

In the unpartitioned cases (c,d), we do not encounter such noisy phases because the threads cooperatively process the entire input stream. Hence, if a computation takes longer, this is compensated by another thread. As expected, the utilized resources adapt to the changes in the workload. For example, the event rate halves slightly after 20 seconds, and so does the used number of threads. However, note that even though the complexity of the 500s window query is less than the one of the 20,000s window, the batch sizes tend to be greater. We attribute this to the congestion control feature since the processing time for the 500s window query is very low. Hence, with small batches, the processing threads would spend a reasonable amount of time waiting to enter the synchronization phase.

Component	Coordinator	Worker (10x)
CPU	Intel Xeon E52640v3 @2,6 GHz	AMD A10-7870K @ 3.9 GHz
RAM	16GB DDR4 FSB2133	32GB DDR4 FSB2133
Storage	2x480GB SSD; 8x8TB HDD	500GB SSD
Network	10 Gbit Ethernet	1 Gbit Ethernet

Table 5.6: Hardware specification of the cluster

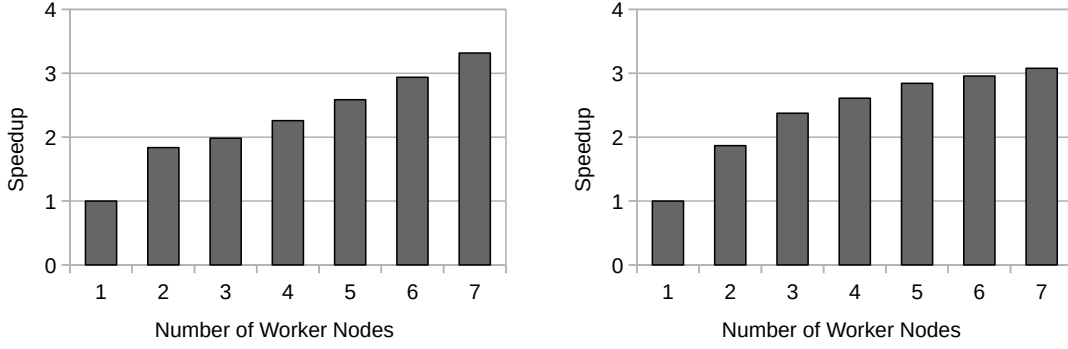
Distributed Environment

For our experiments in a distributed environment, we used a cluster with one coordinator and ten worker nodes, running an Ubuntu Linux (18.04, kernel version 4.15). The hardware specifications can be found in Table 5.6. On the cluster we deployed Apache Kafka 2.0.0 using Zookeeper⁵ 3.4.5 for coordination. Each worker node hosts one Kafka broker. For our experiments, we deploy single-core consumer applications on a variable number of worker nodes. Before each experiment, we insert 10 million input events through a Kafka topic as a warm-up during which Kafka balances partition distributions among the registered brokers. Afterward, we measure the speedup relative to a single worker node setup.

For partitioned data, we evaluate the aggressive drivers experiment with 50M events and the standard Kafka batch size of 10,000 events. We deploy the *K-Slack* consumer on a single worker node and leave the worker node that acts as a controller for all brokers with no workload. On the 7 remaining worker nodes, we create a varying degree of topic partitions and corresponding consumers. Since the controller handles consumer and partition assignments, a consumer not necessarily resides on the same node as the partition it is consuming. Figure 5.19 (a) shows the results of the experiment. Clearly, the approach scales with the used worker nodes since each worker has to read and process only a fraction of the total data.

For unpartitioned data, we again use the query from Section 5.7.2 and ingest 60M events from the generator. On display (Figure 5.19 (b)) are results for a batch size of 10,000 events and a moderate window size of 10,000. In the measurements, the processing time includes creating event batches from the raw data and the time until the last result is generated. Again, we leave the controller broker with no workload and use the 7 remaining worker nodes for a varying number of partitions. Similar to the partitioned data experiment, the approach scales since it distributes the reading and the deriving workload. The effects are less

⁵<https://zookeeper.apache.org/>



(a) Aggressive Drivers; 50M Events

(b) Disconnected pattern; 50M Events

Figure 5.19: Speed up in a cluster for various queries.

prominent than in the partitioned case since only the deriving happens in parallel.

Notably, both experiments take a longer processing time than their respective counterparts on a single machine. This is due to hardware limitations of each cluster node and additional overhead for transferring data over the network. However, since most streaming environments work with frameworks like Kafka to initially ingest data, achieve fault tolerance and provenance, the overall overhead is likely to occur in practice anyway. Thus, our adaptations show that *TPStream* integrates well with and benefits from an environment featuring Kafka.

5.8 Summary

We presented *TPStream*, a novel event processing operator for detecting complex TPs among event streams with low latency and high throughput. By coupling the deriving phase with the matching phase, *TPStream* detects complex TPs at the earliest possible point in time. To handle huge data volumes originating from a variety of sources, we developed parallel and distributed implementations for *TPStream* that can be applied to both partitioned and unpartitioned data streams. Furthermore, to maximize resource utilization in a distributed computing environment while reacting to changing data rates of streams, the parallel implementations are tied to a tuning component that automatically adjusts batch sizes and the number of computing threads. In experiments, we showcased *TPStream*'s performance benefits, scalability, and adaptive capabilities while comparing it to current state-of-the-art solutions.

Summary, Conclusion and Outlook

The efficient on- and offline processing of high-volume event streams is important in many application domains, such as technical infrastructure monitoring or financial market surveillance. Cluster-based scale-out general-purpose stream processing engines, like Apache Flink [Car+15], are the basis for many event processing application. These engines offer powerful high-level abstractions that allow users to implement custom processing logic easily. Moreover, they seamlessly integrate on- and offline processing by executing offline queries in a replay-based fashion. However, those engines offer two significant deficiencies. First, when executing offline queries in a replay-based fashion, I/O quickly becomes the bottleneck of query execution and dominates the overall query execution cost. Second, as a result of their high-level abstractions, those engines fail to fully utilize the available hardware resources of modern machines [Zeu+19]. In this thesis, we tackle these deficiencies in the following way.

First, we present approaches to speed up the offline processing of windowed aggregation and sequential pattern matching (cf. Chapter 3). We utilize well-understood indexing techniques to avoid a full stream replay and minimize I/O costs. Our approaches result in speedups of up to two orders of magnitude for both query types. In addition, we develop cost models for both approaches to decide (i) whether and (ii) how many indexes to use for a given query. We show that our cost models accurately predict the overall query execution cost independent of the query parameters and data distribution and thus ensure efficient query execution for a wide variety of workloads.

Second, we show how to increase hardware resource utilization of single machines by utilizing modern iGPUs in combination with low-level programming interfaces provided by HSA. For filter, windowed aggregation, windowed join, and sequential pattern matching, we provide iGPU-enabled implementations that outperform even multi-threaded CPU-based implementations by up to a factor of five. Moreover, we confirm the findings of Zeuch et al. [Zeu+19] that modern JVM-based

SPEs fail to utilize hardware resources of current machines fully. Even our single-threaded C++-based operator implementations increase processing throughput by two orders of magnitude compared to the state-of-the-art scale-out SPE Apache Flink.

Third, we present *TPStream*, a novel operator for matching patterns on temporal intervals based on Allen’s interval algebra [All83]. *TPStream* is the first interval-based pattern matching operator that tightly couples the derivation of situations and their associated time intervals from events with the matching of temporal patterns. This coupling results in very low processing latency. Moreover, we show that *TPStream* is easily parallelizable and takes advantage of the many cores of modern CPUs (i.e., efficiently utilizes the available hardware). Finally, we show that *TPStream* scales nicely in distributed environments by integrating it into Kafka Streams.

6.1 Outlook

After a summary of the most important results, this final section gives a brief outlook on future research opportunities for the approaches developed in this thesis.

Our index-based approaches for offline processing of CQs significantly increase the processing performance for a wide variety of query parameters. However, for sequential pattern matching, we introduce various restrictions that should be addressed in the future. First, we do not support logical grouping (i.e., a group by clause) so far. This clause allows to detect patterns in a single entity’s behavior (e.g., a sensor) within a stream of events from multiple entities (e.g., all events of a sensor network). However, with grouping, storing a global sequence number within secondary indexes is not sufficient for checking sequential distance constraints. The reason is that adjacent events within a logical group are likely not adjacent within the source stream. Moreover, the attribute values of different groups may follow different distributions, which need to be reflected in the cost model. In order to tackle this issue, one idea is to use an additional secondary index on the grouping attribute. This index would allow checking sequential distance constraints between the events of a group.

Another extension regarding offline sequential pattern matching is the support for predicates that refer to previous events like in the increasing temperature example in Section 2.3.5. Traditional indexes like B⁺-trees do not support those predicates. A first idea towards supporting those predicates is to design a specialized index

structure that stores deltas to previous values instead of absolute values. A hierarchy of such deltas would allow finding variable-length sequences of increasing or decreasing values.

In addition, the pruning power of Kleene-star symbols should be explored. Currently, we ignore KSs since they are not required for a successful match of the pattern. However, if a KS does not occur, the subsequent BS must occur instead. This fact could be used to reduce the number of replay intervals even for queries with only a few BSs.

Our iGPU and HSA-based operator implementations increase the processing throughput up to a factor of five compared to equivalent multi-threaded CPU implementations. However, when running multiple queries in parallel, each consisting of different operators, all processing units should be considered for operator execution to maximize the overall resource utilization. Thus, it seems to be advantageous to develop iGPU-aware scheduling strategies [Zha+21; Zha+20]. These strategies need to consider the overall query workload and the characteristics of both processing units to schedule operators on the best-suited processing unit.

Based on the efficient parallelization strategies of *TPStream* for multi-core machines, we also plan to explore the benefits of iGPU-based parallelism. A first idea is to replace the binary-search-based pattern matching step (cf. Section 5.5.2) with a brute-force approach on the iGPU. A brute-force search would also eliminate the requirement of keeping situations sorted in a buffer and thus allow low-cost integrated handling of out-of-order events.

Finally, we will examine index-based offline support for *TPStream*, analogous to our approach for sequential pattern matching. In a similar manner as for the symbol conditions, secondary indexes could support the conditions of situations. From the attached timestamps and sequence numbers, we can easily derive the corresponding time intervals. Thus, the remaining question is how to adjust the merge phase to recognize the 13 possible interval relations. Moreover, the cost model needs to be adjusted to estimate the probabilities of a particular interval relation.

Appendices

References

- [Aba+03] D. J. Abadi *et al.*, “Aurora: A new model and architecture for data stream management”, *VLDB J.*, vol. 12, no. 2, pp. 120–139, 2003. DOI: 10.1007/s00778-003-0095-z.
- [Aff+17] L. Affetti, R. Tommasini, A. Margara, G. Cugola, and E. D. Valle, “Defining the execution semantics of stream processing engines”, *J. Big Data*, vol. 4, p. 12, 2017. DOI: 10.1186/s40537-017-0072-9.
- [Aki+15] T. Akidau *et al.*, “The dataflow model: A practical approach to balancing correctness, latency, and cost in massive-scale, unbounded, out-of-order data processing”, *Proc. VLDB Endow.*, vol. 8, no. 12, pp. 1792–1803, 2015. DOI: 10.14778/2824032.2824076.
- [Ali+09] M. H. Ali *et al.*, “Microsoft CEP server and online behavioral targeting”, *Proc. VLDB Endow.*, vol. 2, no. 2, pp. 1558–1561, 2009. DOI: 10.14778/1687553.1687590.
- [All83] J. F. Allen, “Maintaining knowledge about temporal intervals”, *Commun. ACM*, vol. 26, no. 11, pp. 832–843, 1983. DOI: 10.1145/182.358434.
- [Ani+11] D. Anicic, P. Fodor, S. Rudolph, and N. Stojanovic, “EP-SPARQL: a unified language for event processing and stream reasoning”, in *Proceedings of the 20th International Conference on World Wide Web, WWW 2011, Hyderabad, India, March 28 - April 1, 2011*, S. Srinivasan, K. Ramamritham, A. Kumar, M. P. Ravindra, E. Bertino, and R. Kumar, Eds., ACM, 2011, pp. 635–644. DOI: 10.1145/1963405.1963495.
- [ABW06] A. Arasu, S. Babu, and J. Widom, “The CQL continuous query language: Semantic foundations and query execution”, *VLDB J.*, vol. 15, no. 2, pp. 121–142, 2006. DOI: 10.1007/s00778-004-0147-z.
- [Ara+03] A. Arasu *et al.*, “STREAM: the stanford stream data manager”, *IEEE Data Eng. Bull.*, vol. 26, no. 1, pp. 19–26, 2003.
- [Ara+04] A. Arasu *et al.*, “Linear road: A stream data management benchmark”, in *(e)Proceedings of the Thirtieth International Conference on Very Large Data Bases, VLDB 2004, Toronto, Canada, August 31 - September 3 2004*, M. A. Nascimento, M. T. Özsu, D. Kossmann, R. J. Miller, J. A. Blakeley, and K. B. Schiefer, Eds., Morgan

- Kaufmann, 2004, pp. 480–491. DOI: 10.1016/B978-012088469-8.50044-9.
- [AH00] R. Avnur and J. M. Hellerstein, “Eddies: Continuously adaptive query processing”, in *Proceedings of the 2000 ACM SIGMOD International Conference on Management of Data, May 16-18, 2000, Dallas, Texas, USA*, W. Chen, J. F. Naughton, and P. A. Bernstein, Eds., ACM, 2000, pp. 261–272. DOI: 10.1145/342009.335420.
- [AS19] S. Ayhan and H. Samet, “Data management and analytics system for online flight conformance monitoring and anomaly detection”, in *Proceedings of the 27th ACM SIGSPATIAL International Conference on Advances in Geographic Information Systems, SIGSPATIAL 2019, Chicago, IL, USA, November 5-8, 2019*, 2019, pp. 219–228. DOI: 10.1145/3347146.3359378.
- [Bab+04] S. Babu, R. Motwani, K. Munagala, I. Nishizawa, and J. Widom, “Adaptive ordering of pipelined stream filters”, in *Proceedings of the ACM SIGMOD International Conference on Management of Data, Paris, France, June 13-18, 2004*, G. Weikum, A. C. König, and S. DeBloch, Eds., ACM, 2004, pp. 407–418. DOI: 10.1145/1007568.1007615.
- [BSW04] S. Babu, U. Srivastava, and J. Widom, “Exploiting k -constraints to reduce memory overhead in continuous queries over data streams”, *ACM Trans. Database Syst.*, vol. 29, no. 3, pp. 545–580, 2004. DOI: 10.1145/1016028.1016032.
- [BG96] R. A. Baeza-Yates and G. H. Gonnet, “Fast text searching for regular expressions or automaton searching on tries”, *J. ACM*, vol. 43, no. 6, pp. 915–936, 1996. DOI: 10.1145/235809.235810.
- [Bal+13] C. Balkesen, N. Dindar, M. Wetter, and N. Tatbul, “RIP: run-based intra-query parallelism for scalable complex event processing”, in *The 7th ACM International Conference on Distributed Event-Based Systems, DEBS ’13, Arlington, TX, USA - June 29 - July 03, 2013*, S. Chakravarthy, S. D. Urban, P. R. Pietzuch, and E. A. Rundensteiner, Eds., ACM, 2013, pp. 3–14. DOI: 10.1145/2488222.2488257.
- [Bar+09] D. F. Barbieri, D. Braga, S. Ceri, E. D. Valle, and M. Grossniklaus, “C-SPARQL: SPARQL for continuous querying”, in *Proceedings of the 18th International Conference on World Wide Web, WWW 2009, Madrid, Spain, April 20-24, 2009*, J. Quemada, G. León, Y. S. Maarek, and W. Nejdl, Eds., ACM, 2009, pp. 1061–1062. DOI: 10.1145/1526709.1526856.

- [BM70] R. Bayer and E. M. McCreight, “Organization and maintenance of large ordered indexes”, in *Record of the 1970 ACM SIGFIDET Workshop on Data Description and Access, November 15-16, 1970, Rice University, Houston, Texas, USA (Second Edition with an Appendix)*, 1970, pp. 107–141.
- [Bei+19a] C. Beilschmidt *et al.*, “Pretty fly for a VAT GUI: visualizing event patterns for flight data”, in *Proceedings of the 13th ACM International Conference on Distributed and Event-based Systems, DEBS 2019, Darmstadt, Germany, June 24-28, 2019*, 2019, pp. 224–227. DOI: 10.1145/3328905.3332507.
- [Bei+19b] C. Beilschmidt *et al.*, “VAT to the future: Extrapolating visual complex event processing”, in *Proceedings of the 7th OpenSky Workshop 2019, Zurich, Switzerland, November 21-22, 2019*, 2019, pp. 25–36.
- [Ben+07] M. A. Bender, M. Farach-Colton, J. T. Fineman, Y. R. Fogel, B. C. Kuszmaul, and J. Nelson, “Cache-oblivious streaming b-trees”, in *SPAA 2007: Proceedings of the 19th Annual ACM Symposium on Parallelism in Algorithms and Architectures, San Diego, California, USA, June 9-11, 2007*, 2007, pp. 81–92. DOI: 10.1145/1248377.1248393.
- [Bha08] U. N. Bhat, *An Introduction to Queueing Theory*. Boston: Birkhäuser Boston, 2008, pp. 13–17. DOI: 10.1007/978-0-8176-4725-4.
- [BGJ06] M. H. Böhlen, J. Gamper, and C. S. Jensen, “Multi-dimensional aggregation for temporal data”, in *Advances in Database Technology - EDBT 2006, 10th International Conference on Extending Database Technology, Munich, Germany, March 26-31, 2006, Proceedings*, 2006, pp. 257–275. DOI: 10.1007/11687238_18.
- [Bol+06] G. Bolch, S. Greiner, H. de Meer, and K. Trivedi, *Queueing Networks and Markov Chains: Modeling and Performance Evaluation with Computer Science Applications*. Wiley, 2006, pp. 24–25.
- [BKS01] S. Börzsönyi, D. Kossmann, and K. Stocker, “The skyline operator”, in *Proceedings of the 17th International Conference on Data Engineering, April 2-6, 2001, Heidelberg, Germany*, 2001, pp. 421–430. DOI: 10.1109/ICDE.2001.914855.
- [Brä15] M. Bräger, “Large infrastructure monitoring at cern”, in *Presentation at Big Data Spain conference 2015, Madrid, Spain, 2015*, 2015.

- [Bre+07] L. Brenna *et al.*, “Cayuga: A high-performance event processing engine”, in *Proceedings of the ACM SIGMOD International Conference on Management of Data, Beijing, China, June 12-14, 2007*, 2007, pp. 1100–1102. DOI: 10.1145/1247480.1247620.
- [Cam15] M. Cammert, “Verbundoperationen über datenströmen und deren optimierung unter verwendung dynamischer metadaten”, Ph.D. dissertation, University of Marburg, 2015.
- [Car+15] P. Carbone, A. Katsifodimos, S. Ewen, V. Markl, S. Haridi, and K. Tzoumas, “Apache flink™: Stream and batch processing in a single engine”, *IEEE Data Eng. Bull.*, vol. 38, no. 4, pp. 28–38, 2015.
- [Car+16] P. Carbone, J. Traub, A. Katsifodimos, S. Haridi, and V. Markl, “Cutty: Aggregate sharing for user-defined windows”, in *Proceedings of the 25th ACM International Conference on Information and Knowledge Management, CIKM 2016, Indianapolis, IN, USA, October 24-28, 2016*, 2016, pp. 1201–1210. DOI: 10.1145/2983323.2983807.
- [Car75] A. F. Cardenas, “Analysis and performance of inverted data base structures”, *Commun. ACM*, vol. 18, no. 5, pp. 253–263, 1975. DOI: 10.1145/360762.360766.
- [CGR03] C. Y. Chan, M. N. Garofalakis, and R. Rastogi, “Re-tree: An efficient index structure for regular expressions”, *VLDB J.*, vol. 12, no. 2, pp. 102–119, 2003. DOI: 10.1007/s00778-003-0094-0.
- [CGM10] B. Chandramouli, J. Goldstein, and D. Maier, “High-performance dynamic pattern matching over disordered streams”, *Proc. VLDB Endow.*, vol. 3, no. 1, pp. 220–231, 2010. DOI: 10.14778/1920841.1920873.
- [Che+08] S. Che, J. Li, J. W. Sheaffer, K. Skadron, and J. Lach, “Accelerating compute-intensive applications with gpus and fpgas”, in *Proceedings of the IEEE Symposium on Application Specific Processors, SASP 2008, held in conjunction with the DAC 2008, June 8-9, 2008, Anaheim, California, USA*, 2008, pp. 101–107. DOI: 10.1109/SASP.2008.4570793.
- [CR02] J. Cho and S. Rajagopalan, “A fast regular expression indexing engine”, in *Proceedings of the 18th International Conference on Data Engineering, San Jose, CA, USA, February 26 - March 1, 2002*, 2002, pp. 419–430. DOI: 10.1109/ICDE.2002.994755.

- [CG08] G. Cormode and M. N. Garofalakis, “Approximate continuous querying over distributed streams”, *ACM Trans. Database Syst.*, vol. 33, no. 2, 9:1–9:39, 2008. DOI: 10.1145/1366102.1366106.
- [CM12] G. Cugola and A. Margara, “Low latency complex event processing on parallel hardware”, *J. Parallel Distributed Comput.*, vol. 72, no. 2, pp. 205–218, 2012. DOI: 10.1016/j.jpdc.2011.11.002.
- [DGR03] A. Das, J. Gehrke, and M. Riedewald, “Approximate join processing over data streams”, in *Proceedings of the 2003 ACM SIGMOD International Conference on Management of Data, San Diego, California, USA, June 9-12, 2003*, 2003, pp. 40–51. DOI: 10.1145/872757.872765.
- [DIG07] Y. Diao, N. Immerman, and D. Gyllstrom, “Sase+: An agile language for kleene closure over event streams”, Tech. Rep., 2007.
- [DBG14] A. Dignös, M. H. Böhlen, and J. Gamper, “Overlap interval partition join”, in *International Conference on Management of Data, SIGMOD 2014, Snowbird, UT, USA, June 22-27, 2014*, C. E. Dyreson, F. Li, and M. T. Özsu, Eds., ACM, 2014, pp. 1459–1470. DOI: 10.1145/2588555.2612175.
- [Din+11] N. Dindar, P. M. Fischer, M. Soner, and N. Tatbul, “Efficiently correlating complex events over live and archived data streams”, in *Proceedings of the Fifth ACM International Conference on Distributed Event-Based Systems, DEBS 2011, New York, NY, USA, July 11-15, 2011*, 2011, pp. 243–254. DOI: 10.1145/2002259.2002293.
- [Din+04] L. Ding, N. K. Mehta, E. A. Rundensteiner, and G. T. Heineman, “Joining punctuated streams”, in *Advances in Database Technology - EDBT 2004, 9th International Conference on Extending Database Technology, Heraklion, Crete, Greece, March 14-18, 2004, Proceedings*, 2004, pp. 587–604. DOI: 10.1007/978-3-540-24741-8_34.
- [DR04] L. Ding and E. A. Rundensteiner, “Evaluating window joins over punctuated streams”, in *Proceedings of the 2004 ACM CIKM International Conference on Information and Knowledge Management, Washington, DC, USA, November 8-13, 2004*, 2004, pp. 98–107. DOI: 10.1145/1031171.1031189.
- [Dut+19] A. Dutt, C. Wang, A. Nazi, S. Kandula, V. R. Narasayya, and S. Chaudhuri, “Selectivity estimation for range predicates using lightweight models”, *Proc. VLDB Endow.*, vol. 12, no. 9, pp. 1044–1057, 2019. DOI: 10.14778/3329772.3329780.

-
- [EWK90] R. Elmasri, G. T. J. Wu, and Y. Kim, “The time index: An access structure for temporal data”, in *16th International Conference on Very Large Data Bases, August 13-16, 1990, Brisbane, Queensland, Australia, Proceedings*, 1990, pp. 1–12.
 - [Ept+99] U. Epting *et al.*, “Cern lhc technical infrastructure monitoring (tim)”, in *Proceedings of the 7th International conference on accelerator and large experimental physics control systems, icapecs 1999, Sincrotrone Trieste, Italy, October 4 - 8, 1999*, 1999.
 - [Erw04] M. Erwig, “Toward spatio-temporal patterns”, in *Spatio-Temporal Databases: Flexible Querying and Reasoning*, Springer Berlin Heidelberg, 2004, pp. 29–53.
 - [esp20] espertech. “Esper complex event processing, streaming analytics, streaming sql”. (2020), [Online]. Available: <https://www.espertech.com/> (visited on 11/17/2020).
 - [Etz+16] O. Etzion, F. Fournier, I. Skarbovsky, and B. von Halle, “A model driven approach for event processing applications”, in *Proceedings of the 10th ACM International Conference on Distributed and Event-based Systems, DEBS '16, Irvine, CA, USA, June 20 - 24, 2016*, A. Gal, M. Weidlich, V. Kalogeraki, and N. Venkasubramanian, Eds., ACM, 2016, pp. 81–92. DOI: 10.1145/2933267.2933268.
 - [Fel71] W. Feller, *An introduction to probability theory and its applications*. Wiley, 1971.
 - [21] *Flinkcep - complex event processing for flink*, 2021.
 - [Gao+05] D. Gao, C. S. Jensen, R. T. Snodgrass, and M. D. Soo, “Join operations in temporal databases”, *VLDB J.*, vol. 14, no. 1, pp. 2–29, 2005. DOI: 10.1007/s00778-003-0111-3.
 - [Gao+15] S. Gao, T. Scharrenbach, J. Kietz, and A. Bernstein, “Running out of bindings? integrating facts and events in linked data stream processing”, in *Joint Proceedings of the 1st Joint International Workshop on Semantic Sensor Networks and Terra Cognita (SSN-TC 2015) and the 4th International Workshop on Ordering and Reasoning (OrdRing 2015) co-located with the 14th International Semantic Web Conference (ISWC 2015), Bethlehem, Pennsylvania, United States, October 11th - and - 12th, 2015*, K. Kyzirakos *et al.*, Eds., ser. CEUR Workshop Proceedings, vol. 1488, CEUR-WS.org, 2015, pp. 63–74.

- [GGZ16] S. Gao, J. Gu, and C. Zaniolo, “RDF-TX: A fast, user-friendly system for querying the history of RDF knowledge bases”, in *Proceedings of the 19th International Conference on Extending Database Technology, EDBT 2016, Bordeaux, France, March 15-16, 2016, Bordeaux, France, March 15-16, 2016*, E. Pitoura *et al.*, Eds., OpenProceedings.org, 2016, pp. 269–280. DOI: 10.5441/002/edbt.2016.26.
- [Gar+20] C. Garcia-Arellano *et al.*, “Db2 event store: A purpose-built iot database engine”, *Proc. VLDB Endow.*, vol. 13, no. 12, pp. 3299–3312, 2020.
- [GBY09] B. Gedik, R. Bordawekar, and P. S. Yu, “Celljoin: A parallel stream join operator for the cell processor”, *VLDB J.*, vol. 18, no. 2, pp. 501–519, 2009. DOI: 10.1007/s00778-008-0116-z.
- [GAE06] T. M. Ghanem, W. G. Aref, and A. K. Elmagarmid, “Exploiting predicate-window semantics over data streams”, *SIGMOD Rec.*, vol. 35, no. 1, pp. 3–8, 2006. DOI: 10.1145/1121995.1121996.
- [Glo+20] N. Glombiewski, P. Götze, M. Körber, A. Morgen, and B. Seeger, “Designing an event store for a modern three-layer storage hierarchy”, *Datenbank-Spektrum*, vol. 20, no. 3, pp. 211–222, 2020. DOI: 10.1007/s13222-020-00356-6.
- [GÖ03] L. Golab and M. T. Özsu, “Processing sliding window multi-joins in continuous queries over data streams”, in *Proceedings of 29th International Conference on Very Large Data Bases, VLDB 2003, Berlin, Germany, September 9-12, 2003*, J. C. Freytag, P. C. Lockemann, S. Abiteboul, M. J. Carey, P. G. Selinger, and A. Heuer, Eds., Morgan Kaufmann, 2003, pp. 500–511. DOI: 10.1016/B978-012722442-8/50051-3.
- [GÖ05] L. Golab and M. T. Özsu, “Update-pattern-aware modeling and processing of continuous queries”, in *Proceedings of the ACM SIGMOD International Conference on Management of Data, Baltimore, Maryland, USA, June 14-16, 2005*, F. Özcan, Ed., ACM, 2005, pp. 658–669. DOI: 10.1145/1066157.1066232.
- [GC08] J. S. Gomes and H. Choi, “Adaptive optimization of join trees for multi-join queries over sensor streams”, *Inf. Fusion*, vol. 9, no. 3, pp. 412–424, 2008. DOI: 10.1016/j.inffus.2007.06.001.
- [Gra+18] P. Graubner *et al.*, “Multimodal complex event processing on mobile devices”, in *Proceedings of the 12th ACM International Conference on Distributed and Event-based Systems, DEBS 2018, Hamilton, New Zealand, June 25-29, 2018*, 2018, pp. 112–123. DOI: 10.1145/3210284.3210289.

- [Gro+16] M. Grossniklaus, D. Maier, J. Miller, S. Moorthy, and K. Tufte, “Frames: Data-driven windows”, in *Proceedings of the 10th ACM International Conference on Distributed and Event-based Systems, DEBS '16, Irvine, CA, USA, June 20 - 24, 2016*, A. Gal, M. Weidlich, V. Kalogeraki, and N. Venkasubramanian, Eds., ACM, 2016, pp. 13–24. DOI: 10.1145/2933267.2933304.
- [Gut84] A. Guttman, “R-trees: A dynamic index structure for spatial searching”, in *SIGMOD'84, Proceedings of Annual Meeting, Boston, Massachusetts, USA, June 18-21, 1984*, 1984, pp. 47–57. DOI: 10.1145/602259.602266.
- [HAB17] S. Hamdi, A. B. Abdallah, and M. H. Bedoui, “Real time qrs complex detection using dfa and regular grammar”, *BioMedical Engineering OnLine*, vol. 16, no. 1, p. 31, Feb. 2017. DOI: 10.1186/s12938-017-0322-2.
- [HOS18] M. Hasan, M. A. Orgun, and R. Schwitter, “A survey on real-time event detection from the twitter data stream”, *J. Inf. Sci.*, vol. 44, no. 4, pp. 443–463, 2018. DOI: 10.1177/0165551517698564.
- [HP16] S. Helmer and F. Persia, “Iseql, an interval-based surveillance event query language”, *Int. J. Multim. Data Eng. Manag.*, vol. 7, no. 4, pp. 1–21, 2016. DOI: 10.4018/IJMDEM.2016100101.
- [HV15] A. Hinze and A. Voisard, “EVA: an event algebra supporting complex event specification”, *Inf. Syst.*, vol. 48, pp. 1–25, 2015. DOI: 10.1016/j.is.2014.07.003.
- [Hir12] M. Hirzel, “Partition and compose: Parallel complex event processing”, in *Proceedings of the Sixth ACM International Conference on Distributed Event-Based Systems, DEBS 2012, Berlin, Germany, July 16-20, 2012*, F. Bry, A. Paschke, P. T. Eugster, C. Fetzer, and A. Behrend, Eds., ACM, 2012, pp. 191–200. DOI: 10.1145/2335484.2335506.
- [Hoß15] B. Hoßbach, “Design and implementation of a middleware for uniform, federated and dynamic event processing”, Ph.D. dissertation, University of Marburg, 2015.
- [Hoß+13] B. Hoßbach, N. Glombiewski, A. Morgen, F. Ritter, and B. Seeger, “JEPC: the java event processing connectivity”, *Datenbank-Spektrum*, vol. 13, no. 3, pp. 167–178, 2013. DOI: 10.1007/s13222-013-0133-y.
- [Hwu15] W.-m. W. Hwu, *Heterogeneous System Architecture: A new compute platform infrastructure*. Morgan Kaufmann, 2015.

- [20] *Insider Trading*, <https://www.investor.gov/introduction-investing/investing-basics/glossary/insider-trading>, accessed August 27, 2020, 2020.
- [16] *ISO/IEC TR 19075-5:2016, Information technology - Database languages - SQL Technical Reports - Part 5: Row Pattern Recognition in SQL*, <http://standards.iso.org/ittf/PubliclyAvailableStandards/>, accessed March 13, 2019, 2016.
- [Ji+16] Y. Ji, A. Nica, Z. Jerzak, G. Hackenbroich, and C. Fetzer, “Quality-driven disorder handling for concurrent windowed stream queries with shared operators”, in *Proceedings of the 10th ACM International Conference on Distributed and Event-based Systems, DEBS '16, Irvine, CA, USA, June 20 - 24, 2016*, 2016, pp. 25–36. DOI: 10.1145/2933267.2933307.
- [Jos+08] M. Joselli *et al.*, “Automatic dynamic task distribution between CPU and GPU for real-time systems”, in *Proceedings of the 11th IEEE International Conference on Computational Science and Engineering, CSE 2008, São Paulo, SP, Brazil, July 16-18, 2008*, 2008, pp. 48–55. DOI: 10.1109/CSE.2008.38.
- [JFB05] V. Josifovski, M. Fontoura, and A. Barta, “Querying XML streams”, *VLDB J.*, vol. 14, no. 2, pp. 197–210, 2005. DOI: 10.1007/s00778-004-0123-7.
- [KMS08] L. Kaghazian, D. McLeod, and R. Sadri, “Scalable complex pattern search in sequential data”, in *Proceedings of the 17th ACM Conference on Information and Knowledge Management, CIKM 2008, Napa Valley, California, USA, October 26-30, 2008*, 2008, pp. 1467–1468. DOI: 10.1145/1458082.1458336.
- [KTP10] R. Kandhan, N. Teletia, and J. M. Patel, “Sigmatch: Fast and scalable multi-pattern matching”, *Proc. VLDB Endow.*, vol. 3, no. 1, pp. 1173–1184, 2010. DOI: 10.14778/1920841.1920987.
- [KNV03] J. Kang, J. F. Naughton, and S. Viglas, “Evaluating window joins over unbounded streams”, in *Proceedings of the 19th International Conference on Data Engineering, March 5-8, 2003, Bangalore, India, 2003*, pp. 341–352. DOI: 10.1109/ICDE.2003.1260804.
- [KRM19] J. Karimov, T. Rabl, and V. Markl, “Ajoin: Ad-hoc stream joins at scale”, *Proc. VLDB Endow.*, vol. 13, no. 4, pp. 435–448, 2019.

- [Kar+13] T. Karnagel, D. Habich, B. Schlegel, and W. Lehner, “The hells-join: A heterogeneous stream join for extremely large windows”, in *Proceedings of the Ninth International Workshop on Data Management on New Hardware, DaMoN 1013, New York, NY, USA, June 24, 2013*, 2013, p. 2. DOI: 10.1145/2485278.2485280.
- [Kau+13] M. Kaufmann *et al.*, “Timeline index: A unified data structure for processing queries on temporal data in SAP HANA”, in *Proceedings of the ACM SIGMOD International Conference on Management of Data, SIGMOD 2013, New York, NY, USA, June 22-27, 2013*, K. A. Ross, D. Srivastava, and D. Papadias, Eds., ACM, 2013, pp. 1173–1184. DOI: 10.1145/2463676.2465293.
- [KAI17] M. S. Kester, M. Athanassoulis, and S. Idreos, “Access path selection in main-memory optimized data systems: Should I scan or should I probe?”, in *Proceedings of the 2017 ACM International Conference on Management of Data, SIGMOD Conference 2017, Chicago, IL, USA, May 14-19, 2017*, 2017, pp. 715–730. DOI: 10.1145/3035918.3064049.
- [Kha02] A. Khan, *501 Stock Market Tips and Guidelines*. Writers Club Pres, 2002.
- [Kie+13] J. Kietz, T. Scharrenbach, L. Fischer, A. Bernstein, and K. Nguyen, “Tef-sparql: The ddis query-language for time annotated event and fact triple-streams”, Technical report, University of Zurich, Department of Informatics, Tech. Rep., 2013.
- [KS95] N. Kline and R. T. Snodgrass, “Computing temporal aggregates”, in *Proceedings of the Eleventh International Conference on Data Engineering, March 6-10, 1995, Taipei, Taiwan, 1995*, pp. 222–231. DOI: 10.1109/ICDE.1995.380389.
- [KJP77] D. E. Knuth, J. H. M. Jr., and V. R. Pratt, “Fast pattern matching in strings”, *SIAM J. Comput.*, vol. 6, no. 2, pp. 323–350, 1977. DOI: 10.1137/0206024.
- [KS18] I. Kolchinsky and A. Schuster, “Join query optimization techniques for complex event processing applications”, *Proc. VLDB Endow.*, vol. 11, no. 11, pp. 1332–1345, 2018. DOI: 10.14778/3236187.3236189.
- [KS19] I. Kolchinsky and A. Schuster, “Real-time multi-pattern detection over event streams”, in *Proceedings of the 2019 International Conference on Management of Data, SIGMOD Conference 2019, Amsterdam, The Netherlands, June 30 - July 5, 2019*, 2019, pp. 589–606. DOI: 10.1145/3299869.3319869.

- [KSS15] I. Kolchinsky, I. Sharfman, and A. Schuster, “Lazy evaluation methods for detecting complex events”, in *Proceedings of the 9th ACM International Conference on Distributed Event-Based Systems, DEBS ’15, Oslo, Norway, June 29 - July 3, 2015*, 2015, pp. 34–45. DOI: 10.1145/2675743.2771832.
- [Kol+16] A. Koliousis, M. Weidlich, R. C. Fernandez, A. L. Wolf, P. Costa, and P. R. Pietzuch, “SABER: window-based hybrid stream processing for heterogeneous architectures”, in *Proceedings of the 2016 International Conference on Management of Data, SIGMOD Conference 2016, San Francisco, CA, USA, June 26 - July 01, 2016*, 2016, pp. 555–569. DOI: 10.1145/2882903.2882906.
- [Kör+19a] M. Körber, J. Eckstein, N. Glombiewski, and B. Seeger, “Event stream processing on heterogeneous system architecture”, in *Proceedings of the 15th International Workshop on Data Management on New Hardware, DaMoN 2019, Amsterdam, The Netherlands, 1 July 2019*, 2019, 3:1–3:10. DOI: 10.1145/3329785.3329933.
- [Kör+19b] M. Körber, N. Glombiewski, A. Morgen, and B. Seeger, “Tpstream: Low-latency and high-throughput temporal pattern matching on event streams”, *Distributed and Parallel Databases*, pp. 1–52, 2019. DOI: 10.1007/s10619-019-07272-z.
- [KGS18] M. Körber, N. Glombiewski, and B. Seeger, “Tpstream: Low-latency temporal pattern matching on event streams”, in *Proceedings of the 21st International Conference on Extending Database Technology, EDBT 2018, Vienna, Austria, March 26-29, 2018*, 2018, pp. 313–324. DOI: 10.5441/002/edbt.2018.28.
- [KGS21] M. Körber, N. Glombiewski, and B. Seeger, “Index-accelerated pattern matching in event stores”, in *SIGMOD ’21: International Conference on Management of Data, Virtual Event, China, June 20-25, 2021*, 2021, pp. 1023–1036. DOI: 10.1145/3448016.3457245.
- [KS04] J. Krämer and B. Seeger, “PIPES - A public infrastructure for processing and exploring streams”, in *Proceedings of the ACM SIGMOD International Conference on Management of Data, Paris, France, June 13-18, 2004*, 2004, pp. 925–926. DOI: 10.1145/1007568.1007699.
- [KS09] J. Krämer and B. Seeger, “Semantics and implementation of continuous sliding window queries over data streams”, *ACM Trans. Database Syst.*, vol. 34, no. 1, 4:1–4:49, 2009. DOI: 10.1145/1508857.1508861.

- [KLP75] H. T. Kung, F. Luccio, and F. P. Preparata, “On finding the maxima of a set of vectors”, *J. ACM*, vol. 22, no. 4, pp. 469–476, 1975. DOI: 10.1145/321906.321910.
- [Li+05a] J. Li, D. Maier, K. Tufte, V. Papadimos, and P. A. Tucker, “No pane, no gain: Efficient evaluation of sliding-window aggregates over data streams”, *SIGMOD Rec.*, vol. 34, no. 1, pp. 39–44, 2005. DOI: 10.1145/1058150.1058158.
- [Li+05b] J. Li, D. Maier, K. Tufte, V. Papadimos, and P. A. Tucker, “Semantics and evaluation techniques for window aggregates in data streams”, in *Proceedings of the ACM SIGMOD International Conference on Management of Data, Baltimore, Maryland, USA, June 14-16, 2005*, 2005, pp. 311–322. DOI: 10.1145/1066157.1066193.
- [Li+11] M. Li, M. Mani, E. A. Rundensteiner, and T. Lin, “Complex event pattern detection over streams with interval-based temporal semantics”, in *Proceedings of the Fifth ACM International Conference on Distributed Event-Based Systems, DEBS 2011, New York, NY, USA, July 11-15, 2011*, 2011, pp. 291–302. DOI: 10.1145/2002259.2002297.
- [LWK14] B. Lohrmann, D. Warneke, and O. Kao, “Nephele streaming: Stream processing under qos constraints at scale”, *Clust. Comput.*, vol. 17, no. 1, pp. 61–78, 2014. DOI: 10.1007/s10586-013-0281-8.
- [Luc98] D. C. Luckham, “Rapide: A language and toolset for causal event modeling of distributed system architectures”, in *Worldwide Computing and Its Applications, International Conference, WWCA '98, Second International Conference, Tsukuba, Japan, March 4-5, 1998, Proceedings*, 1998, pp. 88–96. DOI: 10.1007/3-540-64216-1_42.
- [MM93] U. Manber and E. W. Myers, “Suffix arrays: A new method for on-line string searches”, *SIAM J. Comput.*, vol. 22, no. 5, pp. 935–948, 1993. DOI: 10.1137/0222058.
- [MCS88] M. V. Mannino, P. Chu, and T. Sager, “Statistical profile estimation in database systems”, *ACM Comput. Surv.*, vol. 20, no. 3, pp. 191–221, 1988. DOI: 10.1145/62061.62063.
- [MC11] A. Margara and G. Cugola, “Processing flows of information: From data stream to complex event processing”, in *Proceedings of the Fifth ACM International Conference on Distributed Event-Based Systems, DEBS 2011, New York, NY, USA, July 11-15, 2011*, 2011, pp. 359–360. DOI: 10.1145/2002259.2002307.

- [May+16] R. Mayer, C. Mayer, M. A. Tariq, and K. Rothermel, “Graphcep: Real-time data analytics using parallel complex event and graph processing”, in *Proceedings of the 10th ACM International Conference on Distributed and Event-based Systems, DEBS '16, Irvine, CA, USA, June 20 - 24, 2016*, 2016, pp. 309–316. DOI: 10.1145/2933267.2933509.
- [MD89] D. R. McCarthy and U. Dayal, “The architecture of an active data base management system”, in *Proceedings of the 1989 ACM SIGMOD International Conference on Management of Data, Portland, Oregon, USA, May 31 - June 2, 1989*, 1989, pp. 215–224. DOI: 10.1145/67544.66946.
- [McC76] E. M. McCreight, “A space-economical suffix tree construction algorithm”, *J. ACM*, vol. 23, no. 2, pp. 262–272, 1976. DOI: 10.1145/321941.321946.
- [MM09] Y. Mei and S. Madden, “Zstream: A cost-based query processor for adaptively detecting composite events”, in *Proceedings of the ACM SIGMOD International Conference on Management of Data, SIGMOD 2009, Providence, Rhode Island, USA, June 29 - July 2, 2009*, 2009, pp. 193–206. DOI: 10.1145/1559845.1559867.
- [Moe98] G. Moerkotte, “Small materialized aggregates: A light weight index structure for data warehousing”, in *VLDB'98, Proceedings of 24rd International Conference on Very Large Data Bases, August 24-27, 1998, New York City, New York, USA*, 1998, pp. 476–487.
- [MLI03] B. Moon, I. F. V. López, and V. Immanuel, “Efficient algorithms for large-scale temporal aggregation”, *IEEE Trans. Knowl. Data Eng.*, vol. 15, no. 3, pp. 744–759, 2003. DOI: 10.1109/TKDE.2003.1198403.
- [Mou+15] R. Moussalli, I. Absalyamov, M. R. Vieira, W. A. Najjar, and V. J. Tsotras, “High performance FPGA and GPU complex pattern matching over spatio-temporal streams”, *GeoInformatica*, vol. 19, no. 2, pp. 405–434, 2015. DOI: 10.1007/s10707-014-0217-3.
- [Muk+16] S. Mukherjee, Y. Sun, P. Blinzer, A. K. Ziabari, and D. R. Kaeli, “A comprehensive performance analysis of HSA and opencl 2.0”, in *2016 IEEE International Symposium on Performance Analysis of Systems and Software, ISPASS 2016, Uppsala, Sweden, April 17-19, 2016*, 2016, pp. 183–193. DOI: 10.1109/ISPASS.2016.7482093.

-
- [MTA09] R. Müller, J. Teubner, and G. Alonso, “Streams on wires - A query compiler for fpgas”, *Proc. VLDB Endow.*, vol. 2, no. 1, pp. 229–240, 2009. DOI: 10.14778/1687627.1687654.
 - [Mut+98] P. Muth, P. E. O’Neil, A. Pick, and G. Weikum, “Design, implementation, and performance of the LHAM log-structured history data access method”, in *VLDB’98, Proceedings of 24rd International Conference on Very Large Data Bases, August 24-27, 1998, New York City, New York, USA*, 1998, pp. 452–463.
 - [Nav14] G. Navarro, “Wavelet trees for all”, *J. Discrete Algorithms*, vol. 25, pp. 2–20, 2014. DOI: 10.1016/j.jda.2013.07.004.
 - [Now+18] M. Nowakiewicz, E. Boutin, E. N. Hanson, R. Walzer, and A. Katiappally, “Bipie: Fast selection and aggregation on encoded data using operator specialization”, in *Proceedings of the 2018 International Conference on Management of Data, SIGMOD Conference 2018, Houston, TX, USA, June 10-15, 2018*, 2018, pp. 1447–1459. DOI: 10.1145/3183713.3190658.
 - [ONe+96] P. E. O’Neil, E. Cheng, D. Gawlick, and E. J. O’Neil, “The log-structured merge-tree (lsm-tree)”, *Acta Informatica*, vol. 33, no. 4, pp. 351–385, 1996. DOI: 10.1007/s002360050048.
 - [Org18] I. C. A. Organization. “Icaolong-termtrafficroforecasts - passenger and cargo”. (2018), [Online]. Available: https://www.icao.int/sustainability/Documents/LTF_Charts-Results_2018edition.pdf (visited on 11/18/2020).
 - [PBH17] F. Persia, F. Bettini, and S. Helmer, “An interactive framework for video surveillance event detection and modeling”, in *Proceedings of the 2017 ACM on Conference on Information and Knowledge Management, CIKM 2017, Singapore, November 06 - 10, 2017*, E. Lim *et al.*, Eds., ACM, 2017, pp. 2515–2518. DOI: 10.1145/3132847.3133164.
 - [PHD16] D. Piatov, S. Helmer, and A. Dignös, “An interval join optimized for modern hardware”, in *32nd IEEE International Conference on Data Engineering, ICDE 2016, Helsinki, Finland, May 16-20, 2016*, IEEE Computer Society, 2016, pp. 1098–1109. DOI: 10.1109/ICDE.2016.7498316.
 - [PBS15] M. Pinnecke, D. Brönske, and G. Saake, “Toward GPU accelerated data stream processing”, in *Proceedings of the 27th GI-Workshop Grundlagen von Datenbanken, Gommern, Germany, May 26-29, 2015*, 2015, pp. 78–83.

- [Pop+16] O. Poppe, C. Lei, E. A. Rundensteiner, and D. J. Dougherty, “Context-aware event stream analytics”, in *Proceedings of the 19th International Conference on Extending Database Technology, EDBT 2016, Bordeaux, France, March 15-16, 2016, Bordeaux, France, March 15-16, 2016*, E. Pitoura *et al.*, Eds., OpenProceedings.org, 2016, pp. 413–424. DOI: 10.5441/002/edbt.2016.38.
- [Pop+17] O. Poppe, C. Lei, E. A. Rundensteiner, and D. Maier, “GRETA: graph-based real-time event trend aggregation”, *Proc. VLDB Endow.*, vol. 11, no. 1, pp. 80–92, 2017. DOI: 10.14778/3151113.3151120.
- [Pop+19] O. Poppe, C. Lei, E. A. Rundensteiner, and D. Maier, “Event trend aggregation under rich event matching semantics”, in *Proceedings of the 2019 International Conference on Management of Data, SIGMOD Conference 2019, Amsterdam, The Netherlands, June 30 - July 5, 2019*, 2019, pp. 555–572. DOI: 10.1145/3299869.3319862.
- [Ram+98] R. Ramakrishnan, D. Donjerkovic, A. Ranganathan, K. S. Beyer, and M. Krishnaprasad, “SRQL: sorted relational query language”, in *10th International Conference on Scientific and Statistical Database Management, Proceedings, Capri, Italy, July 1-3, 1998*, 1998, pp. 84–95. DOI: 10.1109/SSDM.1998.688114.
- [RLR16] M. Ray, C. Lei, and E. A. Rundensteiner, “Scalable pattern sharing on event streams”, in *Proceedings of the 2016 International Conference on Management of Data, SIGMOD Conference 2016, San Francisco, CA, USA, June 26 - July 01, 2016*, 2016, pp. 495–510. DOI: 10.1145/2882903.2882947.
- [RBP15] P. S. Rodrigo, H. M. N. D. Bandara, and S. Perera, “Accelerating complex event processing through gpus”, in *22nd IEEE International Conference on High Performance Computing, HiPC 2015, Bengaluru, India, December 16-19, 2015*, 2015, pp. 325–334. DOI: 10.1109/HiPC.2015.36.
- [RG83] M. A. Rosenman and J. S. Gero, “Pareto optimal serial dynamic programming”, *Engineering Optimization*, vol. 6, no. 4, pp. 177–183, 1983. DOI: 10.1080/03052158308902467.
- [Run+15] E. A. Rundensteiner *et al.*, “Exploiting sharing opportunities for real-time complex event analytics”, *IEEE Data Eng. Bull.*, vol. 38, no. 4, pp. 82–93, 2015.

- [SJ11] M. Sadoghi and H. Jacobsen, “Be-tree: An index structure to efficiently match boolean expressions over high-dimensional discrete space”, in *Proceedings of the ACM SIGMOD International Conference on Management of Data, SIGMOD 2011, Athens, Greece, June 12-16, 2011*, 2011, pp. 637–648. DOI: 10.1145/1989323.1989390.
- [SJ13] M. Sadoghi and H. Jacobsen, “Analysis and optimization for boolean expression indexing”, *ACM Trans. Database Syst.*, vol. 38, no. 2, 8:1–8:47, 2013. DOI: 10.1145/2487259.2487260.
- [Sad+01] R. Sadri, C. Zaniolo, A. M. Zarkesh, and J. Adibi, “Optimization of sequence queries in database systems”, in *Proceedings of the Twentieth ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems, May 21-23, 2001, Santa Barbara, California, USA*, 2001. DOI: 10.1145/375551.375563.
- [SG11] M. A. Sakr and R. H. Güting, “Spatiotemporal pattern queries”, *GeoInformatica*, vol. 15, no. 3, pp. 497–540, 2011. DOI: 10.1007/s10707-010-0114-3.
- [Sch+14] M. Schäfer, M. Strohmeier, V. Lenders, I. Martinovic, and M. Wilhelm, “Bringing up opensky: A large-scale ADS-B sensor network for research”, in *IPSN’14*, 2014, pp. 83–94.
- [SMP09] N. P. Schultz-Møller, M. Migliavacca, and P. R. Pietzuch, “Distributed complex event processing with query rewriting”, in *Proceedings of the Third ACM International Conference on Distributed Event-Based Systems, DEBS 2009, Nashville, Tennessee, USA, July 6-9, 2009*, A. S. Gokhale and D. C. Schmidt, Eds., ACM, 2009. DOI: 10.1145/1619258.1619264.
- [Sei+19] M. Seidemann, N. Glombiewski, M. Körber, and B. Seeger, “Chronicledb: A high-performance event store”, *ACM Trans. Database Syst.*, vol. 44, no. 4, 13:1–13:45, 2019. DOI: 10.1145/3342357.
- [SS17] M. Seidemann and B. Seeger, “Chronicledb: A high-performance event store”, in *Proceedings of the 20th International Conference on Extending Database Technology, EDBT 2017, Venice, Italy, March 21-24, 2017*, 2017, pp. 144–155. DOI: 10.5441/002/edbt.2017.14.
- [Sel+79] P. G. Selinger, M. M. Astrahan, D. D. Chamberlin, R. A. Lorie, and T. G. Price, “Access path selection in a relational database management system”, in *Proceedings of the 1979 ACM SIGMOD International Conference on Management of Data, Boston, Massachusetts, USA, May 30 - June 1, 1979*, pp. 23–34. DOI: 10.1145/582095.582099.

- [SLR95] P. Seshadri, M. Livny, and R. Ramakrishnan, “SEQ: A model for sequence databases”, in *Proceedings of the Eleventh International Conference on Data Engineering, March 6-10, 1995, Taipei, Taiwan, 1995*, pp. 232–239. DOI: 10.1109/ICDE.1995.380388.
- [SCL18] A. U. Shein, P. K. Chrysanthis, and A. Labrinidis, “Slickdeque: High throughput and low latency incremental sliding-window aggregation”, in *Proceedings of the 21st International Conference on Extending Database Technology, EDBT 2018, Vienna, Austria, March 26-29, 2018, 2018*, pp. 397–408. DOI: 10.5441/002/edbt.2018.35.
- [SW04] U. Srivastava and J. Widom, “Memory-limited execution of windowed stream joins”, in *(e)Proceedings of the Thirtieth International Conference on Very Large Data Bases, VLDB 2004, Toronto, Canada, August 31 - September 3 2004, 2004*, pp. 324–335. DOI: 10.1016/B978-012088469-8.50031-0.
- [Sun+16] Y. Sun *et al.*, “Hetero-mark, a benchmark suite for CPU-GPU collaborative computing”, in *2016 IEEE International Symposium on Workload Characterization, IISWC 2016, Providence, RI, USA, September 25-27, 2016, 2016*, pp. 13–22. DOI: 10.1109/IISWC.2016.7581262.
- [THS17] K. Tangwongsan, M. Hirzel, and S. Schneider, “Low-latency sliding-window aggregation in worst-case constant time”, in *Proceedings of the 11th ACM International Conference on Distributed and Event-based Systems, DEBS 2017, Barcelona, Spain, June 19-23, 2017, 2017*, pp. 66–77. DOI: 10.1145/3093742.3093925.
- [Tan+15] K. Tangwongsan, M. Hirzel, S. Schneider, and K. Wu, “General incremental sliding-window aggregation”, *Proc. VLDB Endow.*, vol. 8, no. 7, pp. 702–713, 2015. DOI: 10.14778/2752939.2752940.
- [TM11] J. Teubner and R. Müller, “How soccer players would do stream joins”, in *Proceedings of the ACM SIGMOD International Conference on Management of Data, SIGMOD 2011, Athens, Greece, June 12-16, 2011, 2011*, pp. 625–636. DOI: 10.1145/1989323.1989389.
- [The+20] G. Theodorakis, A. Koliosis, P. R. Pietzuch, and H. Pirk, “Lightsaber: Efficient window aggregation on multi-core processors”, in *Proceedings of the 2020 International Conference on Management of Data, SIGMOD Conference 2020, online conference [Portland, OR, USA], June 14-19, 2020, 2020*, pp. 2505–2521. DOI: 10.1145/3318464.3389753.

- [TC11] D. Tsang and S. Chawla, “A robust index for regular expression queries”, in *Proceedings of the 20th ACM Conference on Information and Knowledge Management, CIKM 2011, Glasgow, United Kingdom, October 24-28, 2011*, 2011, pp. 2365–2368. DOI: 10.1145/2063576.2063968.
- [UF00] T. Urhan and M. J. Franklin, “Xjoin: A reactively-scheduled pipelined join operator”, *IEEE Data Eng. Bull.*, vol. 23, no. 2, pp. 27–33, 2000.
- [VG14] F. Valdés and R. H. Güting, “Index-supported pattern matching on symbolic trajectories”, in *Proceedings of the 22nd ACM SIGSPATIAL International Conference on Advances in Geographic Information Systems, Dallas/Fort Worth, TX, USA, November 4-7, 2014*, 2014, pp. 53–62. DOI: 10.1145/2666310.2666402.
- [VSS11] U. Verner, A. Schuster, and M. Silberstein, “Processing data streams with hard real-time constraints on heterogeneous systems”, in *Proceedings of the 25th International Conference on Supercomputing, 2011, Tucson, AZ, USA, May 31 - June 04, 2011*, 2011, pp. 120–129. DOI: 10.1145/1995896.1995915.
- [VNB03] S. Viglas, J. F. Naughton, and J. Burger, “Maximizing the output rate of multi-way join queries over streaming information sources”, in *Proceedings of 29th International Conference on Very Large Data Bases, VLDB 2003, Berlin, Germany, September 9-12, 2003*, 2003, pp. 285–296. DOI: 10.1016/B978-012722442-8/50033-1.
- [WR09] S. Wang and E. A. Rundensteiner, “Scalable stream join processing with expensive predicates: Workload distribution and adaptation by time-slicing”, in *EDBT 2009, 12th International Conference on Extending Database Technology, Saint Petersburg, Russia, March 24-26, 2009, Proceedings*, 2009, pp. 299–310. DOI: 10.1145/1516360.1516396.
- [Wan+06] S. Wang, E. A. Rundensteiner, S. Ganguly, and S. Bhatnagar, “State-slice: New paradigm of multi-query optimization of window-based stream queries”, in *Proceedings of the 32nd International Conference on Very Large Data Bases, Seoul, Korea, September 12-15, 2006*, 2006, pp. 619–630.
- [Wei73] P. Weiner, “Linear pattern matching algorithms”, in *14th Annual Symposium on Switching and Automata Theory, Iowa City, Iowa, USA, October 15-17, 1973*, 1973, pp. 1–11. DOI: 10.1109/SWAT.1973.13.

-
- [WTA10] L. Woods, J. Teubner, and G. Alonso, “Complex event detection at wire speed with fpgas”, *Proc. VLDB Endow.*, vol. 3, no. 1, pp. 660–669, 2010. DOI: 10.14778/1920841.1920926.
- [WTA11] L. Woods, J. Teubner, and G. Alonso, “Real-time pattern matching with fpgas”, in *Proceedings of the 27th International Conference on Data Engineering, ICDE 2011, April 11-16, 2011, Hannover, Germany*, 2011, pp. 1292–1295. DOI: 10.1109/ICDE.2011.5767937.
- [WDR06] E. Wu, Y. Diao, and S. Rizvi, “High-performance complex event processing over streams”, in *Proceedings of the ACM SIGMOD International Conference on Management of Data, Chicago, Illinois, USA, June 27-29, 2006*, 2006, pp. 407–418. DOI: 10.1145/1142473.1142520.
- [YW03] J. Yang and J. Widom, “Incremental computation and maintenance of temporal aggregates”, *VLDB J.*, vol. 12, no. 3, pp. 262–283, 2003. DOI: 10.1007/s00778-003-0107-z.
- [Zah+16] M. Zaharia *et al.*, “Apache spark: A unified engine for big data processing”, *Commun. ACM*, vol. 59, no. 11, pp. 56–65, 2016. DOI: 10.1145/2934664.
- [Zeu+19] S. Zeuch *et al.*, “Analyzing efficient stream processing on modern hardware”, *Proc. VLDB Endow.*, vol. 12, no. 5, pp. 516–530, 2019. DOI: 10.14778/3303753.3303758.
- [ZTS02] D. Zhang, V. J. Tsotras, and B. Seeger, “Efficient temporal join processing using indices”, in *Proceedings of the 18th International Conference on Data Engineering, San Jose, CA, USA, February 26 - March 1, 2002*, 2002, pp. 103–113. DOI: 10.1109/ICDE.2002.994701.
- [ZCT14] D. Zhang, C. Chan, and K. Tan, “An efficient publish/subscribe index for ecommerce databases”, *Proc. VLDB Endow.*, vol. 7, no. 8, pp. 613–624, 2014. DOI: 10.14778/2732296.2732298.
- [Zha+20] F. Zhang, L. Yang, S. Zhang, B. He, W. Lu, and X. Du, “Finestream: Fine-grained window-based stream processing on CPU-GPU integrated architectures”, in *2020 USENIX Annual Technical Conference, USENIX ATC 2020, July 15-17, 2020*, 2020, pp. 633–647.
- [Zha+17] F. Zhang, J. Zhai, B. He, S. Zhang, and W. Chen, “Understanding co-running behaviors on integrated CPU/GPU architectures”, *IEEE Trans. Parallel Distributed Syst.*, vol. 28, no. 3, pp. 905–918, 2017. DOI: 10.1109/TPDS.2016.2586074.

- [Zha+21] F. Zhang, J. Zhai, B. Wu, B. He, W. Chen, and X. Du, “Automatic irregularity-aware fine-grained workload partitioning on integrated architectures”, *IEEE Trans. Knowl. Data Eng.*, vol. 33, no. 3, pp. 867–881, 2021. DOI: 10.1109/TKDE.2019.2940184.
- [ZDI10] H. Zhang, Y. Diao, and N. Immerman, “Recognizing patterns in streams with imprecise timestamps”, *Proc. VLDB Endow.*, vol. 3, no. 1, pp. 244–255, 2010. DOI: 10.14778/1920841.1920875.
- [ZDI14] H. Zhang, Y. Diao, and N. Immerman, “On complexity and optimization of expensive queries in complex event processing”, in *International Conference on Management of Data, SIGMOD 2014, Snowbird, UT, USA, June 22-27, 2014*, 2014, pp. 217–228. DOI: 10.1145/2588555.2593671.
- [ZM11] Y. Zhang and F. Mueller, “Gstream: A general-purpose data streaming framework on GPU clusters”, in *International Conference on Parallel Processing, ICPP 2011, Taipei, Taiwan, September 13-16, 2011*, 2011, pp. 245–254. DOI: 10.1109/ICPP.2011.22.
- [ZU99] D. Zimmer and R. Unland, “On the semantics of complex events in active database management systems”, in *Proceedings of the 15th International Conference on Data Engineering, Sydney, Australia, March 23-26, 1999*, 1999, pp. 392–399. DOI: 10.1109/ICDE.1999.754955.

List of Figures

1.1	A dataflow graph consisting of two sources, four operators and one sink (a), and a possible deployment of this graph in a cluster of 5 nodes (b).	5
2.1	Example for the content of a sliding (a) and tumbling (b) count window of $size = 2$.	17
2.2	Example for the content of a sliding (a) and tumbling (b) time window of $size = 2$.	18
2.3	Example of a filter with $\phi = v > 3$ applied to an event stream.	19
2.4	Example of a windowed sum-aggregation with a sliding time window of $size = 2$.	21
2.5	Example of a windowed join between stream E^1 and E^2 using two sliding count windows of $size = 2$ and $\phi = E^1.v \leq E^2.v$ as the join predicate.	22
2.6	Matching subsequences for each of the three matching strategies applied to an symbol trace.	26
3.1	Query runtime for different pattern evaluation strategies as a function of query complexity.	30
3.2	TAB^+ -tree index layout with lightweight indexing.	35
3.3	Dataflow of the generic replay operator.	36
3.4	Index-based processing of sliding window aggregates: naive approach (left) and optimized approach (right).	38
3.5	Illustration of performed partial aggregate merges during a temporal aggregation query.	40
3.6	Definition of the insider trading detection pattern using the <i>contiguous</i> matching strategy. The shaded areas highlight for every symbol the fractions of the stream where the corresponding predicate holds.	42
3.7	Index-based matching of the example pattern ABC^*D highlighting the replayed fraction of the stream when using (a) an index for symbol A , (b) indexes for symbols A and B , and (c) indexes for symbols A, B and D .	43
3.8	Processing time of sliding time window aggregation for varying values of slide using different event sizes.	59
3.9	Query runtimes (a) and speedup of different index selection strategies compared to replay (b) as a function of the query complexity.	63
3.10	Err_{PIO} as a function of the query complexity.	64

3.11	Average index selection time and number of examined index combinations for Optimal as a function of the query complexity.	64
3.12	Err_{PIO} (a), and query processing time (b) for all execution plans of a single query with $m = 8$	65
3.13	Query runtimes (a) and speedup of different index selection strategies compared to replay (b) for varying data distributions.	66
3.14	Err_{PIO} for varying data distributions without (left) and with (right) utilizing CPM.	66
3.15	Query runtimes for real-world data sets.	67
3.16	Estimated and actual primary I/O (a,b) and processing time with estimated total cost (c,d) for all execution plans of both crime query variants.	68
3.17	Processing Time (a) and Err_{PIO} (b) with varying CPM granularity for the landing pattern.	69
4.1	CPU kernel execution methods	75
4.2	HSA thread group hierarchy	77
4.3	An example of logical queue addresses	79
4.4	An example how results are written by the HSA-based filter implementation	80
4.5	An example of concurrent writes in the join operator	84
4.6	NFA representation of the example pattern $A B^* C$	85
4.7	Example for the removal of two PMs (PM_1, PM_4) in the dense array.	87
4.8	Throughput-comparison of the filter operation between Apache Flink and the variants of our processing framework.	90
4.9	Comparison of kernel turnaround time	90
4.10	Speedup of filter implementations compared to the single-threaded implementation	91
4.11	Speedup of aggregation implementations compared to the single-threaded implementation with varying window sizes	92
4.12	Speedup of join implementations compared to the single-threaded implementation	93
4.13	Speedup of pattern matching approaches compared to the single-threaded implementation using the <i>skip-till-any</i> matching strategy.	95
4.14	Speedup of pattern matching approaches compared to the single-threaded implementation using the <i>skip-till-next</i> matching strategy.	97
5.1	Detecting aggressive driving with situations.	99
5.2	$TPStream$ Architecture	111
5.3	Temporal Matching via Range Queries	115

5.4	Earliest detection time ($tp_{min}(P)$) of different <i>temporal configurations</i> for the sample pattern P	118
5.5	Overview partition parallel <i>TPStream</i>	125
5.6	Overview: parallel processing of unpartitioned data	126
5.7	Example of the buffer implementation with logical addressing.	131
5.8	Processing time for aggressive driver detection as a function of the input size.	139
5.9	Processing time for disconnected pattern detection as a function of the window size	140
5.10	Processing time for various query patterns	140
5.11	Relative detection latency per TR compared to <i>ISEQ</i>	142
5.12	Comparison of result latency (a) under maximum possible throughput as a function of the window size, (b) under varying event rates with a fixed size window (EL = event latency, PL = processing latency).	143
5.13	Quality of the initial plans for Q1 – Q3	144
5.14	Throughput comparison: dynamic plan adaption vs. best initial plans	144
5.15	SpeedUp compared to single-threaded mode for partition parallel execution of the aggressive drivers query with with different batch sizes and (a) data loaded from file, (b) preloaded data	146
5.16	Processing time for parallel disconnected pattern detection for varying batch and window sizes.	147
5.17	Relative processing times spent in the different phases of unpartitioned parallel processing for varying batch and window sizes.	148
5.18	Evaluation of the configured batch size and number of threads over time for different queries.	150
5.19	Speed up in a cluster for various queries.	152

List of Tables

3.1	Symbols used in the cost estimation for windowed aggregation. . . .	39
3.2	Pattern rewrite rules.	56
4.1	Result of the check phase and the corresponding action for each of the three matching strategies.	86
4.2	Details of the evaluation platform	89
5.1	Allen’s Interval Algebra	105
5.2	Temporal relations R and their prefix groups G with their earliest detection times $tr_{min}(R)$ and $tg_{min}(G)$	116
5.3	Initial estimates for the selectivity of TRs (φ_R)	122
5.4	Overview of parameters and sensors for the auto-tune component. .	132
5.5	Specification of the queries executed for the auto-tuning experiment	149
5.6	Hardware specification of the cluster	151

List of Algorithms

3.1	Create Replay Intervals	46
3.2	SelectIndexes	51
5.1	DeriveSituations	112
5.2	UpdateMatcher	113
5.3	PerformMatch	113
5.4	Low-Latency MatcherUpdate	120
5.5	MergeSituations	129

List of Abbreviations

BB	basic block
BS	basic symbol
CEP	complex event processing
CERN	European Organization for Nuclear Research
CPM	coarse pattern map
CPU	central processing unit
CQ	continuous query
DAG	directed acyclic graph
DBMS	database management system
dGPU	dedicated GPU
ECG	electrocardiogram
EMA	exponential moving average
ESP	event stream processing
FPGA	field programmable gate array
GPU	graphics processing unit
HDD	hard disk drive
HSA	Heterogeneous System Architecture
iGPU	integrated GPU
JEPC	Java Event Processing Connectivity
JVM	Java Virtual Machine
KS	Kleene-star symbol
MR	MATCH_RECOGNIZE
NFA	nondeterministic finite automaton
PCIe	PCI express
PM	partial match
SIMD	single instruction multiple data
SIMT	single instruction multiple thread
SMA	small materialized aggregate
SPE	stream processing engine

List of Abbreviations

SSD	solid state drive
SVM	shared virtual memory
TC	temporal constraint
TP	temporal pattern
TR	temporal relation
WF	wavefront