Design and Implementation of a Middleware for Uniform, Federated and Dynamic Event Processing

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

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

Bastian Hoßbach
aus Hessisch-Lichtenau

Marburg an der Lahn, 2015

Vom Fachbereich Mathematik und Informatik der
Philipps-Universität Marburg (Hochschulkennziffer: 1180)
als Dissertation am 12. Juni 2015 angenommen.

Erstgutachter:	Prof. Dr. Bernhard Seeger
Zweitgutachter:	Prof. Dr. Bernd Freisleben
Weitere Mitglieder der Prüfungskommission:	Prof. Dr. Gabriele Taentzer
	Prof. Dr. Thorsten Thormählen

Tag der mündlichen Prüfung am 17. Juni 2015.

Abstract

In recent years, real-time processing of massive event streams has become an important topic in the area of data analytics. It will become even more important in the future due to cheap sensors, a growing amount of devices and their ubiquitous interconnection also known as the Internet of Things (IoT). Academia, industry and the open source community have developed several event processing (EP) systems that allow users to define, manage and execute continuous queries over event streams. They achieve a significantly better performance than the traditional “store-then-process” approach in which events are first stored and indexed in a database. Because EP systems have different roots and because of the lack of standardization, the system landscape became highly heterogenous. Today’s EP systems differ in APIs, execution behaviors and query languages. This thesis presents the design and implementation of a novel middleware that abstracts from different EP systems and provides a uniform API, execution behavior and query language to users and developers. As a consequence, the presented middleware overcomes the problem of vendor lock-in and different EP systems are enabled to cooperate with each other. In practice, event streams differ dramatically in volume and velocity. We show therefore how the middleware can connect to not only different EP systems, but also database systems and a native implementation. Emerging applications such as the IoT raise novel challenges and require EP to be more dynamic. We present extensions to the middleware that enable self-adaptivity which is needed in context-sensitive applications and those that deal with constantly varying sets of event producers and consumers. Lastly, we extend the middleware to fully support the processing of events containing spatial data and to be able to run distributed in the form of a federation of heterogenous EP systems.

Zusammenfassung

In den letzten Jahren hat sich die Echtzeitverarbeitung von massiven Ereignisströmen zu einem wichtigen Thema im Bereich der Datenanalyse entwickelt. Sie wird in Zukunft sogar noch wichtiger aufgrund günstiger Sensoren, einer wachsenden Zahl von Geräten und deren allgegenwärtigen Vernetzung auch bekannt als das Internet der Dinge (IdD). Hochschulen, Industrie und die Open Source Gemeinschaft haben viele Systeme zur Ereignisverarbeitung (EV) entwickelt, die es Benutzern ermöglichen kontinuierliche Anfragen auf Ereignisströmen zu definieren, zu verwalten und auszuführen. Diese erzielen eine deutlich bessere Leistung als der traditionelle Ansatz, in dem Ereignisse zuerst in einer Datenbank gespeichert und indexiert werden. Da EV-Systeme unterschiedliche Wurzeln haben und es keine Standards gibt, wurde die Systemlandschaft sehr heterogen. EV-Systeme unterscheiden sich in APIs, Verhalten und Anfragesprachen. In dieser Arbeit werden der Entwurf und die Umsetzung einer neuartigen Middleware präsentiert, die von unterschiedlichen EV-Systemen abstrahiert und Benutzern sowie Entwicklern eine einheitliche Plattform anbietet. Damit überwindet die Middleware das Problem des Vendor-Lock-in und ermöglicht unterschiedlichen EV-Systemen die Zusammenarbeit. In der Praxis unterscheiden sich Ereignisströme deutlich in Volumen und Geschwindigkeit. Wir zeigen wie sich die Middleware nicht nur zu EV-Systemen verbinden kann, sondern auch zu Datenbanken und einer nativen Implementierung. Anwendungen wie das IdD führen zu besonderen Herausforderungen und benötigen eine dynamischere EV. Wir stellen Erweiterungen der Middleware vor, die Selbst-Adaptivität ermöglichen, welche in kontextabhängigen Anwendungen und jenen, die sich mit sich ständig ändernden Mengen von Quellen und Senken befassen müssen, benötigt wird. Schließlich erweitern wir die Middleware um die Unterstützung der Verarbeitung von räumlichen Daten und die Fähigkeit verteilt als Föderation heterogener EV-Systeme zu laufen.

Acknowledgements

First of all, I would like to thank my thesis advisor Prof. Dr. Bernhard Seeger for supervising my research during the last years. In particular, I am deeply grateful for all our fruitful discussions. A conservation with him was always inspiring and helped me to solve problems as well as to generate new ideas. I also wish to thank Prof. Dr. Bernd Freisleben who was kindly willing to review my thesis. At the department, I would like to thank my present and former colleagues of the database systems and distributed systems research groups as well as Mechthild Keßler. I would particularly like to express a special thank you to Marc Seidemann who took over the development of the B^+-tree event store. The Java Event Processing Connectivity would not have been possible without many committed students who assisted in the implementation. In alphabetical order I would like to thank Sebastian Brust, Nikolaus Glombiewski, Marcus von Keil, Christian Koch, Michael Körber, Andreas Morgen, Marcus Pinnecke, Franz Ritter, Anneke Walter and Stephan Wöllauer. Last but not least, I would also like to gratefully acknowledge the support of the German Federal Ministry of Education and Research (Bundesministerium für Bildung und Forschung, BMBF) under grant numbers 01BY1206A and 16BY1206A.

Contents

Part I

Introduction

1

Big Data in Motion

Outline

1.1 Introduction

Over the last years, the processing and analysis of streaming data in real-time evolved to an established and important class of data management in a broad range of domains. Especially many business applications are facing real-time data more and more. In such applications, information is updated continuously and often at a very high rate. Hence, the information comes in the form of data streams that can be massive in terms of volume and velocity. Today's stream processing technology does not require data to be stored first as in traditional data management. Instead, new data is processed efficiently by continuous queries on-the-fly. In particular, the processing is done completely in main memory and incrementally. On new data, the output of a running continuous query is updated and, thus, forms a new stream of data. Because data streams can be huge or even unbounded, sliding windows are used to limit the scope of continuous queries to finite and small sections of data streams. Those sliding windows keep only the most recent data which is the most relevant data in nearly all cases and continuous queries are evaluated only on the current content of the associated sliding windows. To motivate this technology, we present different application domains that have to deal with big data in motion. The following examples are use cases of stream processing technology in general and event processing in particular.

Fraud Detection. Credit card fraud is a big issue nowadays. The damage it causes has a volume of billions of U.S. dollars worldwide every year. For instance, the banking industry estimated that losses through credit card fraud had a volume of 535 million British pounds only in the United Kingdom in 2007 [BBC]. Luckily, fraudulent use of credit cards results in specific patterns appearing in the stream of transactions. Stream processing technology is perfectly suited for searching patterns in those streams in order to detect credit card fraud in real-time [SMP09]. This gives the opportunity to reduce or even avoid damage by canceling fraudulent transactions. Another area where stream processing technology has been successfully applied for real-time fraud detection is cellular telephony. Here, massive streams of call description records are analyzed in order to detect mobile phone fraud [Gul10].

Healthcare. Modern intensive care units monitor critically ill patients by permanently measuring a variety of vital signs such as heart rate, temperature or blood pressure via body sensors. Typically, the most recent information is visualized on screens and analyzed by healthcare professionals to detect changes in the condition of a patient as soon as possible. One medical professional is responsible for more than one patient in general. In times where medical staff is absent, intensive care units must analyze the sensor data by themselves and trigger an alarm when the condition of a patient becomes worse. For detecting serious situations in real-time, stream processing technology is optimal and has gained attention therefore [Blo10, HRP06, Sow10].

Internet of Things. Already today mobile devices outnumber people [Cis14]. And in the near future, the number of mobile devices will continue to grow enormously. Besides smartphones and tablet computers, also household appliances, cars, clothes, and public infrastructures are attached more and more with embedded computers, sensors and actors. Connected to the Internet and having unique IP addresses, all those objects can interact with each other, become remotely controllable, and provide enormous streams of information. This massive interconnection of everyday objects over the Internet is known as the Internet of Things (IoT) [AIM10, VF13]. Its main applications are automation and monitoring. Much work is currently done in the areas of home automation (smart home), car automation (smart vehicle), and resource management (smart grid and smart city) [Fre14, Mon13, Piy13]. Because there are huge streams of information to analyze and because actions must be taken in time, stream processing technology plays obviously a key role in the IoT.

IT Infrastructure Monitoring. Stream processing technology is ideal for permanently monitoring objects. This is, for example, extremely important in the case of IT infrastructures. Today's companies rely so much on their IT infrastructures that every single downtime and failure is very costly (up to millions of U.S. dollars per hour downtime) [Con04]. Longer downtimes can even ruin some kinds of business (e.g., banks or cloud service providers). Possible threats are diverse and come from two directions. First, hardware as well as software can fail. The holistic monitoring of both is crucial to achieve high availability and to early detect problems that can impact the performance [HFS12, Hoß11]. Second, IT infrastructures are confronted with attacks coming from insiders as well as outsiders. Successful attacks not only might decrease performance or make services unavailable, but also sensible and business-critical data might be leaked. Attacks must be detected as they happen and sensible data should be prevented from leaving the internal network. For instance, every network packet having a target destination outside the intranet can be first scanned for sensible information using stream processing technology [Kan08].

Logistics. Today's logistics is important for economies all over the world and quite complex through its global scale. A single product is typically the result of a collaboration of multiple partners organized in a supply chain [Chr11]. While some suppliers provide the most basic resources and raw materials, others produce intermediate and final products. At the end, distribution and sales partners take care of the delivery to end consumers. Supply chains must work optimally all time and require continuous monitoring, controlling and forecasting therefore. Many supply chains implement the just-in-time production strategy [BS11]. This means that goods are moved upstream to the next partner at the very moment they are needed. Just-in-time logistics leads to more flexible supply chains and reduces the need of stores which saves costs but makes the entire supply chain more vulnerable because there are no buffers to overcome temporary problems. Stream processing technology is well-suited for synchronizing entire supply chains accurately [TIB11].

Stock Market Analysis. The integration of stream processing technology into systems for stock market analysis and electronic trading is one of the most famous examples of its application [Agr08, Bar07, Dem07, DIG07, GÖ03, MM09, ZS02]. Stream processing technology is excellently suited, because it is not only capable of processing massive stock-ticker streams in real-time, but also able to detect complex patterns that are of high interest in technical analysis [Bal13, SÇZ05].

1.2 Data Stream Processing

Data streams can be defined and modeled in many different ways. But in general, a data stream is a potentially unbounded sequence of individual data items. In this thesis, we use the expression *data stream processing* (or simply *stream processing*) as a general term for any kind of tool, algorithm or system that consumes and processes data streams in an online manner. This means that new data items are directly processed without the need to store them first and that the result of the processing is continuously updated as new data items arrive.

Figure 1.1 shows the three essential components of the stream processing paradigm [Aba03, Bab02, Hoß13, HS13, SÇZ05]. First, there are data sources which provide streams of data items. Second, the data streams are consumed and processed by continuous queries (CQ). Third, the output streams of CQs are consumed by data sinks. Note that it is fundamentally possible that a CQ consumes multiple data streams or the output data streams of other CQs (both are depicted in the figure).

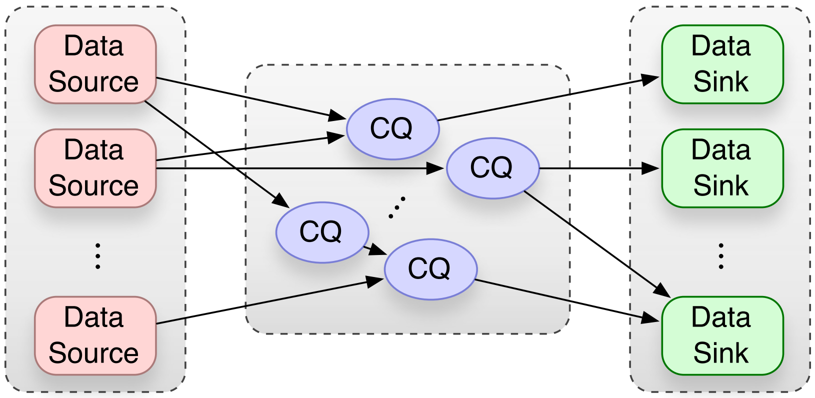


Figure 1.1: Data stream processing

In the above description, the data model of data streams and the processing performed by CQs remain abstract. Since there are various areas focusing on different kinds of data stream processing, our description is as abstract as possible in order to cover them all. We introduce some important kinds of data stream processing in the following. The database systems community has developed data stream management systems (DSMS) which are complementary to traditional database management systems (DBMS). In a DSMS, the CQs perform exactly the same type of processing that is done by ad-hoc queries in a DBMS efficiently over streams of tuples (in particular projection, filtering, aggregation, join and set operations such as union and difference). Many DSMSs evolved into general-purpose stream processing engines (SPE) whose functionality goes beyond the functionality of DSMSs (e.g., pattern matching over data streams is supported). For example, modern SPEs are Esper [Esp] (open source), Odysseus [App12] (academia) and webMethods Business Events [web] (commercial). DSMSs as well as SPEs provide powerful query languages, which are mostly based on SQL [SQL], to define CQs. Query definitions are translated into directed and acyclic operator graphs and compiled into final execution plans. The data mining community has modified their basic algorithms (e.g., for clustering [BH06]) to one-pass algorithms so that they can be applied in the form of CQs over data streams. Furthermore, the publish/subscribe community has developed brokers that are capable of matching streams of messages against a large number of subscriptions. As a final example, the IT security community has built intrusion detection systems (IDS) (e.g., Snort [Sno]) that search for signs of penetration in massive network streams.

1.3 Event Processing

The main area in which the research presented in this thesis was done is event processing (EP) that is another specific kind of data stream processing. Event processing focuses on the analysis of streams of events via basic operations. The notion of an event comprises any type of information from either real world (e.g., position update of a moving object, price update of a stock) or virtual world (e.g., notification about changed state of a software component). In addition, every event is associated with the point in time it happened. The analysis of event streams has a concrete objective in general, namely the detection of situations of interests (SoI) in real-time.

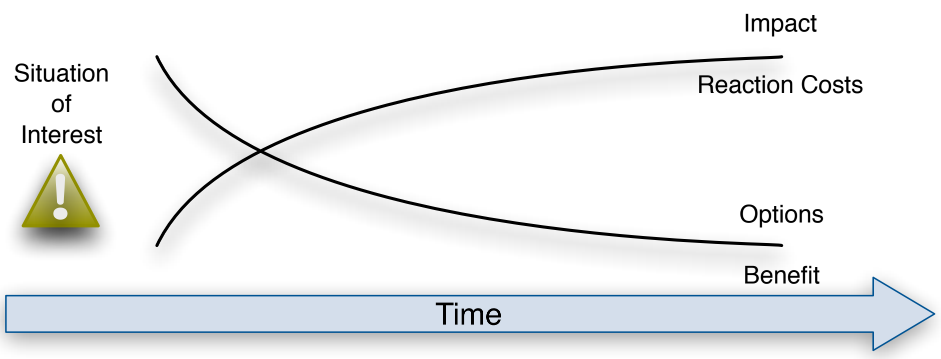


Figure 1.2: Situation of interest

Figure 1.2 motivates this aim of event processing. Single events as well as combinations of events indicating a problem or an opportunity (we summarize both under the term SoI) must be detected as early as possible in order to be able to take (counter-)measures in good time. Otherwise, the benefit out of opportunities and the total number of options decrease over time while the impact and the costs caused by problems increase over time. Event processing supports the detection of SoIs by providing basic types of CQs, called *event processing agents* (EPA), as atomic building blocks for creating detection rules [EB09, EN10]. The most important types of EPAs are as follows. The filter EPA is for selecting events having certain user-defined properties. A set of events can be summarized via the aggregation EPA using user-defined aggregate functions. The correlation EPA joins two or more event streams on basis of a user-defined correlation condition. Lastly, the pattern matching EPA detects user-defined patterns in streams of events. The power of event processing lies not primarily in the set of provided elementary EPAs, but in the possibility to arbitrarily compose EPAs to more complex CQs called *event processing networks* (EPN).

Occasionally and particularly in the context of pattern matching, EP is also called *complex event processing* (CEP). In this thesis, we prefer the term *event processing* since pattern matching is an important but only a small aspect of EP. The exact terminology throughout this thesis is as follows. In the presentations of work specifically for event processing we use the corresponding specialized terms (i.e., event, event stream, event source, EPA, EPN, event sink). But some of the presented work is more general and can be adopted by other kinds of data stream processing. Also when such work is presented specifically in the context of event processing, we use the general terms of data stream processing (i.e., data item, data stream, data source, CQ, data sink).

1.4 Remainder of the Part

The rest of the introductory part is structured as follows. Chapter 2 presents serious issues of state-of-the-art event processing technology. Chapter 3 gives a high-level overview of the contributions of this thesis. Chapter 4 outlines the rest of this thesis.

2

Problems and Emerging Challenges of Event Processing

Outline

2.1 Introduction

Although event processing has been in focus by academia and industry for over a decade now, it is still a young technology facing many problems. In addition, new and future applications such as, for example, IT security monitoring [Bau15] and the Internet of Things cause challenges that cannot be successfully mastered by state-of-the-art event processing technology.

2.2 Heterogeneity

In the absence of standards such as the famous query language SQL [SQL] of DBMSs and generally accepted principles, the area of event processing became extremely heterogenous in terms of data models, APIs, query languages, query semantics, expressiveness, and processing behaviors. This problem is well-known and has been studied intensively [Bot10, CM12, Din13, Etz10, Hoß13, Jai08, Tat10]. But until now there are no efforts to establish binding standards so that systems for event processing will strongly differ also in the future and heterogeneity will remain a serious problem in the event processing area. The consequences of high heterogeneity are manifold and critical. Applications implemented with one specific EP system cannot be moved to another EP system. Thus, applications are highly intermeshed with the used EP system and cannot be shared or sold. Moreover, if the used EP system becomes no longer suitable (e.g., the vendor goes out of business, an increased workload cannot be handled anymore, or there is simply a better or cheaper alternative system), the exchange for another EP system is nearly impossible, because it would require a costly rewrite of the entire application (vendor lock-in). The same is true for extensions built on top of a specific EP system. An extension to EP is platform-dependent and only compatible with the system it was implemented for. To support multiple different EP systems, an extension must be implemented for all EP systems separately. Again, heterogeneity causes high costs and prevents universal extensions. Lastly but not less importantly, different EP systems cannot cooperate in a federation. Let us take the supply chain example of the last chapter. In the case that the partners of a supply chain use different EP systems, it is quite hard to integrate the different systems in order to implement a holistic monitoring of the entire supply chain. Based on their inherent incompatibility, different EP systems also cannot work together in order to exploit synergies (from a functionality point of view as well as from a performance point of view).

2.3 Performance

Good performance in terms of high event throughput and low latency is an important issue in event processing. Many real-world event streams are already massive and expected to grow further in the future. As in any other area of data management, there are good chances and various possibilities to improve the performance by better algorithms or better implementations also in the event processing area. Furthermore, there are plenty of opportunities to optimize (continuous) queries that are specified with respect to a solid operator algebra. On basis of algebraic equivalences, operator plans can be transformed into equivalent but more efficient operator plans. This powerful technique of query optimization is an integral part of query compilers for DBMSs, but only rarely implemented by existing EP systems.

The rise of MapReduce [DG08, Läm07] and NoSQL systems [Gro13] in the last years clearly showed that challenging workloads can only be processed adequately by distributed infrastructures. This is because data volumes grew significantly faster than the performance of computing hardware. Of course, this development affects not only batch processing, but also stream processing. Analogues to the past development in the area of batch processing, distributed infrastructures will play an important role in the future of stream processing. Therefore, the ability to efficiently run distributed while still remaining easy to use will become crucial for EP systems.

2.4 Dynamic Applications

Independent of concrete EP systems, applications built on top are static during runtime. The entire workflow from the external event sources through an EP system up to final consumers is hard-wired. Changes of one element require manual adaption of subsequent elements and of the connections between them. For example, it is currently almost impossible to seamlessly integrate new types of event sources or to update an existing event type at runtime (e.g., a temperature sensor measuring the temperature in degrees Fahrenheit is exchanged for a new sensor that measures the temperature in degrees Celsius). The problem of inflexibility becomes really serious in cases where EP applications must connect to a constantly varying set of event sources and event sinks. In order to enable novel applications such as the Internet of Things, EP systems must be self-adaptive and need a more flexible way of connecting event sources to queries, queries to other queries, and queries to event sinks as it is currently provided by today's EP systems.

Another important aspect of state-of-the-art EP systems is that continuous queries are static during runtime. Once created and started, the definition of a CQ cannot be modified anymore. Updating a CQ requires to stop it, to redefine it and to deploy it again. Then, it must be waited until the updated CQ reaches a consistent state. This procedure is unacceptable in business-critical applications and not working in applications that need queries to be updated with high frequency. The inability to update continuous queries on-the-fly leads to serious problems in context-sensitive application domains and prevents anomaly detection via event processing. Context-sensitive domains require applications to be highly adaptive. Otherwise, SoIs depending on a specific context (e.g., time, location or the states of some objects) cannot be detected adequately. Today's static CQs are simply signatures specifying precisely the SoIs they are looking for. Signature-based detection is known for being expensive (each single SoI must be specified as a signature), potentially incomplete (there might exist unknown or unexpected SoI without an active signature) and hard to maintain. On the opposite side, perfect signatures have a 100 % detection rate and produce no false positives. A different way of detecting SoIs is searching for anomalies. Anomalies are significant differences from a normal state and behind each anomaly a problem or an opportunity could be concealed [CBK09]. Anomaly detection can produce false positives, but is inexpensive, complete (every anomaly is detected) and easy to maintain. Additionally and in comparison to the signature-based strategy, the detection of anomalies discovers unexpected or even unknown SoIs as a side effect.

2.5 Temporal and Spatiotemporal Applications

The temporal dimension is inherent in event processing and separates event processing from most other areas of data stream processing. In particular, every event is associated with a point in time and CQs can always take into consideration the temporal dimension. This allows, for example, to support sliding time windows that capture the most recent events within a fixed range of time. However, some of the basic EPAs still do not fully utilize the temporal dimension. Pattern matching is one of the most famous and important basic EPAs. But the underlying detection model of existing approaches is based on the sequential order of events and does not utilize the temporal dimension. In existing approaches, only three different relationships between two events can be expressed in total. One event can happen before, after or at the same time as another event. But the temporal information contained in every event allows for expressing more detailed and manifold relationships between two events.

Another problem is that all basic EPAs support neither spatial data types nor spatial operations, because an event is not associated with a location by default. Spatial information, if present, must be manually encoded using elementary data types only. Moreover, only simple spatial operations can be performed via provided functionality. This workaround is quite limited when it comes to representing not only simple geometries such as points, but also complex geometries such as lines and areas. Consequently, the integration of EP systems into spatial applications is nearly impossible. But there are numerous spatial applications which would benefit from event processing technology. For example, the city of Stockholm has installed numerous fixed point and moving sensors in order to monitor the city traffic [Kar12]. An automated traffic management system that utilizes the sensor data must react timely. Event processing technology would be ideal, but obviously there is a lot of spatial data involved.

3

Contributions of the Thesis

Outline

3.1 Introduction

This thesis presents the design and implementation of a novel middleware for event processing named *Java Event Processing Connectivity* (JEPC). Its aim is to overcome the problems of state-of-the-art event processing technology described in Chapter 2. Therefore, the contributions of this thesis coincide with improvements of existing features of EP achieved by JEPC and novel features of EP introduced by JEPC. In the following, we give a basic overview of the JEPC middleware.

3.2 Abstraction and Unification

JEPC provides an abstract view on EP. In particular, JEPC specifies a simple and clear but complete API and query language for EP. The API is needed to create and maintain EP applications. For instance, event streams must be registered and continuous queries must be created. A main feature of JEPC is the well-founded semantics of its queries that is based on a solid theoretical foundation. Continuous queries can be defined by composing EPAs to EPNs purely in Java or in a SQL-like query language. Via a REST [Fie00] interface, JEPC is also remotely available as a Web service.

The abstraction from specific EP providers allows JEPC to unify them with respect to APIs, query languages and behaviors. This is done by creating a tailor-made bridge for each EP provider being supported. The bridges are used to map the API, query language and behavior of JEPC to different EP providers. This concept enables not only dedicated EP systems to implement the JEPC specification, but also database systems and novel computing platforms such as GPUs and FPGAs. Besides JEPC bridges to different stream processing engines, we also developed a bridge to JDBC that enables every standard database system to support event processing and we offer a direct implementation of the JEPC specification in the form of a bridge. This native EP provider achieves high performance, because we can use very efficient algorithms and implementations tailored to the JEPC specification.

3.3 High-Performance Event Processing

Since performance is one of the most important non-functional requirements of event processing, it has been considered in several ways. JEPC provides efficient algorithms and implementations for most essential tasks, includes a query optimizer and can be easily distributed in many respects.

3.3.1 Efficient Operator Implementations

The native EP provider of JEPC contains user-defined implementations of the basic EPAs. For the filter and aggregation EPAs, we present novel algorithms and implementations that are superior to state-of-the-art algorithms and implementations. Filtering is a simple but very important operation in event processing. Most applications install a filter layer in front of the actual analysis of events. This is done for performance reasons and naturally given in most cases, because query definitions usually include filter conditions. Therefore, query indexes can be used to efficiently route incoming events directly to qualifying queries. We adopted the currently best performing query index and added several optimizations which further push the performance of EP applications. Aggregation is a basic operation in every kind of data analytics and event processing is no exception. With respect to the state-of-the-art algorithm for aggregation over event streams with time-interval semantics, we present a new algorithm that achieves a significantly better performance and scalability.

3.3.2 Novel Optimization Techniques

The foundation of JEPC queries is a temporal operator algebra. Therefore, we provide a query optimizer that transforms operator plans on basis of algebraic equivalences using rule-based and cost-based techniques. Besides the adoption of several efficient and widely known techniques from the area of database systems, our query optimizer also takes pattern matching EPAs into account. We introduce a novel transformation that allows both the optimization of a single pattern matching query as well as the optimization of an entire set of pattern matching queries.

3.3.3 Distributed Event Processing

Because challenging workloads require distributed event processing infrastructures, JEPC applications can be distributed in any granularity. In particular, multiple individual JEPC instances can be distributed, entire EPNs can be distributed and also single EPAs can be distributed individually. Since JEPC unifies different types of EP providers under a common specification, a distributed JEPC infrastructure is allowed to consist of different types of EP providers without any restrictions. We call such a heterogenous infrastructure a *federation*. In a federation, we can take advantage of the heterogeneity of EP, because different EP providers execute a concrete query with different performances in general. EPNs and EPAs can be distributed across a federation so that they are executed by the best suited type of EP provider. JEPC includes a novel optimizer that distributes queries across a federation of different EP providers with respect to not only load balancing, but also performance optimization. We show that federated event processing infrastructures perform significantly better than the pure parallelization of a single type of EP provider only.

3.4 Dynamic Event Processing

In order to address the special requirements of dynamic and context-sensitive applications, we propose a novel paradigm named *dynamic event processing* (DEP). Dynamic event processing consists of two different components. Note that each of them can also be used alone. First, continuous queries are made efficiently but safely updatable on-the-fly in order to be able to react to changes in the context of an application. Second, static and user-defined connections between the elements of a stream processing application prevent them from being self-adaptive and are replaced by dynamic and fine-grained connections which are automatically managed therefore.

The lack of update functionality for continuous queries makes the holistic management of their life cycles impossible. We introduce an update method for continuous queries that quickly enables an updated query definition by loading a new continuous query having the updated definition with historical data items first and then safely switches from the outdated continuous query to the new one. Because our method depends on efficient recording and reloading of potentially massive data streams, we also present the design and implementation of a high-performance stream store.

Regarding the problem of static and user-defined connections, we propose a hybrid approach that combines the high performance of fixed connections with a new type of flexible publish/subscribe. In particular, we introduce the concept of automatic matchmaking. Because of the high performance requirements, automatic matchmaking still establishes fixed connections between the elements of a stream processing application. But in contrast to state-of-the-art stream processing technology, all connections are fine-grained and established automatically as well as dynamically whenever a data producer or data consumer is deployed or updated. This not only reduces the effort of building and maintaining complex applications, but also enables the use of stream processing technology in highly dynamic applications such as the ones from the Internet of things. Furthermore, our proposed concept of matchmaking is able to automatically handle semantic differences among the connected data producers and data consumers in a declarative manner.

3.5 Temporal and Spatiotemporal Pattern Matching

Because the sequence-based processing model of today's pattern matching approaches does not fully utilize the temporal nature of event processing, we present a novel pattern matching EPA named *TPStream* for temporal pattern matching. TPStream is for the detection of complex temporal patterns in event streams, but still able to detect traditional sequential patterns. Therefore, it is an evolution of pattern matching rather than a competing approach.

On the basis of TPStream, we present a general, simple and powerful way of extending EPAs by spatial functionality. In the special case of TPStream, this enables it to process spatial data in order to detect spatiotemporal event patterns. The presented procedure for extending an EPA by spatial functionality is not limited to TPStream and can be easily applied to any other type of EPA making an entire EP system capable of interpreting and processing arbitrarily complex spatial data. Therefore, JEPC provides a version of every basic EPA with full support of spatial data.

4

Outline of the Thesis

Outline

4.1 Introduction

From an architectural point of view, the JEPC middleware consists of two main components. The first component contains the JEPC core and JEPC bridges. Both are necessary in every JEPC application. JEPC extensions are optional and belong to the second main component. Consequently, the presentation of JEPC is divided into two main parts one describing the JEPC core and JEPC bridges (Part II), the other describing the JEPC extensions (Part III). Besides the main parts, this thesis starts with an introductory part (Part I) motivating the JEPC middleware and ends with Part IV that summarizes the main findings and suggests future research directions.

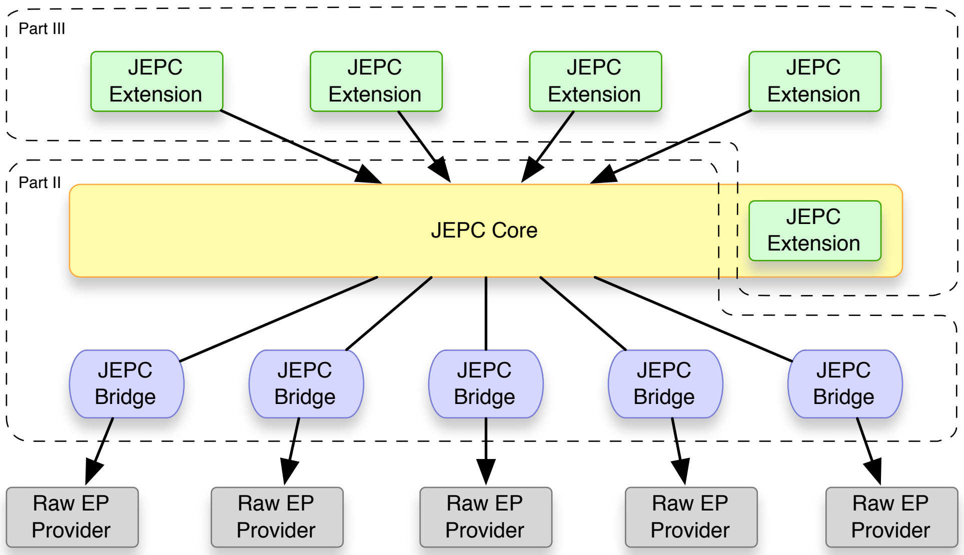


Figure 4.1: Architecture of JEPC and thesis outline

4.2 JEPC Core and JEPC Bridges

Part II presents the core of JEPC which consists of the interfaces defining the API and the specification of the query language. Also some of the available JEPC bridges to different raw EP providers are presented. The combination of a raw EP provider and its corresponding bridge constitute an implementation of the JEPC specification. Figure 4.1 shows the set of all main elements of the core and their interaction. On top of the JEPC core, end-users and developers can access a common event processing API and query language that are independent of the raw EP provider beneath.

4.3 JEPC Extensions

The core of JEPC offers a holistic API and query language for event processing and can be used like any other EP provider to implement extensions to event processing technology. But the advantage of JEPC in contrast to raw EP providers is that extensions on top benefit from JEPC in exactly the same way as applications on top. In particular, only one implementation of an extension is necessary to become compatible with all by JEPC supported raw EP providers. Figure 4.1 illustrates the positions of JEPC extensions in the overall architecture of JEPC. As in the case of applications, most extensions are purely on top of the core of JEPC. In certain cases it might be convenient or necessary that parts of an extension are integrated into the JEPC core or into the JEPC bridges (e.g., for performance reasons). This fact is also indicated in the figure. Part III presents multiple JEPC extensions we developed with regard to the remaining problems and challenges of EP technology listed in Chapter 2.

Part II

Core of the Middleware

5 Introduction

Outline

5.1 An Abstraction Layer for Event Processing

This part presents the core of JEPC as well as multiple JEPC bridges to different SPEs, standard database systems and native EPA implementations. The core of JEPC is a virtual EP provider for EP applications from the user's perspective. Similar to virtual machines (e.g., CLR [Ric12], JVM [Lin14], LLVM [The14], Smalltalk VM [GR83]) that make applications independent of the underlying platform (e.g., operating system, computer architecture), the core of JEPC makes EP applications independent of the underlying raw EP provider. The advantages of such an abstraction layer for event processing are manifold. First, EP applications can be moved and shared between different raw EP providers. Second, entire EP applications and collections of queries (e.g., detection rules for specific attacks on computer systems) can be offered and sold without being dependent on a concrete raw EP provider. Thus, an abstraction layer overcomes the problem of vendor lock-in and helps to enlarge both user base and acceptance of event processing technology. Third, raw EP providers that have different APIs, behaviors and query languages can cooperate in a federation. This is because through the implementation of the abstraction layer raw EP providers are unified. Fourth, extensions to event processing technology that are developed for a virtual EP provider achieve exactly the same advantages as EP applications.

In the area of standard database systems, the success stories of ODBC [Gei95, Mic15] and JDBC [And11, GUW08] clearly prove the importance and benefits of abstraction layers in data management. Even though there are a lot of parallels between ODBC/JDBC and JEPC, the problems to solve are harder and more numerous in the case of event processing than in the case of standard database systems. This is because standard database systems have a standard, namely SQL [SQL]. Among others, SQL clearly specifies a database query language in terms of syntax and, more importantly, semantics. This specification is respected by all standard database systems so that queries can be directly forwarded to database systems. In contrast, raw EP providers differ dramatically in not only APIs, but also query languages. As a consequence, an abstraction layer for event processing must specify a common query language besides a common API. While it is relatively easy to unify the APIs of data management systems of the same type, the unification of different query languages is a challenging problem. Therefore, an abstraction layer for event processing must incorporate substantially more aspects than its counterparts for database systems.

5.2 Design Goals and Principles

JEPC as a whole and particularly the core of JEPC strictly follow a minimalistic design principle. The core of JEPC specifies and provides only functionality that is essential for EP and that cannot be (efficiently) provided by composing already existing functionality. This is mostly due to the process of unification. An abstraction layer for different and heterogenous raw EP providers can only support functionality that is explicitly or implicitly provided by all of them. Special functionality that is only supported by few raw EP providers is not part of the JEPC specification therefore. Furthermore, the more functionality part of the JEPC specification is the more likely additional raw EP providers cannot be supported at all or only at high cost. Lastly, a minimalistic JEPC core is maintainable, easy to learn and simple to use. Nevertheless, we want the JEPC specification to be based on a sound theoretical foundation and to comprise all essential functionality for EP so that it can substitute raw EP providers in almost all applications including business-critical applications.

5.3 Remainder of the Part

The rest of this part is structured as follows. Chapter 6 specifies the data model and the semantics of JEPC queries. Chapter 7 presents the API and the query language of the JEPC core. Chapter 8 gives insights into the development of available JEPC bridges to different SPEs. Chapter 9 describes a special JEPC bridge to JDBC that allows the connectivity to all standard database systems. Another special JEPC bridge to native EPA implementations is illustrated in Chapter 10. Its implementations of the filter and aggregation EPAs are presented in Chapter 11 and Chapter 12 respectively. Chapter 13 concludes this part.

6

Theoretical Foundations

Outline

6.1 Introduction

In this chapter, we specify the semantics of JEPC queries in the form of a well-founded temporal operator algebra which allows declarative query languages and graphical query composers to be built on top. Our event processing algebra is mainly based on the snapshot-reducible data stream algebra [KS09]. Snapshot-reducible means that the algebra interprets a data stream as a sequence of snapshots and defines operators so that they produce exactly the same output for each snapshot as its counterparts in the extended relational algebra [Cod70, DGK82]. This approach not only carries over the widely known and accepted relational algebra to event streams, but also makes JEPC queries compatible with standard databases. For example, recorded event streams can be queried via standard databases to reproduce the output of JEPC queries. However, since pattern matching is not part of the relational algebra and operates on a sequence of events rather than on a sequence of snapshots, we have to define it in addition and must seamlessly integrate it into our event processing algebra.

6.2 Event Streams

Sequences are a natural data model to represent event streams [LWZ11] because they occur in that form in practice. Then, an event stream E is a potentially infinite sequence of events. Each event is a pair (p, t) consisting of a payload p and a timestamp t. All payloads of an event stream are from the same domain. Timestamps are from a discrete and totally ordered time domain. The meaning of an event is that p happened at the instant of time t. All events of an event stream are ordered by their timestamps and for events with identical timestamps there is no order defined.

Besides the representation of event streams as sequences, the snapshot-reducible data stream algebra introduces another representation on the basis of multisets. This representation can be derived from every event stream E by counting all identical

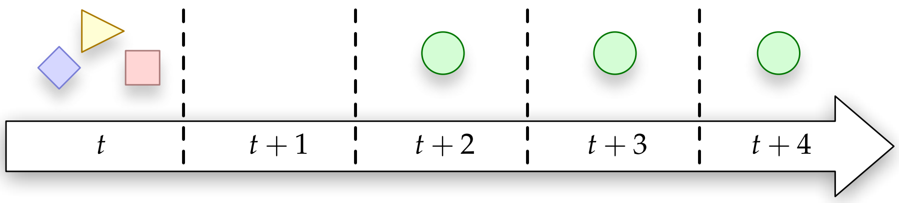


Figure 6.1: Event stream as a sequence of snapshots

events (p,t) and aggregating them into a single logical event (p,t,n). The additional number n denotes the total count of all identical events (p,t). For an event stream E, the corresponding logical event stream E^L can be produced via the function η^L:

$$E^L = \eta^L(E) := \{(p,t,n) \mid n = |(p,t) \in E| \wedge n > 0\}$$

A logical event stream can be best interpreted as a temporally ordered sequence of snapshots. For each point in time there is exactly one multiset containing all valid events. Because there is no order defined on the elements of a set, there is also no order within single instants of time. Therefore, every snapshot is identical to a traditional non-temporal database table. The snapshot of a logical event stream E^L for a point in time t can be obtained via the function τ:

$$\tau_t(E^L) := \left\{(p,n) \mid (p,t',n) \in E^L \wedge t' = t\right\}$$

Figure 6.1 illustrates the interpretation of an event stream as a sequence of snapshots. It shows five snapshots in total along the timeline. The first snapshot at time instant t contains three different events (*diamond*, *triangle*, and *square*) each appearing exactly one time. Snapshots such as the second at time instant $t+1$ may also be empty. Furthermore, it might be possible that an event is contained in multiple successive snapshots. For instance, the event *circle* is valid at the instants of time $t+2$, $t+3$ and $t+4$. In this case, its valid time can also be encoded as the closed time interval $[t+2 : t+4]$ or as the half-open time interval $[t+2 : t+5)$ respectively. We use time intervals in our implementation, because it is a more compact representation of long-living events and allows for more efficient processing (see Chapter 7 for details).

Note that simultaneous events are allowed to be in any order without affecting the output of operators. According to Lamport's happend-before relation [Lam78], simultaneous events have no total temporal order because none of them happened before the others. While every event stream E can be easily transformed into its logical representation E^L via η^L, the inverse transformation η can only be clearly defined for the specific case where the logical event stream has no snapshots that contain more than one event (i.e., the event stream has no simultaneous events at all). But because the snapshot-reducible algebra is defined on logical event streams and some operators such as the pattern matching operator require strict event sequences, we need an inverse transformation η for logical event streams that is total:

$$\begin{tikzcd} < (p,t),(p',t'),\ldots > = E \arrow[r, bend left, "\eta^L"] & E^L = \{(p_i,t_i,n_i)\}_{i=1,2,3,\ldots} \arrow[l, bend left, "\eta"] \end{tikzcd}$$

t	E^L	T_1	T_2
101	(A,101,1)	(A,101,1)	(A,101,1)
102	(B,102,1), (C,102,1)	(B,102,1)	(C,102,1)
103			
104	(D,104,1)	(D,104,1)	(D,104,1)

Table 6.1: Universes of a logical event stream

To solve the problem, we define that operators requiring event sequences handle simultaneous events as alternatives. Then, for each point in time there can exist at most one event in a sequence. Whenever two events happened at the same time instant t, then there are two different interpretations called *universes* in the following. One of the events happened in some universe T_1 at time instant t and the other event happened in a different universe T_2 at time instant t. This approach has clear semantics and is implemented by some existing systems [Dem07, web]. Table 6.1 explains the concept of universes by an example. The logical event stream shown in the second column has simultaneous events with payloads B and C at the point in time 102. Two different event sequences T_1 and T_2 that are both totally ordered can be derived from the logical event stream. They are shown in the righthand columns and differ at time instant 102. One event sequence contains the event with payload B and the other contains the event with payload C. For a logical event stream E^L the corresponding multiset containing all universes T of E^L can be created via the function η^U:

$$\begin{aligned} E^U = \eta^U(E^L) := \Big\{ (T, n) \;\Big|\; & \exists X \subseteq E^L : \; (\\ & X = \{(p_1, t_1, n_1), (p_2, t_2, n_2), \dots, (p_k, t_k, n_k)\} \\ & \wedge \forall i \in [1:k) : \big(t_i < t_{i+1} \wedge \forall t' \in (t_i : t_{i+1}) : \nexists (p', t', n') \in E^L\big) \\ & \wedge \forall t' < t_1 : \nexists (p', t', n') \in E^L \wedge \forall t' > t_k : \nexists (p', t', n') \in E^L \;) \\ & \wedge T = \{(p, t, 1) | (p, t, \hat{n}) \in X\} \; \wedge \; n = \prod_{(p,t,\hat{n}) \in X} \hat{n} \Big\} \end{aligned}$$

A universe T in $\eta^U(E^L)$ is a totally ordered sequence and consists only of events that are also contained in E^L. Furthermore, all events in T have unique timestamps and T has no holes. A hole exists if and only if there is at least one event (p', t', n') in E^L but no event in T at the instant of time t'. Because a universe T can be derivable multiple times from E^L, the number n denotes the multiplicity of T. Note that the index k is only used to indicate the last timestamp in E^L. This index can be arbitrarily large in case of an unbounded event stream.

6.3 Event Processing Algebra

This section presents the operator algebra that specifies the semantics of JEPC queries. Each basic type of EPA is represented by its own operator. Because all basic types of EPAs are part of an algebra, they can be composed arbitrarily to EPNs.

6.3.1 Parameters

Every basic type of EPA is customizable by users via parameters. Therefore, there are also specifications for Boolean expressions (BE), aggregates and event patterns needed. These parameters are introduced first.

6.3.1.1 Boolean Expressions

The algebra $\mathcal{BE} = (\mathbb{C}, \mathbb{V}, \mathbb{P}, \mathbb{O})$ describes all Boolean expressions. Each component of the algebra is explained in Table 6.2. Of course, BEs can consist of constant values such as numbers and strings. Besides constants, there are variables that have no fixed values. The values of variables must be set before evaluating a BE. Predicates are Boolean-valued functions that can be applied to constants and variables. To create more complex expressions, BEs can be connected with the help of operators. Additionally, we define the set of all terms $\mathbb{T}$ as $\mathbb{T} := \mathbb{C} \cup \mathbb{V}$. The syntax of a Boolean expression $\varphi \in \mathcal{BE}$ is defined inductively as follows:

- $\varphi = p(a,b)$, with $p \in \mathbb{P},\ (a,b) \in \mathbb{T} \times \mathbb{T}$
- $\varphi = \psi \circ \chi$, with $\circ \in \mathbb{O},\ (\psi,\chi) \in \mathcal{BE} \times \mathcal{BE}$

Each binary predicate on two terms is a syntactically correct BE and the application of the operators to BEs results in a new BE. To define the semantics of BEs, we need the notion of an interpretation. An interpretation f exchanges each variable for a constant value and maps constant values to themselves so that each predicate on terms can be

Component	*Symbol*	*Examples / Definition*
Constants	$\mathbb{C}$	"ABC", 1000, $1/4 * 42, \ldots$
Variables	$\mathbb{V}$	X, Y, Card_No, Amount, …
Predicates	$\mathbb{P}$	$\mathbb{P} := \{<, \leq, =, \neq, \geq, >\}$
Operators	$\mathbb{O}$	$\mathbb{O} := \{\wedge, \vee\}$

Table 6.2: Components of Boolean expressions

evaluated to either `TRUE` or `FALSE`. Because BEs are always evaluated in the context of an event (p, t), we use the notion $f(p, t)$ later in this chapter. An interpretation $f(p, t)$ exchanges each variable (that must be identical to the name of an event attribute) for the value of the associated event attribute of a given event. Now, the semantics of a Boolean expression $\varphi \in \mathcal{BE}$ can be defined with respect to the syntax:

- $\varphi_f = p(x, y) :\Leftrightarrow p(x_f, y_f)$
- $\varphi_f = \psi \wedge \chi :\Leftrightarrow \psi_f$ and χ_f
- $\varphi_f = \psi \vee \chi :\Leftrightarrow \psi_f$ or χ_f

6.3.1.2 Aggregates

Aggregates are functions that reduce a set of events with payload from the same domain to exactly one new event whose payload can be from another domain. All aggregates are summarized in the set $\mathbb{A}$:

$$\mathbb{A} := \{max, min, avg, stddev, count, sum\}$$

These standard aggregates (maximum, minimum, average, standard derivation, count and sum) are supported by JEPC.

6.3.1.3 Patterns

A pattern is a regular expression over symbols that specifies a (potentially infinite) set of totally ordered symbol sequences. Symbols are emitted by events on the basis of user-defined conditions. Each symbol definition consists of a condition in the form of a Boolean expression $\varphi \in \mathcal{BE}$ and an optional binding of global variables:

$$Symbol \leftarrow s_{var_1 \leftarrow X, var_2 \leftarrow Y, \ldots}[\varphi_{f(p,t,var_1,var_2,\ldots)}]$$

The optional binding of global variables $(var_1, var_2, \ldots)$ is used to hold globally attribute values $(X, Y, \ldots)$ of the event that emits the corresponding symbol. Within a symbol condition φ variables can be attributes of the payload, the timestamp and all initialized global variables. Whenever a symbol is reused later in a pattern, its condition and variable bindings must not be repeated. The following grammar describes all legal patterns P over symbols:

$$P \leftarrow \emptyset \mid Symbol \mid PP \mid (Symbol|Symbol) \mid Symbol^* \mid Symbol^+ \mid Symbol\{n \in \mathbb{N}\}$$

The empty pattern as well as a single symbol are legal patterns. Patterns are allowed to be concatenated in order to create sequences. Two symbols can be alternatives. This means that only one of the symbols must occur. Finally, the Kleene star, the Kleene plus, and the exact count operators can be applied to a symbol. The Kleene star operator means that a symbol is allowed to occur any number of times in a row. The Kleene plus operator is equal but forces the symbol to occur at least once. The exact count operator requires that the symbol occurs exactly n times in a row.

For example, a simple fraud detection on a stream of credit card transaction events can be implemented by searching for the pattern P_{fraud}. Each credit card transaction event consists of the credit card number (`card_no`) and the amount (`amount`):

$$\begin{aligned} P_{fraud} := a_{card \leftarrow \texttt{card\_no}}[\texttt{amount} > 1{,}000] \\ b[\texttt{card\_no} \neq card]^* c[\texttt{card\_no} = card \wedge \texttt{amount} > 1{,}000] b^* c b^* c \\ b^* d[\texttt{card\_no} = card \wedge \texttt{amount} > 1{,}000] \end{aligned}$$

The pattern P_{fraud} defines all event sequences in which the same credit card was used five times in a row with an amount greater than 1,000 each time. In a fraudulent sequence, the first transaction event emits the symbol a. At this point, the credit card number is assigned to the global variable $card$. After the symbol a, the pattern b^*c must occur exactly three times. Symbol c in the pattern represents the next fraudulent use of the credit card (i.e., the same credit card was used with an amount greater than 1,000 again) and any number of transaction events for other credit cards are allowed to be mixed in (symbol b with the Kleene star). The pattern ends with the fifth fraudulent transaction event (symbol d). Before the last fraudulent transaction event occurs, any number of transaction events for other credit cards are again allowed to be mixed in by reusing the symbol b with the Kleene star.

t	E^L	$\omega_3^{time}(E^L)$
101		
102	(A,102,1)	(A,102,1)
103		(A,103,1)
104	(B,104,7)	(A,104,1), (B,104,7)
105		(B,105,7)
106		(B,106,7)
107		

Table 6.3: Time window example

6.3.2 Windows

Windows are required for increasing the valid time of events in order to control the states of stateful operators [Krä07, PS10]. In particular, they allow to express continuous queries that take into account all events within a user-defined range of time or a user-defined number of the most recent events.

6.3.2.1 Time Window

The time window operator ω^{time} extends the valid time of each logical event (p, t, n) of its input event stream by any range of time. A user-defined parameter w controls the size of the extension. At each instant of time in the half-open interval $[t : t + w)$ the input event occurs additionally n times in the output stream:

$$\omega_w^{time}(E^L) := \left\{ (p, t', n') \,\middle|\, \exists X \subseteq E^L : X = \{(p, t, n) \in E^L \mid \right.$$
$$\left. \max\{t' - w + 1,\, 0\} \le t \le t'\} \wedge X \neq \emptyset \wedge n' = \sum_{(p,t,n)\in X} n \right\}$$

For example, to continuously evaluate an aggregate over the x most recent time instants of an event stream, a time window of size x must be applied before the aggregation operator. Table 6.3 shows a time window in action. The event stream given in the second column consists of the event A at the point in time 102 and seven times of the event B at the point in time 104. In the third column, the output of a time window operator with $w = 3$ is shown. Each event of the input event stream occurs in the output not only at the original point in time, but also at the two subsequent ones.

6.3.2.2 Count Window

The count window operator ω^{count} extends the valid time of each incoming logical event (p, t, n) such that exactly the N most recent events are valid at every point in time. Therefore, the size of the extension is not fixed as in the case of time windows:

$$\omega_N^{count}(E^L) := \left\{ (p, t', n') \mid \exists X \subseteq E^L : X = \{(p, t, n) \in E^L \mid \right.$$
$$\max(\gamma(t', E^L) - N + 1, 1) \le \gamma(t, E^L) \le \gamma(t', E^L)\} \;\wedge\; X \neq \emptyset$$
$$\left. \wedge\, n' = \sum_{(p,t,n)\in X} n \right\}, \text{with } \gamma(v, E^L) := \left| \left\{ (p, u, n) \in E^L | u \le v \right\} \right|$$

t	E^L	$\omega_2^{count}(E^L)$
101		
102	(A,102,1)	(A,102,1)
103	(B,103,1)	(A,103,1), (B,103,1)
104	(C,104,1)	(B,104,1), (C,104,1)
105		(B,105,1), (C,105,1)
106	(U,106,1)	(C,106,1), (U,106,1)
107	(V,107,1)	(U,107,1), (V,107,1)

Table 6.4: Count window example

Supposing that an aggregate shall be continuously evaluated not over the x most recent time instants but over the x most recent events of an event stream, a count window with $N = x$ instead of a time window must be applied before the aggregation operator. Table 6.4 shows the output of a count window with $N = 2$ in the third column for an input event stream given in the second column. At every point in time, exactly the two most recent events of the input stream occur in the output stream.

6.3.2.3 Partitioned Window

The partitioned window operator $\omega^{partition}$ first partitions an event stream via a function β and then applies a time or count window to each partition separately. Its output stream is the union of the output streams of all separately applied windows. The function π is for obtaining all events belonging to the same partition:

$$\omega_{\beta,N}^{partition}(E^L) := \bigcup_{i=1}^{k} \omega_N^{time/count}(\pi_i(\beta(E^L)))$$

A partitioned window only makes sense in combination with count windows. In combination with time windows a partitioned window has no effect [Krä07].

6.3.2.4 Reduction

The reduction operator ω^{-1} reduces an event with arbitrary valid time to its very first occurrence. To be more precise, the first occurrence of every individual event in a logical event stream is preserved while all subsequent occurrences of that event are removed. In case the reduction operator is applied directly after a window operator, the effect of the window is eliminated. The following equations hold in particular:

$$\omega^{-1}(\omega^{time}(E^L)) = \omega^{-1}(\omega^{count}(E^L)) = \omega^{-1}(\omega^{partition}(E^L)) = E^L$$

t	E^L	$\omega_3^{time}(E^L)$
101	(A,101,1)	(A,101,1)
102		(A,102,1)
103		(A,103,1)
104	(A,104,1)	(A,104,1)
105		(A,105,1)
106		(A,106,1)

Table 6.5: Event occlusion effect

Note that ω^{-1} can be applied to every logical event stream and not only to the output streams of window operators. In our implementation that encodes the valid time of an event as a time interval, ω^{-1} simply shrinks the time intervals of incoming events so that every time interval covers only its first time instant afterwards. Because a logical event stream does not allow to recover individual events due to the event occlusion effect, either the data model for logical event streams must be extended by provenance or ω^{-1} cannot be defined formally. We decided to leave ω^{-1} abstract, because it is intuitively understandable and implementable without the need for provenance. Thus, the extension of the logical data model by provenance would be an overkill.

Table 6.5 demonstrates the event occlusion effect. The second column shows two individual events having the same payload. In the third column, the resulting event stream after the application of a time window of size 3 is shown. Without exact knowledge about the applied window, it cannot be decided how many individual events there are (the possibilities range from a single individual event up to six individual events). In our implementation that is based on the time interval approach, individual events are preserved so that ω^{-1} is applicable to every event stream.

6.3.3 Event Processing Agents

In the following, we define the elementary EPAs filter, correlator, aggregator and pattern matcher as operators of the algebra. All these basic types of EPAs take logical event streams as inputs and create one logical event stream as output. The EPAs filter, correlator and aggregator are specified to be snapshot-reducible to their non-temporal counterparts in the extended relational algebra. For the EPA pattern matcher, the incoming logical event stream must be transformed into a sequence of events.

t	E^L	$\sigma_{p>40 \wedge p<45}(E^L)$
101	(40,101,1), (41,101,1)	(41,101,1)
102	(42,102,1), (43,102,1)	(42,102,1), (43,102,1)
103	(44,103,1), (45,103,1)	(44,103,1)

Table 6.6: Filter example

6.3.3.1 Filter

The filter σ forwards an incoming event if and only if it fulfills a Boolean expression $\varphi \in \mathcal{BE}$. Attributes of the payload and the timestamp can be used as variables in φ:

$$\sigma_\varphi(E^L) := \left\{(p,t,n) \mid (p,t,n) \in E^L \ \wedge \ \varphi_{f(p,t)}\right\}$$

For example, the filter $\sigma_{p>40 \wedge p<45}(E^L)$ selects all events of an event stream with integer numbers as payload whose payloads are in $(40 : 45)$. Table 6.6 shows the output of this filter in the third column for an input event stream given in the second column.

6.3.3.2 Correlator

The correlator $\bowtie$ forwards a joined event of the Cartesian product of two arbitrary event streams E_1^L and E_2^L (i.e., the payloads of a pair of events are combined and their counters are multiplied) if and only if it fulfills a Boolean expression $\varphi \in \mathcal{BE}$ and the original events have identical timestamps. All attributes and the timestamp of the joined event can be used as variables in the Boolean expression φ:

$$\begin{aligned}\bowtie_\varphi (E_1^L, E_2^L) := \{ \ (p_1 \circ p_2, t, n_1 * n_2) \mid \\ \big((p_1,t,n_1),(p_2,t,n_2)\big) \in E_1^L \times E_2^L \ \wedge \ \varphi_{f(p_1 \circ p_2, t)}\}\end{aligned}$$

As an example, consider the logical event streams E_1^L and E_2^L with integer numbers as payloads shown in Table 6.7. The third column gives the output event stream of the correlator $\bowtie_{E_1^L.p > E_2^L.p} (\omega_2^{time}(E_1^L), \omega_3^{time}(E_2^L))$. An event of E_1^L is joined with an event of E_2^L if both events have identical timestamps and the corresponding event of E_1^L has an integer number as payload that is greater than the integer number of the payload of the event belonging to E_2^L. Before the input event streams enter the correlator, time windows of sizes two and three are applied to E_1^L and E_2^L respectively.

t	E_1^L	E_2^L	$\bowtie_{E_1^L.p>E_2^L.p}(\omega_2^{time}(E_1^L), \omega_3^{time}(E_2^L))$
101			
102	(39,102,1)	(40,102,2)	
103	(42,103,8)	(44,103,7)	(42∘40,103,16)
104			(42∘40,104,16)
105			

Table 6.7: Correlator example

t	E^L	$\alpha_{avg(p)}(\omega_2^{time}(E^L))$
101	(40,101,1), (41,101,1)	(40.5,101,1)
102	(42,102,1), (43,102,1)	(41.5,102,1)
103	(44,103,1), (45,103,1)	(43.5,103,1)

Table 6.8: Aggregator example

6.3.3.3 Aggregator

The aggregator α evaluates an aggregate $agg \in \mathbb{A}$ on each snapshot of a logical event stream that is not empty:

$$\alpha_{agg}(E^L) := \Big\{(p', t, 1) \mid \tau_t(E^L) \neq \emptyset \wedge p' = agg\big(\tau_t(E^L)\big)\Big\}$$

In addition, an aggregate agg can be evaluated separately for each single group of events with identical values of the attributes $A_1, A_2, \ldots, A_n$:

$$\alpha_{A_1,A_2,\ldots,A_n,agg}(E^L) := \bigcup_{e \in E^L} \alpha_{agg}\big(\sigma_{A_1=e.A_1 \wedge A_2=e.A_2 \wedge \ldots \wedge A_n=e.A_n}(E^L)\big)$$

For example, the aggregator $\alpha_{avg(p)}(\omega_2^{time}(E^L))$ computes the average value within a time window of size 2 over an event stream with integer numbers as payload. This aggregator applied to an example event stream is shown in Table 6.8. The first snapshot at $t = 101$ consists of two events having the payloads 40 and 41 so that the average value is 40.5. Because of the time window, the two succeeding snapshots consist of four events each. At $t = 102$ the events having the payloads 40, 41, 42, 43 are valid and at $t = 103$ the events having the payloads 42, 43, 44, 45 are valid. Therefore, the average values are 41.5 at $t = 102$ and 43.5 at $t = 103$.

6.3.3.4 Pattern Matcher

The pattern matcher ρ detects sequential event patterns in event streams. To find a pattern P within a time window of size w over a logical event stream E^L, all universes contained in the window are derived first. Then, for each universe T its corresponding symbol stream is created. For each event $(p, t, 1)$ in a universe T, all emitted symbols $s_1, \ldots, s_k$ according to the symbol definitions in P form its symbol stream. The corresponding symbol stream of a universe T is created via the transformation η^S:

$$\eta^S((T, n)) := \{\ (s, t, 1) \mid (p, t, 1) \in T\ \wedge\ p \text{ emits } s\ \}$$

A symbol stream contains as many symbol events for an event in T as symbols are emitted by that event. If an event in T emits no symbols at all, then the symbol stream contains the empty symbol $\square$ to signal that there is an event. A symbol stream is again a logical event stream. It has multiple symbol events at a point in time in general. Therefore, universes contained in a symbol stream must be derived in order to obtain final symbol sequences. Then, pattern matching can be performed on each of those symbol sequence separately. The final output of ρ is the multiset union of the outputs of all applied pattern matching instances:

$$\rho_{P,w}(E^L) := \big\{\ (m, t, \hat{n} * n_1) \ \big|$$

$$\exists X \subseteq E^L : X = \{\ (p, t', n) \in \omega^{-1}(E^L)\ \mid \max\{t - w + 1,\ 0\} \leq t' \leq t\}$$

$$\wedge\ \exists Y : Y = \bigcup_{(T_1, n_1) \in \eta^U(X)} \Bigg(\bigcup_{(T_2, n_2) \in \eta^U(\eta^S((T_1, n_1)))} matches_P((T_2, n_2)) \Bigg) \wedge (m, t, \hat{n}) \in Y\}$$

The function *matches* produces exactly one output event for each successful match. A successful match requires a finite symbol sequence S that matches the pattern P. The timestamp of the output event is set to the timestamp of the last symbol event in S, because the pattern occurred completely at this instant of time:

$$matches_P((< (s_1, t_1, 1), (s_2, t_2, 1), \ldots, (s_x, t_x, 1) >, n)) :=$$

$$\big\{ (m, t_x, n) \ \big|\ \exists i \in \mathbb{N} : \big(i < x\ \wedge\ \kappa_P(< (s_i, t_i, 1), \ldots, (s_x, t_x, 1) >) = m \wedge m \neq \emptyset\ \big) \big\}$$

The payload m of each output event is created by a second parameter *return*. This parameter defines all attributes of m and specifies how their values are set. The value of an attribute in m can be set to either a global variable of P or to any constant value. For a symbol event sequence, the function κ returns the final payload of the output event in case of a successful match:

$$\kappa_P(S) := \begin{cases} return & \text{if } S \text{ matches } P \\ \emptyset & \text{otherwise} \end{cases}$$

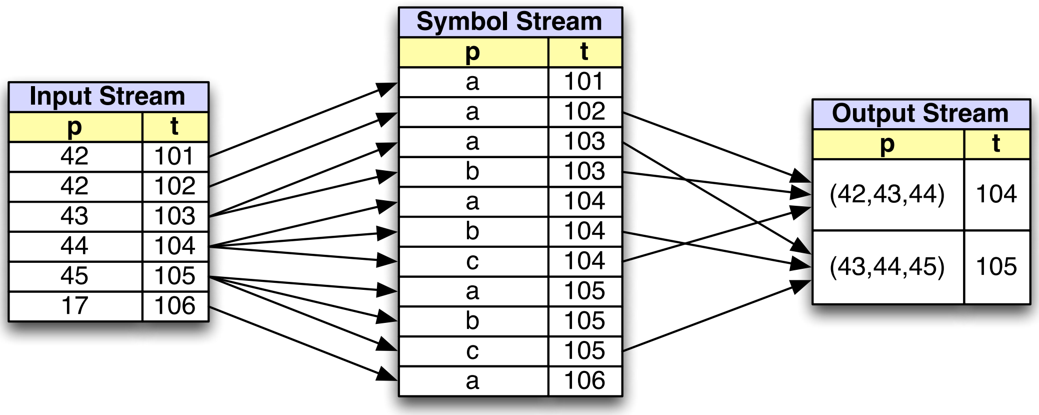


Figure 6.2: Pattern matcher example

The function κ ignores all Kleene operators that are applied to the first and last symbol in P. In the case of a Kleene star, the symbol is removed completely from the beginning or ending of the pattern. And in the case of a Kleene plus, only the operator is removed. This ensures that every match is reported only once and only for the shortest of all sequences that match and contain the symbol events $(s_i, t_i), \ldots, (s_x, t_x)$. Based on the potentially rewritten pattern definition and a sequence of symbol events, the problem has been reduced to an ordinary regular expression matching task. Thus, the semantics of "matches" is the same as in regular expression matching.

Exactly as the aggregator, a pattern matcher can be used in combination with partitioning according to event attributes $A_1, A_2, \ldots, A_n$:

$$\rho_{A_1,A_2,\ldots,A_n,P,w}(E^L) := \bigcup_{e \in E^L} \rho_{P,w}\left(\sigma_{A_1=e.A_1 \wedge A_2=e.A_2 \wedge \ldots \wedge A_n=e.A_n}(E^L)\right)$$

Let us consider a concrete example based on the following configuration of parameters of the pattern matcher for an event stream with integer numbers as payload:

- $w := 3$
- $P := a_{var_1 \leftarrow p}[\texttt{TRUE}]b_{var_2 \leftarrow p}[p > var_1]c_{var_3 \leftarrow p}[p > var_2]$
- $return := (var_1, var_2, var_3)$

The pattern P defines the symbol sequence abc. At the first symbol a, the attribute p of the corresponding event is bound to the global variable var_1. The second symbol b is correlated with a, because the payload of its corresponding event must be greater than the value of var_1. Symbol c is defined in the same way as b, but correlated with symbol b instead of symbol a.[1] Therefore, the pattern defines all event sequences of size three having constantly increasing payloads. The output event consists of the values of all global variables and gives the exact sequence that has been detected. Figure 6.2 shows this configuration in action. The left-hand table gives an example event stream. In the table at the center, the corresponding symbol stream can be seen. Finally, the results are presented in the right-hand table.

Summary

In this chapter, we formally specify the semantics of JEPC queries. At first, we present a data model for event streams that models an event stream as a sequence of snapshots. On basis of the data model, we define an operator algebra that includes elementary operators for event filtering, event aggregation, event correlation and event pattern matching as well as window operators that allow for the evaluation of JEPC queries over arbitrarily large but finite sections of event streams.

[1]Due to transitivity of $>$, symbol c is also implicitly correlated with symbol a.

7

Design of the JEPC Core

Outline

7.1 Introduction

While the previous chapter formally specifies the semantics of JEPC queries on basis of an operator algebra, this chapter presents the main API of JEPC for creating and maintaining EP applications running JEPC queries.

7.2 Time, Events and Event Streams

According to the JEPC semantics, timestamps are from a discrete and totally ordered time domain. Our implementation uses 64-bit integer numbers for the timestamps of events and has multiple different representations of events. We first name and define the more natural representation of events.

Definition 1 (External Event)**.** *An event in the form of a pair (p, t) having some payload p and a timestamp t is called an* external event. *Such external events occur in the outside world of JEPC and can be directly injected into JEPC.*

External events follow the intuitive interpretation of an event and are usually represented as such pairs outside of JEPC. However, events are represented and processed in the form of triples inside JEPC.

Definition 2 (Internal Event)**.** *An event in the form of a triple (p, t_s, t_e) having some payload p as well as two timestamps t_s and t_e with $t_e > t_s$ is called an* internal event. *The meaning of the timestamps is that the payload is valid from t_s (inclusive) to t_e (exclusive). Alternatively, the valid time can also be encoded as a half-open time interval $[t_s : t_e)$.*

Recall that the valid time of an individual event can be extended by the window operators. The naïve method of processing an internal event with payload p being valid in $[t_s : t_e)$ is to process exactly one event instance (p, t) for each timestamp t in $[t_s : t_e)$. Because every instance must be processed separately and causes costs, JEPC uses a more compact representation internally. In total, there are two different approaches. In the positive-negative approach (PNA) [ABW03, Ham03] two event instances (p, t) must be processed for each individual event. The first instance has t_s as timestamp and signals the beginning of the valid time of the event. And the second instance has t_e as timestamp signaling the ending of its valid time. In the time interval approach [KS05, Krä07] the valid time is encoded as a time interval so that only one event instance must be processed for each individual event. Obviously, internal events of JEPC implement the time interval approach. We preferred it in our implementation, because it has been proven to be more efficient than the PNA [Krä07].

Definition 3 (Chronon)**.** *A* chronon [Ari86, CR87] *is the smallest segment of the time line and, thus, the atomic unit of time. Since the time domain of JEPC is discrete, chronons help to represent the continuity of the real time line.*

Because external events are injected into JEPC but only internal events can be processed by JEPC, an external event must be somehow transformed into an internal event directly after its injection. Therefore, every external event (p, t) is mapped to the equivalent internal event $(p, t, t + 1)$. Time intervals of size one are called *chronons* (see Definition 3) that are the continuous equivalents of discrete points in time.

```java
Attribute[] serverEventSchema = new Attribute[] {
  new Attribute("serverID",   DataType.SHORT),   // identification of server
  new Attribute("cpuUser",    DataType.FLOAT),   // CPU utilization by users
  new Attribute("cpuSystem",  DataType.FLOAT),   // CPU utilization by system
  new Attribute("memory",     DataType.FLOAT),   // memory utilization
  new Attribute("disk",       DataType.FLOAT),   // disk utilization
  new Attribute("packetsIn",  DataType.INTEGER), // network packets per second in
  new Attribute("packetsOut", DataType.INTEGER)  // network packets per second out
};
```

Listing 7.1: Schema definition in JEPC

```java
Object[] serverEvent = new Object[] {
  42,    // identification of server
  0.34f, // CPU utilization by users
  0.16f, // CPU utilization by system
  0.68f, // memory utilization
  0.51f, // disk utilization
  8755,  // network packets per second in
  9697   // network packets per second out
};
```

Listing 7.2: Payload of an event in JEPC

In JEPC, each event stream has a relational schema and the payload of a corresponding event is simply a tuple. An event stream schema is represented as a list of attributes. Each attribute has a name that must be unique within a schema and a data type. Listing 7.1 shows a valid schema definition for a stream of measurements each keeping the current utilization of a server machine. Note that the temporal dimension is not specified. For external event streams an additional attribute keeping the timestamps is automatically added. Analogues, two additional attributes for the start and end timestamps are automatically added in the case of internal event streams.

JEPC data type	*Corresponding Java data type*
`DataType.BYTE`	`java.lang.Byte`
`DataType.SHORT`	`java.lang.Short`
`DataType.INTEGER`	`java.lang.Integer`
`DataType.LONG`	`java.lang.Long`
`DataType.FLOAT`	`java.lang.Float`
`DataType.DOUBLE`	`java.lang.Double`
`DataType.STRING`	`java.lang.String`
`DataType.GEOMETRY`	`java.lang.String`

Table 7.1: JEPC data types

Table 7.1 shows the available JEPC data types that all correspond to Java data types. The JEPC data type `GEOMETRY` is also for keeping Java strings. However, `GEOMETRY` indicates that the attribute values are geometries being encoded as WKT (well-known-text) strings. For more details of this extra data type that is for supporting spatial features in JEPC see Section 15.5 on page 213.

The payload of an event is represented as a list of attribute values. Of course, the structure of the payload must conform to the associated schema. Listing 7.2 shows a syntactically correct payload conforming to the schema of Listing 7.1.

7.3 Event Processing Agents and Networks

Queries can be defined directly as operator graphs in JEPC. All basic types of EPAs as well as their parameters are modeled as simple Java classes and EPA objects can be arbitrarily composed to EPNs. In the following, we explain the creation of EPAs and EPNs in detail. Parameters are introduced when they are used for the first time.

```
Stream(String streamName, Window window) : EPA
```

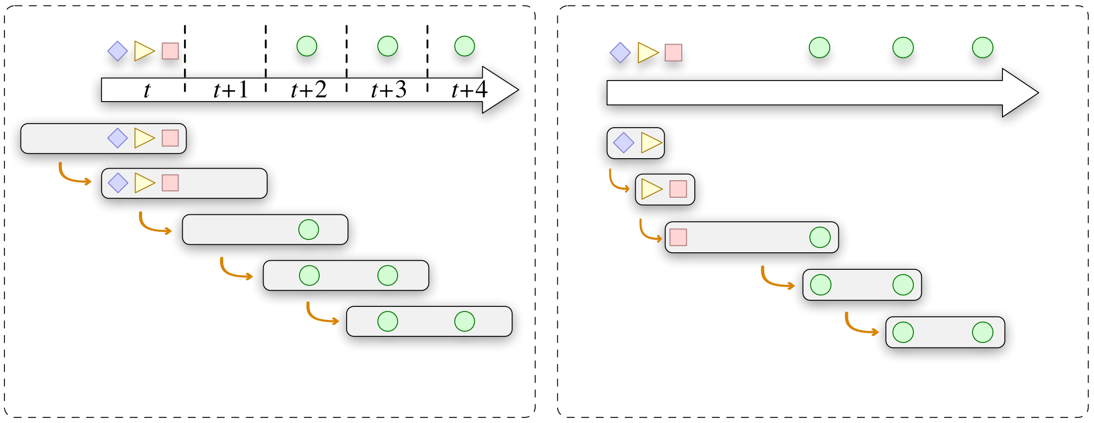
Listing 7.3: Signature of Stream

The EPA `Stream` is for accessing arbitrary event streams. Listing 7.3 shows its most general signature. The parameter `streamName` is mandatory and specifies the name of the event stream to access. Window operators can be applied to external event streams as first operator to extend the valid times of incoming events. For this purpose, the optional parameter `window` can be used to specify a window to apply. According to the operator algebra, JEPC supports three different types of windows.

```
// sliding windows
TimeWindow(long size) : Window
CountWindow(long size) : Window
PartitionedCountWindow(long size, String partitionAttribute,
                                        String... partitionAttributes) : Window
// jumping windows
TimeWindow(long size, long jump) : Window
CountWindow(long size, long jump) : Window
PartitionedCountWindow(long size, long jump, String partitionAttribute,
                                        String... partitionAttributes) : Window
```

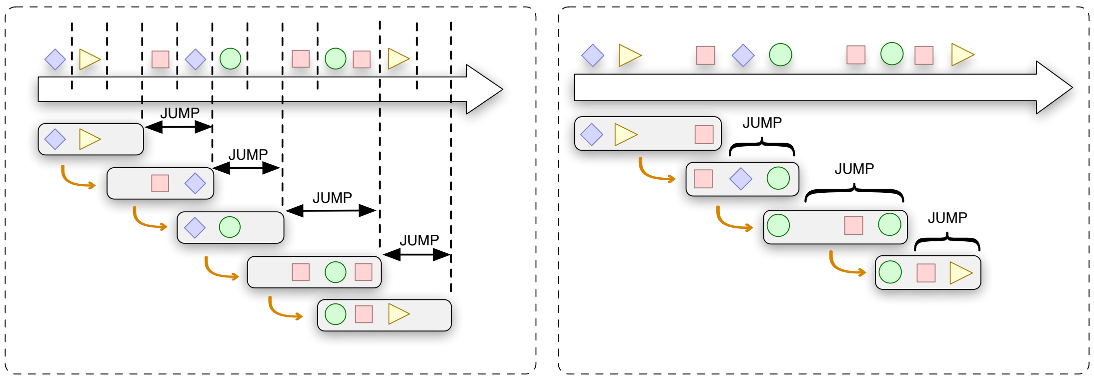
Listing 7.4: Signatures of Window

The first three signatures in Listing 7.4 belong to the time-based, count-based and partitioned count-based sliding windows. All types have a user-defined size that is specified by the parameter `size`. For time-based sliding windows this parameter gives the number of time instants a window covers and for the other two types this parameter gives the number of events a window covers. In case of partitioned count-based sliding windows, `size` is used for the count-based sliding windows that are applied to the partitions. The partitioning is done according to user-defined event attributes. These three window types constitute the class of sliding windows that slide from time instant to time instant or from event to event respectively. They should be used by default because they give the best possible response times of queries. Note that the sliding windows exactly implement the operators ω^{time}, ω^{count} and $\omega^{partitioned}$.



a) Sliding time window of size 2 **b) Sliding count window of size 2**

Figure 7.1: Sliding windows



a) Jumping time window of size 3 **b) Jumping count window of size 3**

Figure 7.2: Jumping windows

Figure 7.1 illustrates a time-based sliding window and a count-based sliding window, each of size 2, applied to an example event stream given in the upper part. The time-based sliding window slides from time instant to time instant, covers a fixed number of time instants and contains a variable number of events. In contrast, the count-based sliding window slides from event to event, covers a variable number of time instants and contains a fixed number of events.

In some cases, it may be desired that a query reports its results with a lower frequency (e.g., an aggregation EPA shall update its output only once every 15 seconds). Therefore, JEPC provides an extended version of each type of sliding window. The signatures of those so-called *jumping windows* [GÖ10] are shown in the lower part of Listing 7.4 and have the additional parameter `jump`. In case of time-based windows, this parameter specifies the period of time that must pass until a window is updated for the next time. And in case of count-based windows, the parameter `jump` specifies the number of events that must occur until a window is updated for the next time. Note that jumping windows are just slightly modified versions of the operators ω^{time}, ω^{count} and $\omega^{partitioned}$. The modified semantics is simply achieved by making the granularity of the time domain coarser. Details are found in [Krä07].

Figure 7.2 shows a time-based jumping window and a count-based jumping window in action. The corresponding input event stream is given in the upper part of the figure. Both jumping windows have the following configuration. The parameter `size` is set to 3 and the parameter `jump` is set to 2. While the time-based jumping window is updated after a period of two time instants has passed, the jumping count-based window is updated after two events have occurred.

Relationship	*Effect*
`jump` = 1	Jumping window becomes a sliding window
`jump` < `size`	Consecutive windows overlap
`jump` > `size`	Gaps between consecutive windows
`jump` = `size`	Consecutive windows meet (no gaps and no overlaps)

Table 7.2: Effects of window parameters

Note that the semantics of count-based windows is only clear provided that all events have unique timestamps [Krä07]. Otherwise, the output might not be deterministic. In our implementation of the count-based windows we decided therefore that simultaneous events are processed according to their order of insertion into JEPC. This fact is illustrated in the figures where some snapshots contain multiple events.

Based on the relationship between `size` and `jump` different effects occur. Table 7.2 lists all possible relationships and the resulting effects. The case `jump` > `size` is important to mention because gaps between consecutive windows occur. Events that are located in such a gap are not contained in any window and, thus, not processed.

All window operators are applied completely at the middleware layer by extending the time intervals of transformed external events accordingly. This ensures the specified semantics of the windows and is completely independent of the window types supported by raw EP providers as well as their semantics.

The core of JEPC abstracts from all basic types of EPAs via the interface `EPA`. With respect to the JEPC algebra, we provide Java classes that represent the four elementary EPAs. In addition, there are two extra types of EPAs for accessing external data. One of these extra EPAs, the EPA `Stream`, has been already introduced.

```
Filter(String name, EPA input, BooleanExpression be) : EPA
```

Listing 7.5: Signature of Filter

The EPA `Filter` corresponds to the operator σ. Its signature is shown in Listing 7.5. This EPA needs for instantiation a unique identification (`name`), another EPA (`input`) as well as a Boolean expression which defines the filter condition (`be`). The EPA `input` can be any other EPA. Its output event stream is used as input event stream for the EPA `Filter`. For creating Boolean expressions, the core of JEPC provides also Java classes. In particular, there are classes that represent the constant Boolean values `TRUE` and `FALSE`, all supported predicates and all supported Boolean operations.

```java
EPA overloadedServers = new Filter(
        "OverloadedServers",                            // identification
        new Stream("Servers"),                          // input event stream
        new And(                                        // filter condition
            new Greater("memory", 0.9f),
            new Greater("cpuUser + cpuSystem", 0.95f),
            new Greater("disk", 0.8f)
        )
);
```

Listing 7.6: Example of Filter

Listing 7.6 shows an example of the EPA `Filter` on an external event stream `Servers` having the schema of the example event stream from Section 7.2. Its purpose is to select overloaded servers. A server is considered to be overladed if its memory utilization is greater than 0.9 and its total CPU utilization (i.e., the sum of `cpuUser` and `cpuSystem`) is greater than 0.95 and its disk utilization is greater than 0.8.

```java
Aggregator(String name, EPA input, Aggregate... aggregates) : EPA
```

Listing 7.7: Signature of Aggregator

The EPA `Aggregator` corresponds to the operator α and has the signature shown in Listing 7.7. Besides a unique identification (`name`) and another EPA (`input`), it requires a list of aggregates (`aggregates`) being evaluated over the output event stream of the EPA `input`. Again, we provide Java classes for all supported aggregates. Grouping is supported in the form of an extra aggregate (`Group`). This grouping aggregate must be applied to each event attribute used for grouping.

```java
EPA smoothedServers = new Aggregator(
        "SmoothedServers",
        new Stream("Servers", new PartitionedCountWindow(5, "serverID")),
        new Average("memory",    "avgMemory"),
        new Average("cpuUser",   "avgCpuUser"),
        new Average("cpuSystem", "avgCpuSystem"),
        new Average("disk",      "avgDisk"),
        new Group("serverID")
);
```

Listing 7.8: Example of Aggregator

Listing 7.8 presents an example of the EPA `Aggregator`. The shown query specifies a window that contains the five most recent events from every monitored server. Then, it computes the average values of the event attributes `memory`, `cpuUser`, `cpuSystem`, and `disk` per server. The event attribute an aggregate refers to is set in its first argument. Since an aggregate (except `Group` that simply takes over the event attribute) results in a completely new attribute of the output schema, the name of the associated attribute is defined in the second argument. The data types of the resulting attributes are automatically determined and set by JEPC. Note that the grouping by `serverID` is necessary, because the window contains events from different servers.

```
PatternMatcher(String name, EPA input, long within, String partitionBy,
                                                  Pattern... pattern) : EPA
```

Listing 7.9: Signature of PatternMatcher

The EPA `PatternMatcher` corresponds to the operator ρ and has the signature shown in Listing 7.9. Besides the usual parameters `name` and `input`, this EPA can have an optional partitioning defined in the form of a comma-separated list of attribute names (`partitionBy`). Furthermore, it requires a size of its time window (`within`) and a pattern to search for (`pattern`). A pattern is simply a list of symbol definitions. Each single symbol definition can set any number of global variables and pattern operators can be applied to symbol definitions.

```
EPA diskCleaned = new PatternMatcher(
        "DiskCleaned",
        new Stream("Servers"),
        60_000,
        "serverID",
        new Symbol("a", new Greater("disk", 0.9f), new Variable("peak", "disk")),
        new KleeneStar(new Symbol("b", new And(new LessEqual("disk", "peak"),
                                            new GreaterEqual("disk", 0.8f)))),
        new Symbol("c", new Less("disk", 0.8f), new Variable("finished", "disk"))
);
```

Listing 7.10: Example of PatternMatcher

Listing 7.10 shows an example of the EPA `PatternMatcher`. The size of the time window is set to 60,000 time instants, the input stream is partitioned by `serverID` and the pattern being searched is ab^*c. Symbol a is emitted by all events reporting a disk utilization greater than 0.9. Simultaneously, a global variable named `peak` is set to the current value of disk. Because of the applied Kleene Star, symbol b is emitted by all subsequent events reporting a disk utilization less than or equal to the value of peak and greater than or equal to 0.8. The pattern matches when the disk utilization becomes less than 0.8. Then, a global variable named finished is set to the current value of disk and an output event with peak and finished as payload is created.

```
Correlator(String name, EPA input1, String var1, EPA input2, String var2,
                                                     BooleanExpression be) : EPA
```

Listing 7.11: Signature of Correlator

The EPA Correlator has the signature shown in Listing 7.11 and corresponds to the operator $\bowtie$. Since $\bowtie$ is a binary operator, the EPA Correlator consumes the output streams of two other EPAs (input1 and input2). Both streams must be assigned with different variables (var1 and var2), because they can have identically named attributes or even be identical (self correlation). Lastly, the EPA needs an identification (name) and a Boolean expression defining the correlation condition (be).

```
EPA increasedMemory = new Correlator(
        "IncreasedMemory",
        new Stream("Servers"), "servers",
        smoothedServers, "averages",
        new And(
                new Equal("servers.serverID", "averages.serverID"),
                new Greater("servers.memory * 0.8", "averages.avgMemory")
        )
);
```

Listing 7.12: Example of Correlator

An example of the EPA Correlator is given in Listing 7.12. The shown query consumes the external event stream Servers having the variable servers and the output event stream of the previously defined query smoothedServers having the variable averages. Combining these streams, the most recent values of a server are available together with the average values of most event attributes. The correlation condition specifies that only events belonging to the same server are correlated. In addition, it requires that the current memory utilization scaled down by factor 0.8 is greater than the average memory utilization. This example shows how to compose EPAs to EPNs by connecting the output of one EPA to the input of another EPA. Figure 7.3 depicts a graphical illustration of the EPN defined in Listing 7.12.

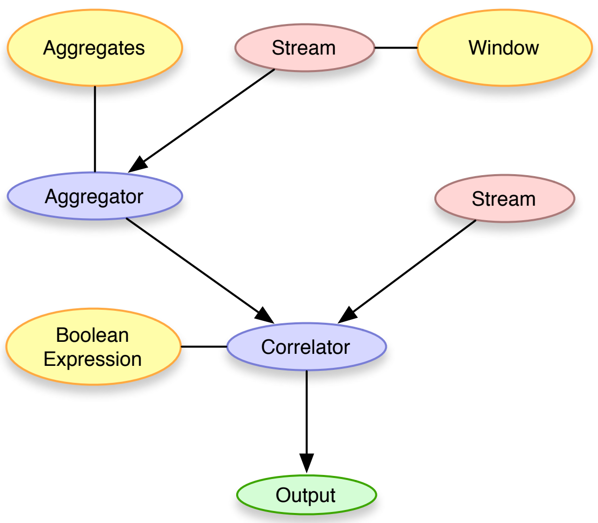


Figure 7.3: Event processing network

An instance of `Correlator` is allowed to have a special type of EPA named `Relation` as one of its inputs. The EPA `Relation` is similar to the EPA `Stream`. It also provides access to events that are added from the outside. But once added, an event of a relation is valid forever. This is achieved by setting the time interval to $[-\infty : \infty)$ during its transformation into an internal event. Relations are meant to represent context knowledge that is mostly static and usually stored in databases. The EPA `Relation` caches such data in order to make it available in JEPC queries. A relation can only be used as input for correlation EPAs and the other input must be a stream. Then, live events from the stream can be extended (content enrichment) or filtered on basis of data stored in the relation. For example, events from a fixed point sensor containing some measurements and the sensor identification as payload can be extended by the location being provided by a relation. Because also relations may change occasionally, users can insert, delete and update events at any time.

Besides the presented EPAs, JEPC can be extended by user-defined EPAs (UEPAs) that run at the middleware layer. An UEPA can consume any number of input event streams and provides exactly one output event stream. Thus, it can be placed anywhere in EPNs. Its operation is defined by users in the form of arbitrary Java code. A user-defined EPA must extend the abstract EPA `UserDefinedEPA` which contains all necessary code for the execution by JEPC. User-defined EPAs can extend JEPC by not only simple operators such as the set operators (e.g., union, difference), but also complex and novel operators such as TPStream (see Chapter 15).

The completely object-oriented representation of parameters, EPAs and EPNs leads to several advantages. First, declarative query languages and graphical query composers following the boxes-and-arrows principle can be directly used on top. The compilation of declarative and graphical query definitions is straightforward, because EPN definitions are operator graphs. Second, parameters, EPAs and EPNs can be easily analyzed. This is essential for their translation into another representation (e.g., JEPC bridges must translate EPAs and EPNs into the query languages of raw EP providers). Third, parameters, EPAs and EPNs can be easily modified by algorithms. For instance, a query optimizer can directly traverse and transform EPNs.

7.4 Output Processors

Output processors implement the concept of event sinks in JEPC. Any number of them can be added to an arbitrary EPA in order to access its output event stream. Analogues to Java listeners, an output processor is immediately notified when a new output event is available. As in the case of user-defined EPAs, users can define arbitrary output processors by extending the abstract class `OutputProcessor`.

```java
OutputProcessor prettyPrinter = new OutputProcessor() {
  @Override
  public void process(String stream, Object[] event) {
      Attribute[] schema = getInputSchema(stream);
      for(int i = 0; i < event.length; i++)
          System.out.print(schema[i].getAttributeName() + "=" + event[i] + " ");
      System.out.print("\n");
  }
};
```

Listing 7.13: Example of OutputProcessor

Listing 7.13 shows the definition of a simple output processor that writes every received event to the standard output. The only method which must be implemented by users is `process`. This method is called on every new event and gets not only the event itself, but also the identification of the corresponding event stream. Therefore, a universally defined output processor can be reused and added to multiple different EPAs. Besides logging, possible reactions to new events range from updating visualizations to triggering actions (including complex analytics and decision-making).

Method	*Description*
REGISTERSTREAM(e, s)	Registers a new event stream e having the schema s
UNREGISTERSTREAM(e)	Unregisters the event stream e
CREATERELATION(r, s)	Creates a new relation r having the schema s
DESTROYRELATION(r)	Destroys the relation r
CREATEQUERY(q)	Creates a new EPA q
DESTROYQUERY(q)	Destroys the EPA q
ADDOUTPUTPROCESSOR(q, o)	Adds the output processor o to the EPA q
REMOVEOUTPUTPROCESSOR(q, o)	Removes the output processor o from the EPA q
PUSHEVENT(e, p, t)	Pushes the event (p, t) into the event stream e
INSERTINTORELATION(r, p)	Inserts the payload p into the relation r
DELETEFROMRELATION(r, p)	Deletes the payload p from the relation r

Table 7.3: API of EP providers

7.5 Event Processing Providers

The interface `EPProvider` specifies the core API of JEPC that is the API of a virtual EP provider. This interface is intended to be implemented in the form of JEPC bridges by not only raw EP providers, but also other kinds of data management systems (e.g., standard database systems) and native implementations (e.g., for highly parallel computing platforms such as GPUs and FPGAs, or for ordinary CPUs but implementing tailor-made algorithms). Applications and extensions which are using only methods of `EPProvider` become independent of concrete EP providers.

Table 7.3 shows an excerpt of the interface `EPProvider` that includes the most important methods. The core API provides methods to register and unregister external event streams. Via PUSHEVENT, external events are injected into registered event streams. There are also methods for the management of relations and their contents. Arbitrarily complex JEPC queries can be created using CREATEQUERY. Of course, there is also a method for destroying existing EPAs. Lastly, output processors can be added to and removed from existing EPAs via the core API.

The core API of JEPC can be accessed in two different ways. Java applications can use JEPC directly as a library. For remote and platform-independent access, JEPC can also be executed in a server mode. Then, its API is made available via HTTP by a REST Web service [Fie00]. For defining the elements of a JEPC application such as EPAs and schemas, a query language is provided. External events are injected in serialized form directly via TCP and not via the Web service for performance reasons.

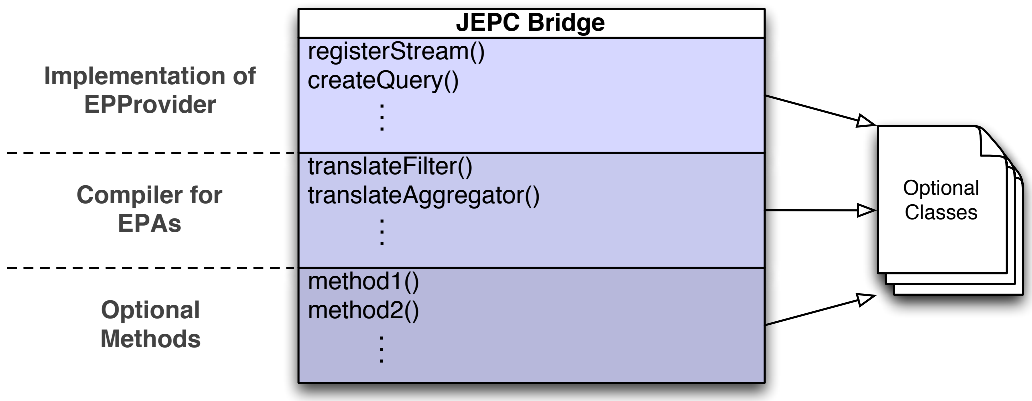


Figure 7.4: Typical structure of a JEPC bridge

Figure 7.4 illustrates the overall structure of a typical JEPC bridge. All JEPC bridges we implemented (i.a., multiple JEPC bridges to different SPEs, a JEPC bridge to standard database systems and a native CPU-based implementation of the JEPC specification) have this overall structure. In total, there are three main parts as shown in the figure. The first part is an implementation of the interface `EPProvider`. Here, the core API of JEPC is mapped to the API of the target raw EP provider. The second part is a compiler for JEPC queries. For each basic type of EPA there is a method that translates it into the corresponding query language of the raw target EP provider under preservation of semantics. We decided for a fine-grained translation of single EPAs. This is not only the easiest way to create compilers for JEPC queries, but also the most flexible way that allows to arbitrarily distribute EPAs (see Chapter 20 for details). The third part consists of optional help methods which might be necessary or convenient. Lastly, a JEPC bridge must not consist of a single Java class only. If the code base is large or complex, parts of it can be separated out into additional Java classes.

7.6 JEPC Query Language

Despite the fact that the object-oriented representation of all elements of a JEPC application leads to several advantages, it is not the best representation for end-users and for remote access. Therefore, we provide a declarative query language named *JEPC-QL*. Our query language is based on standard SQL [SQL] and offers extensions for window operators as well as for pattern matching. In the following, we would like to give a brief impression of it.

```
(SELECT *
 FROM   Servers
 WHERE  memory > 0.9f AND cpuUser + cpuSystem > 0.95f AND disk > 0.8f
) AS OverloadedServers
```

Listing 7.14: Filter in JEPC-QL

A markable difference to standard SQL is that every part of a query defining an EPA requires a unique identification. This is because EPAs run continuously in contrast to database queries that are ad-hoc. The identification is needed to destroy the corresponding EPA and to access its output stream (e.g., via output processors or other EPAs). Listing 7.14 shows how to express the filter EPA defined in Listing 7.6.

```
(SELECT *
 FROM   Servers AS servers,
        (SELECT   AVG(memory) AS avgMemory, AVG(cpuUser) AS avgCpuUser,
                  AVG(cpuSystem) AS avgCpuSystem, AVG(disk) AS avgDisk
         FROM     Servers WINDOW(COUNT 5 EVENTS JUMP 1 EVENT PARTITION BY serverID)
         GROUP BY serverID ) AS averages
 WHERE  servers.serverID = averages.serverID
   AND  servers.memory * 0.8 > averages.avgMemory
) AS IncreasedMemory
```

Listing 7.15: EPN consisting of an Aggregator and a Correlator in JEPC-QL

Listing 7.15 shows the syntax for aggregation and correlation EPAs as well as for windows. It also demonstrates how to define an entire EPN in one go. Note that EPNs must not necessarily be specified as a whole. Existing EPAs can be accessed by using their identification in the `FROM` clause. The subquery `averages` defines the aggregator presented in Listing 7.8. All attributes used for grouping (only `serverID` in this particular example) are automatically added to the output schema. In addition to the syntax of standard SQL, external event streams in the `FROM` clause can have an assigned window that is defined using the keyword `WINDOW`. In the subsequent brackets, all parameters of the window are specified. First of all, its type must be specified by using either the keyword `TIME` or the keyword `COUNT`. The former defines a time-based window while the latter defines a count-based window. Next, the size of the window must be defined. In case of a count-based window, the size is defined in number of events. And in case of a time-based window, the size is specified in the form of a time expression. By default, JEPC interprets the length of a chronon as one millisecond. Legal time expressions are for instance `200 MILISECONDS`, or `1 SECOND`, or `45 MINUTES`. Optionally, a jumping behavior can be achieved by using the keyword `JUMP` followed by the number of events in case of a count-based window or a time expression in case of a time-based window. If `JUMP` is not specified, it is automatically set to 1 event or 1 millisecond respectively (in Listing 7.15 it is specified for the sake of completeness). An optional partitioning can be defined by using the keyword `PARTITION BY` followed by a list of attributes. Note that partitioning can also be specified for time-based windows in JEPC-QL but it has no effect. The outer query `IncreasedMemory` defines a correlation EPA that joins the output stream of the aggregator and the external event stream `Servers` according to Listing 7.12. Note that JEPC-QL supports multiway correlations. Our compiler automatically translates multiway correlations into a hierarchy (a left-deep tree to be exact) of multiple correlation EPAs, decomposes the correlation condition and assigns the resulting parts to the correlation EPAs such that the multiway correlation can be efficiently executed.

```
(SELECT *
 FROM   Servers
 MATCH_RECOGNIZE_SEQUENTIAL (
   MEASURES     a.disk AS peak, c.disk AS finished
   WITHIN       1 MINUTE
   PARTITION BY serverID
   PATTERN      ab*c
   DEFINE       a AS disk > 0.9f,
                b AS disk <= peak AND disk >= 0.8f,
                c AS disk < 0.8f
 )
) AS DiskCleaned
```

Listing 7.16: PatternMatcher in JEPC-QL

The syntax for pattern matching EPAs has been adopted from match-recognize queries (MRQ) [Zem07]. Listing 7.16 shows the JEPC-QL definition of the pattern matching EPA presented in Listing 7.10. Sequential pattern matching EPAs are defined by using the keyword `MATCH_RECOGNIZE_SEQUENTIAL`.[1] For each parameter of a pattern matching EPA there is a corresponding clause in JEPC-QL. The size of the time window is specified in the `WITHIN` clause using a time expression. Partitioning is specified in the optional `PARTITION BY` clause by listing the partitioning attributes. In the

[1]JEPC also provides a novel type of EPA for temporal pattern matching (see Chapter 15 for details). Temporal pattern matching EPAs have exactly the same JEPC-QL syntax but begin with the keyword `MATCH_RECOGNIZE_TEMPORAL`.

PATTERN clause the sequential pattern being searched is defined in the form of a regular expression over symbols. Note that pattern operators are applied in the PATTERN clause as in the example above. Conditions of symbols used in the pattern are defined in the DEFINE clause. If a used symbol is not defined here, its condition is automatically set to TRUE which means that the symbol is emitted by every event. The binding of global variables is done in the MEASURES clause.

Summary

This chapter presents the design of the core of the JEPC middleware. In particular, its API is described and illustrated on basis of a comprehensive collection of examples. The core API provides everything that is needed to create and maintain platform-independent EP applications. It can be used directly from Java and via a declarative query language from everywhere.

8

JEPC Bridges to Stream Processing Engines

Outline

8.1 Introduction

Today, general-purpose stream processing engines are widely used for EP in practice. They are not only capable of fully supporting EP, but also usually offer functionality that goes beyond EP. The development of multiple JEPC bridges to different stream processing engines took some time. This is because a target SPE must be studied extensively with respect to execution behavior and query semantics first. Then, an efficient mapping of the JEPC specification must be found and implemented. Lastly, unit tests that check test workloads for correct processing must be performed. Because most of these tasks are creative and intellectual, no holistic approach to developing a JEPC bridge to a stream processing engine can be given. Nevertheless, this chapter presents some details of our solutions that very likely can be adopted by additional JEPC bridges to other stream processing engines. In general, the mapping of the core API of JEPC (i.e., implementing the interface `EPProvider`) is easy, because it is a subset of the API of every modern SPE. The semantically correct translation of EPAs into the query language of a target SPE is more challenging.

8.2 Implementation of the API

As already mentioned in the last section, mapping the core API of JEPC to the API of a SPE is straightforward, because both are made for stream processing and require the management of streams and continuous queries therefore. Due to its simplicity and orientation towards existing systems, every method of the API of JEPC has an exact or nearly exact counterpart in the API of a target SPE.

Also the architectural style of a target SPE does not matter. We implemented JEPC bridges to the SPEs Esper [Esp] and webMethods Business Events [web] which follow a library approach. Their corresponding JEPC bridges only need access to all libraries and can then create and use their own instances of the associated target SPE. We also implemented a JEPC bridge to the SPE Odysseus [App12] which implements a client-server architecture. Its corresponding JEPC bridge requires a running server instance and is fully responsible for the communication with the server in addition. For example, events must be transferred in both directions. Because such a JEPC bridge has to implement also an entire client therefore, the mapping of the core API requires more effort and lines of code but is not more difficult than the mapping of the core API to the API of SPE following the library approach.

8.3 Compiler for EPAs

Translating every basic EPA into the query language of a target SPE under preservation of semantics is challenging in general. Depending on the degree of mismatch between the semantics of JEPC and the semantics of a target SPE, correct translations might be complicated to achieve. However, all modern SPEs support filter, aggregation, join and sequential pattern matching queries. A roughly correct translation of EPAs should be possible in every case therefore. For an exact alignment of the semantics, it might be necessary that some tasks are performed within the JEPC bridge before and after a translated EPA running within the target SPE. Figure 19.1a on page 292 illustrates the translation of an EPA shown in the upper part. A corresponding CQ that exactly or approximately produces the results of the EPA runs within the target SPE. In general, manipulations of the input streams or the output stream of this CQ are necessary to meet the JEPC semantics. Among others, time intervals of events can be modified, extra events can be added and unwanted events can be removed.

Since all window operators are applied within the middleware, only the basic EPAs must be translated. A translation must process arbitrary internal event streams correctly. In the following, we shortly discuss the translations of the basic EPAs.

8.3.1 Translation of the Snapshot-Reducible EPAs

We focus on the basic EPAs that are snapshot-reducible in this section. From a high-level point of view, EPAs can be classified into stateless EPAs and stateful EPAs. In the context of JEPC, the filter EPA is stateless and the aggregation and correlation EPAs are stateful. Translating stateless EPAs correctly is simple in almost every case. Particularly the filter EPA of JEPC is easy to translate, because all modern SPEs provide a mechanism for event filtering on basis of user-defined Boolean expressions.

In contrast, stateful EPAs are challenging to translate, because they have an assigned state that keeps all relevant events. There are SPEs such as Esper that have a data model, execution behavior and semantics that totally mismatch the JEPC specification. For example, Esper models every event stream as a totally ordered sequence of external events (the order is according to the arrival order of events) and does not support internal events. A sliding window aggregation query produces for each event of the input event stream exactly one output event (and not only for entire snapshots as required by the JEPC specification). In the case of a high degree of mismatch between the semantics of JEPC and a target SPE, it is recommended to manually manage the state of stateful EPAs in order to achieve correct translations.

Method
NEWSWEEPAREA($\leq, f_{query}, f_{remove}$)
INSERT((p, t_s, t_e))
REPLACE($(\hat{p}, \hat{t}_s, \hat{t}_e), (p, t_s, t_e)$)
ITERATOR()
QUERY((p, t_s, t_e))
EXTRACTELEMENTS((p, t_s, t_e))
PURGEELEMENTS((p, t_s, t_e))
SIZE()

Table 8.1: SweepArea

In [Krä07, KS09] so-called *SweepAreas* are proposed as data structures for the states of stateful EPAs that process internal event streams. A SweepArea is an abstract data type for keeping and managing a collection of internal events. Table 8.1 shows all methods of a SweepArea. A new SweepArea is initialized with a total order relation $\leq$ and two binary predicates f_{query} and f_{remove} each requiring two events as arguments. Events are added to a SweepArea via INSERT. It is also possible to replace a stored event by another one using REPLACE. The method ITERATOR returns an iterator that traverses all stored events in an order according to $\leq$. To select and obtain only certain events, the method QUERY can be used. Then, an event is only returned if it and a reference event given by the caller satisfy f_{query}. A subset of all stored events is returned and removed from the SweepArea by EXTRACTELEMENTS that returns and removes a stored event if $f_{removed}$ is satisfied for the event itself and a reference event given by the caller. The method PURGEELEMENTS is similar but does not return qualifying events. Lastly, SIZE is for obtaining the size of a SweepArea.

SweepAreas are intended for low-level implementations of stateful EPAs. This comprises both the implementation of the state and the implementation of the operation. For the translations of stateful EPAs only the storage capabilities of SweepAreas are needed. The operation is performed by a corresponding CQ of the target SPE. Therefore, it is sufficient to simulate a lightweight version of SweepAreas within the target SPE. In particular, only f_{remove}, INSERT and PURGEELEMENTS are necessary to fully control the content of a SweepArea. In case of Esper, we could easily simulate such a lightweight SweepArea by using only data structures and functionality provided by Esper. Then, aggregation and correlation queries over our lightweight SweepAreas result in correct results with respect to the JEPC semantics.

8.3.2 Translation of the Pattern Matching EPA

Although pattern matching is a complicated operation in general, sequential pattern matching queries have quite similar semantics among different SPEs. Furthermore, their semantics can often be manipulated via parameters. In the case of the most popular approaches to sequential pattern matching, namely SASE [WDR06, DIG07] and MRQ [Zem07], users can choose from different semantics. Because SASE is implemented by the SPE Odysseus and MRQ is implemented by the SPE Esper, we had to develop translations of the pattern matching EPA for SASE and MRQ. For both we identified parameter configurations so that the JEPC semantics is met in the case of strict event sequences. Unfortunately, SASE as well as MRQ always force strict event sequences by ordering input events according to their insertion order. This mismatches the JEPC semantics that interprets simultaneous events as alternatives.

We found a solution that works for both SASE and MRQ. Instead of mapping a pattern matching EPA to exactly one sequential pattern matching query of the target SPE, we map it to a variable number of sequential pattern matching queries each responsible for exactly one active universe currently being contained in the time window. Because a universe is a strict sequence of events, the JEPC semantics is met for single universes. Therefore, the JEPC semantics is met for arbitrary event streams by deriving all universes currently being contained in the time window, placing a sequential pattern matching query on each universe, and joining the output streams of all those queries. The corresponding JEPC bridge must take care that new universes and sequential pattern matching queries are created when events occurred simultaneously, that an existing universe is removed when the event by which it was started leaves the time window, and that a new event is forwarded only to sequential pattern matching queries being responsible for a universe the event belongs to.

Summary

In this section, we discuss the implementation of JEPC bridges to SPEs and present techniques for the alignment of semantics. While the mapping of the API of JEPC and the translation of filter EPAs are easy in general, the translations of the stateful snapshot-reducible EPAs and of the pattern matching EPA are difficult if the semantics substantially mismatch. We show how a lightweight version of the SweepArea can be used to achieve correct semantics for the stateful snapshot-reducible EPAs. Furthermore, we present a translation of the pattern matching EPA to SASE and MRQ which are popular and widely used approaches to sequential pattern matching.

9

JEPC Bridge to Standard Database Systems

Outline

9.1 Introduction

The main purpose of JEPC is primarily the connectivity to high-performance raw EP providers. But a JEPC bridge to standard database systems, whose performance cannot be expected to be very good, is appealing to some applications. Many applications are not dealing with enormous event streams. If there happen only a few dozens of events every second, a standard database system can easily handle them. In addition, a standard database system allows stateful EPAs to have states that are significantly larger than the available main memory, because states are kept on external memory. Lastly, a JEPC bridge to standard database systems leads to reduced integration effort when event processing technology is used for the very first time. Nearly all IT infrastructures consist of one or more database systems. Instead of adding and integrating a new kind of system, one of the existing database systems could perform the processing of event streams. However, if the workload increases someday so that a standard database system cannot handle it anymore, then switching to a high-performance EP provider is immediately and seamlessly possible because of JEPC.

Our JEPC bridge to standard database systems [Glo13, Hoß13] implements an EP provider purely on top of JDBC [And11, GUW08]. This allows the use of the JEPC bridge with every standard database system for which a JDBC driver exists. Figure 9.1 illustrates how event processing is provided by a standard database systems via the combination of JDBC and JEPC.

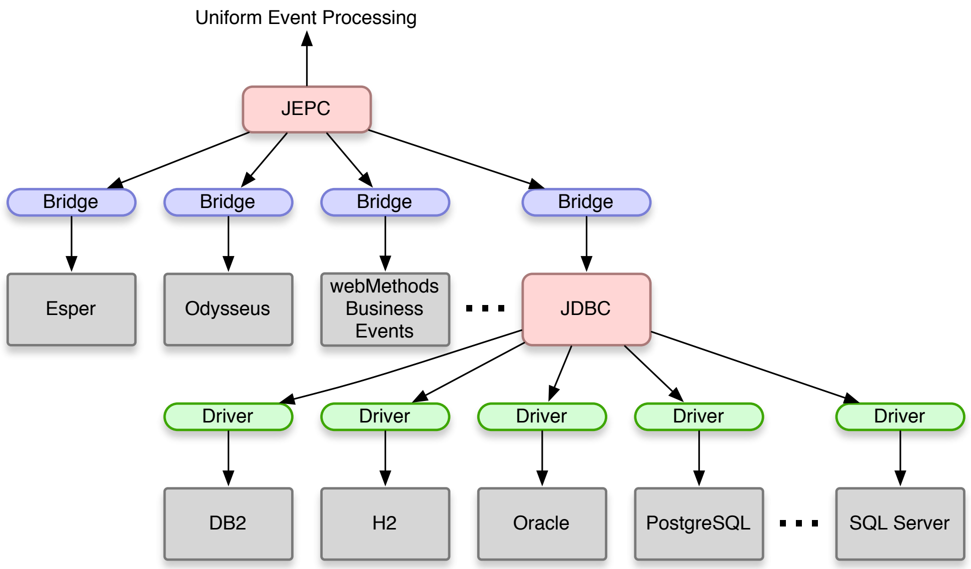


Figure 9.1: Event processing via standard database systems

9.2 Implementation of the JEPC Bridge to JDBC

While the mapping of the core API of JEPC is easy in case of SPEs, it requires more effort in case of JDBC because standard database systems do not conform to the stream processing paradigm. Queries must be kept within the JEPC bridge and manually executed whenever a new relevant event is pushed. However, the translation of the snapshot-reducible basic EPAs is straightforward, because their semantics is derived from the semantics of standard database systems. In contrast, sequential pattern matching is not supported by standard database systems. Therefore, we show how sequential pattern matching can be implemented purely via standard SQL [SQL].

9.2.1 Management of Events in Database Tables

Standard database systems organize data in the form of tuples in static database tables. Tuples have no inherent temporal dimension. They are inserted into and removed from database tables completely on demand. Because the payloads of events are tuples, they can be directly stored in database tables. Furthermore, the temporal dimension of events can be added by simply extending database tables by two new columns keeping the start and end timestamps of events. This way, an event stream can be stored in a database table. But storing entire event streams is neither required nor convenient for the evaluation of EPAs. Therefore, the basic idea is to keep only events which are relevant for correct evaluation of JEPC queries in database tables. Consequently, expired events must be continuously removed as new events stream in. On basis of the available end timestamps of events, expired events of a database table can be easily selected and removed via a single SQL statement.

Since events can be managed in database tables, we have to clarify which and how many database tables are needed. For each event stream (i.e., all externally registered event streams as well as the output stream of every running EPA) the JEPC bridge creates exactly one fixed table and a variable number of additional tables. The fixed table, named *inbox*, keeps always the last event of the associated event stream. On arrival of a new event, the corresponding inbox of an event stream is updated by exchanging the event currently being stored for the new one. For each stateful EPA, one or more tables are created in addition to the inboxes. Those additional tables are used as the states of stateful EPAs and keep all relevant events of the associated input event streams. A new event must be inserted into not only the inbox, but also all associated tables being the states of running stateful EPAs.

9.2.2 Translation of the Snapshot-Reducible EPAs

According to the JEPC semantics, the basic EPAs filter, aggregator and correlator of JEPC are snapshot-reducible to the selection, aggregation and join operators of the relational algebra. This makes the translation of these basic EPAs into SQL statements straightforward. The execution of the resulting SQL statements is performed as follows. A filter statement is executed on the inbox of its input event stream for each arriving event. Aggregation statements are directly executed on the corresponding state tables when time progresses. A correlation statement requires a more advanced execution in order to avoid duplicates of output events. It is not correct to join all corresponding state tables of a correlation EPA. In general, this also joins events that already have been joined. Instead, the inbox of the input stream having a new event must be joined with the state tables of all other input streams. Additionally to the user-defined correlation condition, only events with intersecting time intervals can successfully join. This is simply achieved by extending the correlation condition.

9.2.3 Translation of the Pattern Matching EPA

Because sequential pattern matching is not part of the relational algebra, there is no operator corresponding to the pattern matching EPA. Therefore, we developed an approach which allows standard database systems to perform sequential pattern matching purely on database tables using standard SQL. A slightly modified version of our approach requiring an external automaton can be found in [Hoß13].

Recall that the pattern to detect is a regular expression over symbols. It is common practice to use finite automata for regular expression matching [HMU00]. Figure 9.2 shows an automaton for detecting the pattern ab^*c^+d over $\{a, b, c, d\}$. Automata are implemented typically in the form of transition tables. For each combination of automaton state and symbol, a transition table keeps the next automaton state [HMU00].

	a	b	c	d
q_0	q_1	q_e	q_e	q_e
q_1	q_e	q_2	q_3	q_e
q_2	q_e	q_2	q_3	q_e
q_3	q_e	q_e	q_3	q_4
q_4	q_e	q_e	q_e	q_e
q_e	q_e	q_e	q_e	q_e

Table 9.1: Transition table of example automaton (standard layout)

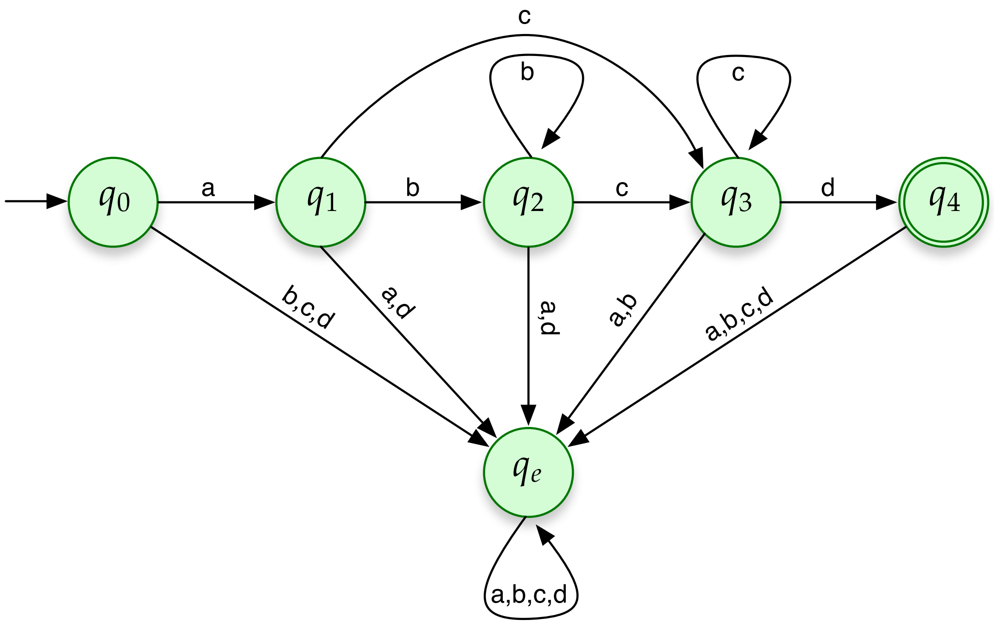


Figure 9.2: Finite automaton for regular expression matching

Table 9.1 shows the transition table of the automaton depicted in Figure 9.2. The first row gives all symbols and the first column gives all automaton states. All other cells contain the target automaton states. A transition table can be directly put into a database system as a new database table. However, the usual layout of transition tables is beneficial for main memory but not for external memory because of the following reasons. First, this layout cannot be efficiently indexed and lookups are expensive therefore. Second, database systems read entire rows in general. The amount of data that is read for a lookup increases with the number of symbols. Therefore, we propose a layout which does not suffer from these disadvantages. In our proposed layout, each row of a transition table is a triple (q_x, s, q_y). The element q_x is the current automaton state, s is the symbol and q_y is the next automaton state.

The corresponding transition table of the automaton depicted in Figure 9.2 having the new layout is shown in Table 9.2. Because also all symbols are now organized in a column, we can create a compound index on $[SourceState, Symbol]$. Then, lookups for a given state and symbol can be done efficiently via the index. Furthermore, rows are only of size three and all transitions to the error state q_e must not be stored. Whenever a lookup fails, the next state is q_e by default. Lookups for an automaton with 1,000 states and 1,000 symbols were more than four times faster using a transition table with the new layout than with the usual layout in PostgreSQL [Pos].

Source State	*Symbol*	*Target State*
q_0	a	q_1
q_1	b	q_2
q_1	c	q_3
q_2	b	q_2
q_2	c	q_3
q_3	c	q_3
q_3	d	q_4

Table 9.2: Transition table of example automaton (efficient layout for databases)

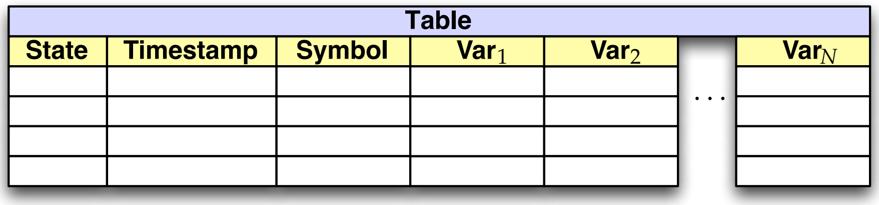


Figure 9.3: Schema of the AIT

Besides the transition table implementing the automaton, each pattern matching EPA gets another database table named *active instances table* (AIT) where all active instances are managed. Figure 9.3 shows the schema of the AIT. Each active instance is represented by exactly one row in the AIT. The column *Symbol* keeps the last matched symbol of an active instance, the columns $Var_1, Var_2, \ldots, Var_N$ store for each global variable the current value, *Timestamp* gives the point in time when the active instance started and *State* points to the current automaton state. An AIT always contains a fixed row being in the initial automaton state. It is required for starting new active instances. Whenever time progresses, the AIT is updated and not changed until time progresses again. All updates (i.e., new active instances that started and existing active instances that changed their state) within the same instant of time are logged in a second table having the same schema as the AIT. This table is only temporary and named *buffer table* (BT). The BT is needed to derive new universes on simultaneous events and symbols. Immediately after time progressed, all updates are moved from the BT into the AIT. Lastly, there is a third table having also the schema of the AIT (but its first column is named *NewState*). This table, named *Transitions*, logs the transitions of active instances. The meaning of its columns is as follows. *NewState* and *Timestamp* keep the new state and the time instant of the transition. *Symbol* stores the current symbol and the other columns keep the new values of global variables.

```
INSERT INTO Transitions
SELECT      TransitionTable.TargetState, TransitionTable.Symbol,
            Inbox.$t_s$, AIT.Var$_1$, ..., AIT.Var$_N$
FROM        Inbox, AIT, TransitionTable
WHERE       AIT.State = TransitionTable.SourceState
        AND Inbox.$t_s$ - AIT.Timestamp <= $Within$
        AND (     (TransitionTable.Symbol = "a" AND $\varphi_a$)
              OR (TransitionTable.Symbol = "b" AND $\varphi_b$)
                         ...
              OR (TransitionTable.Symbol = "z" AND $\varphi_z$) )
```

Listing 9.1: Computing automaton transitions on new events via SQL

Listing 9.1 shows the query that is executed for a pattern matching EPA when a new event arrives at the inbox of its input event stream. The query joins the new event, the AIT and the transition table of the automaton (`TransistionTable`) on `AIT.State = TransitionTable.SourceState`. This creates for each active instance all possible transitions. The query also checks whether an active instance has expired (`Inbox.`t_s `- AIT.Timestamp <=` *Within*). Lastly, the corresponding condition φ_x belonging to the symbol `x` of a possible transition must be fulfilled. Symbol conditions can be checked here, because the new event as well as the values of all global variables are available. The result set of the query includes all successful transitions and is saved in the table `Transitions`. This table contains now active instances with potentially outdated values of global variables. Therefore, a SQL `UPDATE` statement sets all affected global variables to the new values using the event in the inbox. Then, the content of `Transitions` is moved into the BT and an output event is created for each active instance that reached a final state.

9.3 Evaluation

We evaluated the JEPC bridge to JDBC on a machine with an i7-2600 CPU, 8 GiB main memory and an extra magnetic disk (WD1002FAEX) which was exclusively available for our experiments. JEPC was executed in a HotSpot JVM (1.2.0_13) and used the open source database systems H2 (1.3.172) [H2] and PostgreSQL (9.2.4) [Pos]. Connections to PostgreSQL were established through a JDBC type 4 driver. The test events had three randomly chosen 32-bit integer numbers as payload (a, b and c). Thus, every event had a size of 28 bytes (including two timestamps). All processed event streams contained exactly 10 test events per time instant.

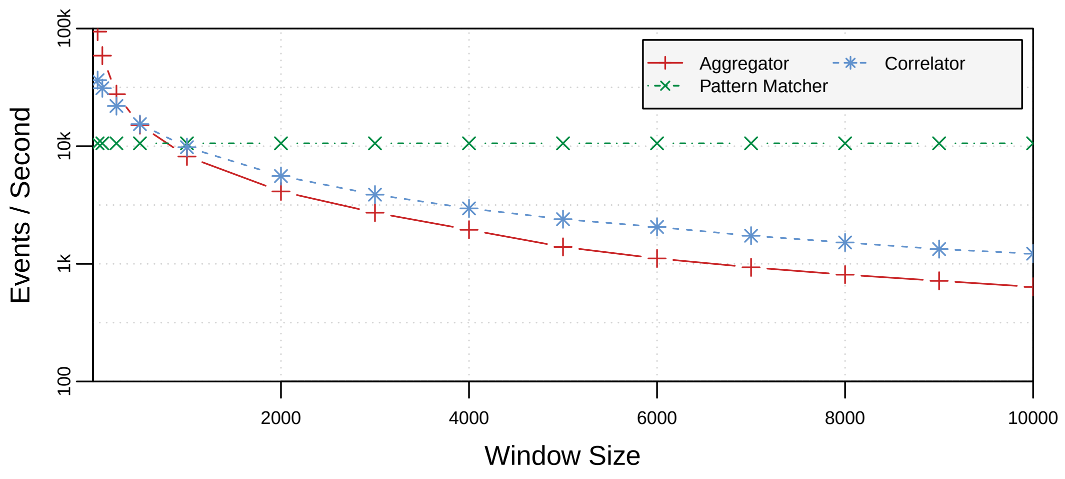


Figure 9.4: Performance of basic EPAs using H2

At first, we measured the maximum event throughput of each basic EPA by running a single instance of it. The filter EPA checked the attribute a for equality with a fixed number, the aggregation EPA computed the average value of the attribute a within a time-based sliding window, and the correlation EPA joined two test event streams E_1 and E_2 on $E_1.a = E_2.b$. In front of the correlation EPA, we applied equally-sized time-based sliding windows to the input streams. The pattern matching EPA was configured to detect three increasing numbers in a row for the attribute a. In this experiment, we tested different sizes of the time windows. Because there were ten events per time instant, the total number of events within each time window was ten times its size. For example, a time window of size 10,000 time instants contained always 100,000 events. Figure 9.4 shows the maximum throughputs using H2. As expected, the absolute numbers were noticeably lower in comparison to the maximum throughputs of SPEs. Note that the y-axis has a logarithmic scale, because the throughputs decreased significantly as the window sizes increased. However, the pattern matching EPA was not affected by the size of its window. This was because an event sequence with a fixed length of three events was searched via an automaton. The stateless filter EPA achieved a maximum throughput of about 135,000 events per second.

After we exchanged H2 for PostgreSQL, we got quite different numbers. Figure 9.5 plots the results of the same experiment using PostreSQL. The filter EPA achieved a maximum throughput of about 4,100 events per second. While the EPAs scaled better with the window size in comparison to H2, the absolute numbers were lower. The database system H2 clearly outperformed PostgreSQL for the tested range of window

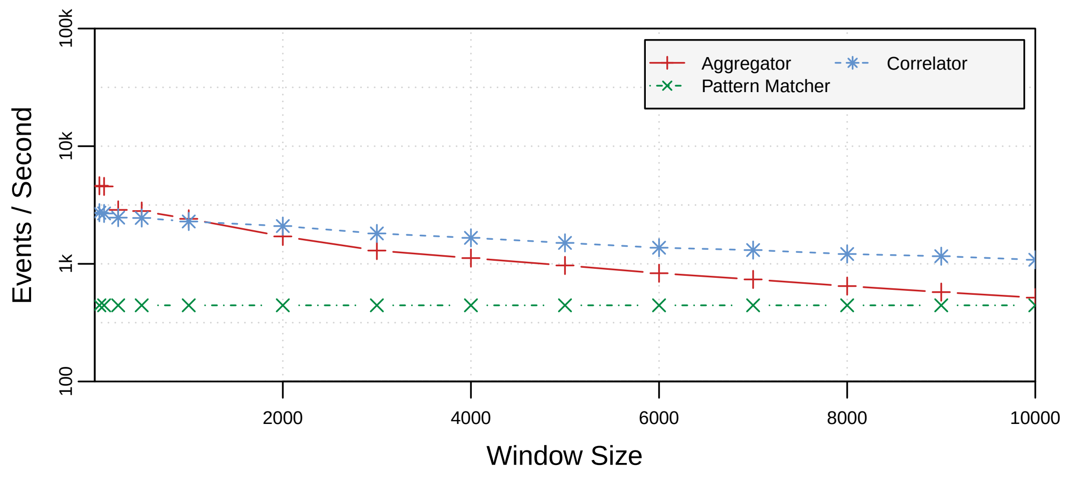


Figure 9.5: Performance of basic EPAs using PostgreSQL

sizes. However, this was not caused by the database systems themselves. PostgreSQL was simply limited by communication. Its JDBC driver communicates with an external server process over a network protocol. Note that our JEPC bridge to JDBC leads to a lot of communication. In particular, every single input event of an EPA is sent from JEPC to the database system followed by a series of SQL statements that perform the execution of EPAs. In addition, all output events must be sent back from the database system to JEPC. The database system H2 did not suffer from communication overhead, because it directly implements JDBC. But the communication costs became less influencing when the load in the database system increased (e.g., in case of EPAs having huge states). For example, the correlation EPA of this experiment processed 512 events per second for a window size of 30,000 using PostgreSQL, but it processed only 416 events per second using H2. As a rough guide, H2 should be preferred for query workloads which are not computing-intensive and a tenured database system such as PostgreSQL should be preferred in the other case.

Because real-world query workloads consist of more than a single EPA, we measured the overall performance for query workloads containing multiple EPAs and changed their sizes in the next experiment. In each tested query workload the types of EPAs were uniformly distributed. This means that a query workload containing x EPAs in total consisted of $x/4$ filter EPAs, $x/4$ aggregation EPAs, $x/4$ correlation EPAs and $x/4$ pattern matching EPAs. The EPAs were configured as in the previous experiment, but parameters were slightly varied so that no two EPAs were identical to each other. On average, the size of time windows was 100. Figure 9.6 shows the

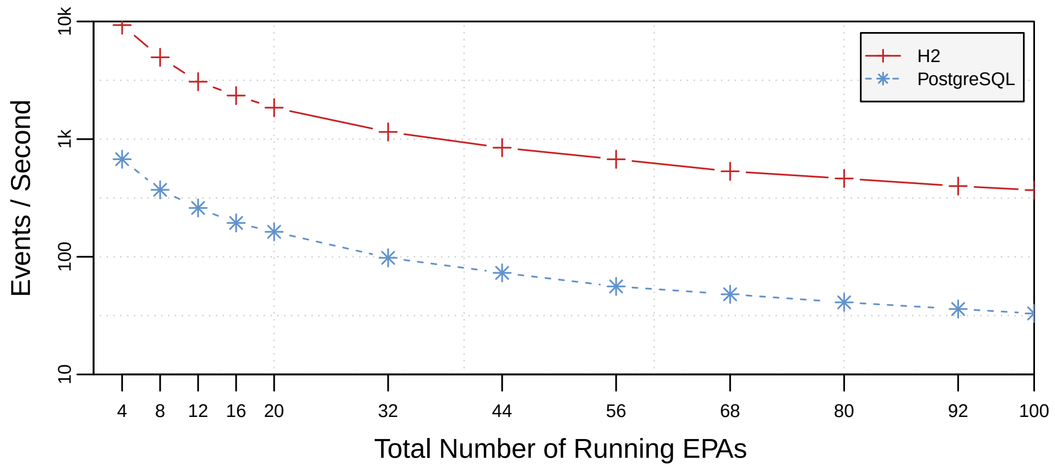


Figure 9.6: Performance of multiple running EPAs (small windows)

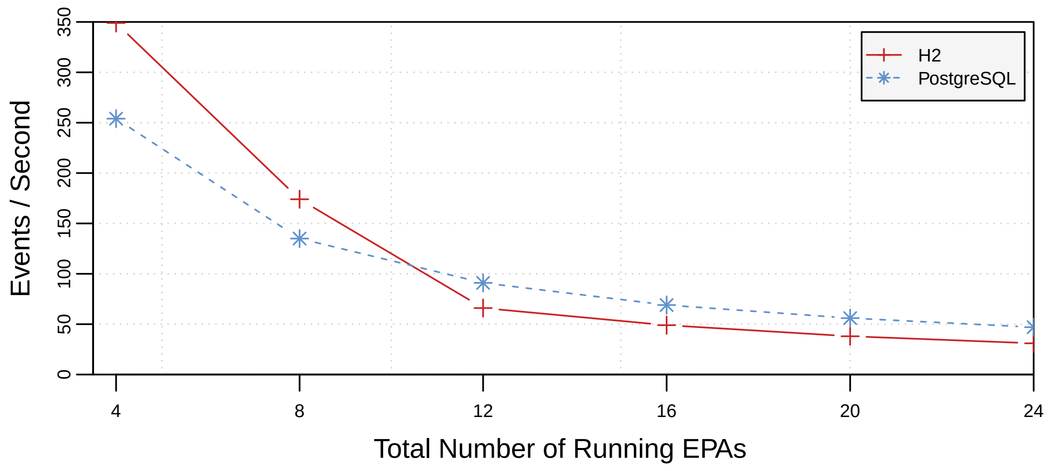


Figure 9.7: Performance of multiple running EPAs (large windows)

results for both database systems and different counts of running EPAs. The absolute numbers show that H2 could easily process hundreds of events per second. For many EP applications this performance is sufficient. Again, the performance got worse by multiple times after we exchanged H2 for PostgreSQL. As in the previous experiment, this indicates that the communication costs dominated the computation costs. When the computation costs became higher, then the difference between H2 and PostgreSQL reduced until PostgreSQL even outperformed H2. Figure 9.7 shows the results of the same experiment but for an average time window size of 15,000. In the case of 12 or more running EPAs, PostgreSQL was superior to H2.

Summary

This chapter presents a JEPC bridge that implements the JEPC specification on top of JDBC and is compatible with all standard database systems therefore. The states of EPAs are maintained in database tables on which standard SQL statements perform the execution of the basic EPAs including the pattern matching EPA. An experimental evaluation shows that a standard database system is able to run dozens of EPAs in EP applications dealing with hundreds of new events per second.

10

JEPC Bridge to Native EPA Implementations

Outline

10.1 Introduction

Besides JEPC bridges to different SPEs and standard database systems, we also developed a special JEPC bridge that has no target raw EP provider. Instead, this JEPC bridge, named *native EP provider*, is able to execute the basic EPAs by itself, because it contains implementations of them. The main advantages of such native implementations are that the JEPC specification can be directly implemented without the need for alignment of semantics and that EPAs can be executed using the best performing algorithms and implementations. Thus, it can be guaranteed that the native EP provider achieves high performance in every situation.

The following two chapters present our native implementations of the filter and aggregation EPAs in detail. Filtering is an important task in EP and extensively used in real-world applications to reduce enormous event streams to only the relevant events. Therefore, EPNs start with filter EPAs very often. The native EP provider indexes large sets of JEPC queries on basis of the BE-tree [SJ11, SJ13] which is currently the best performing query index. We extend the original BE-tree by several optimizations that result in significant performance improvements in EP applications.

Aggregation is an integral part of any kind of data analytics and widely used in EP applications therefore. The only proposed algorithm that implements the semantics of JEPC [Krä07, KS09] has linear time complexity. In challenging applications, it has unsatisfactorily performance therefore. We present a novel algorithm that has logarithmic time complexity and achieves a significantly better performance.

The native EP provider should not be the only direct implementation of the JEPC specification. In the future, we plan to develop parallel native implementations of the basic EPAs (intra-operator parallelization) that fully utilize graphics cards as well as modern multi-core and many-core CPUs.

10.2 Outline

Chapter 11 presents our improved version of the BE-tree that implements the filter EPA in the native EP provider. In Chapter 12, we introduce a faster algorithm for sliding-window aggregation over event streams with time-interval semantics that implements the aggregation EPA in the native EP provider.

11

An Efficient Index for Large Sets of Continuous Queries

Outline

11.1 Introduction

In database systems, index structures such as B-trees [BM72] and R-trees [Gut84] (to name the most commonly used index structures in today's database systems) are used to index large sets of persistent data items in order to speed up query execution. The general idea of every index structure is to look up all relevant data items instead of naïvely processing all data items including every irrelevant data item. In many cases, only few data items are relevant for determining the result of a query. Then, index structures can reduce the time needed for query execution by several orders of magnitude [GUW08]. Additionally, most index structures perform well even for very large databases. For example, the size of the web search engine index of Google increased from about 25 million unique webpages in 1998 to about one trillion unique webpages in 2008 [Jes]. However, the web search engine of Google was still able to look up all relevant webpages for a given set of keywords within a fraction of a second.

Because the roles of data items and queries are interchanged in the stream processing paradigm, queries are persistent and, thus, can be indexed. The problem of indexing CQs via their filter conditions has gained much attention in the past, because filter conditions specify which data items are relevant (analogous to filter conditions of ad-hoc queries in database systems) and have a wide application in many types of CQs. Therefore, the problem of indexing CQs reduces to the problem of indexing Boolean expressions and their atoms, the predicates. Filter conditions are commonly used not only in systems for event processing, but also in systems for publish/subscribe, information filtering, data stream management and many others. For instance, subscriptions can be indexed in publish/subscribe systems and rules can be indexed in rule-based intrusion detection systems (IDS).

While database indexes return for a given ad-hoc query a set of data items, query indexes return for a given new data item a set of continuous queries that must be executed for the new data item. For example, assume that there are eight continuous queries running as shown in the lower part of Figure 11.1. Furthermore, assume that each of the continuous queries processes only data items having a certain content (e.g., the *price* must be between 10 and 20 and the *color* must be "yellow"). Without an index, all eight continuous queries must be executed for every single data item that comes in (analogous to full table scans in databases) [Agu99, YG94]. Just as full table scans that scale linearly with the total number of data items, the naïve execution of all CQs scales linearly with the total number of CQs. For instance, if the total number of CQs doubles, then the effort for processing a single data item doubles too (assuming that

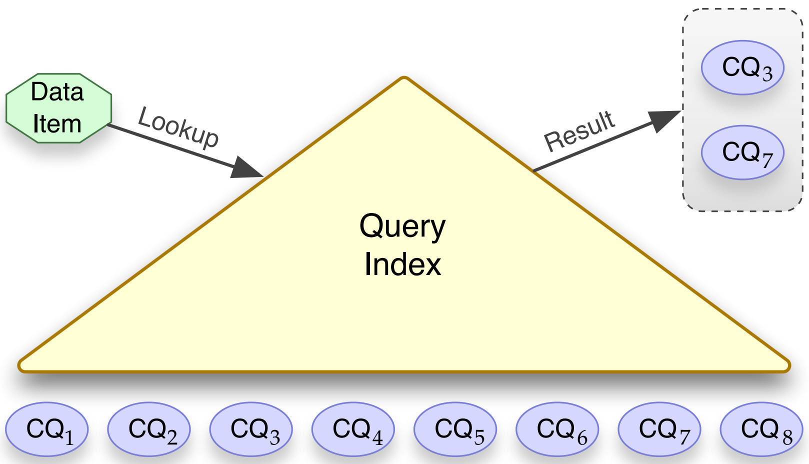


Figure 11.1: Query index

the execution costs are the same for each CQ). In the presence of a query index, a lookup can be performed for every new data item in order to obtain all continuous queries that must be executed. For example, the illustrated data item is relevant only for the queries CQ_3 and CQ_7 in Figure 11.1. Thus, it is correct and less costly to execute only CQ_3 and CQ_7 instead of all continuous queries. Similar to their counterparts in database systems, also query indexes scale well in general and are essential for applications dealing with large sets of continuous queries.

The BE-tree [SJ11, SJ13] is a recently presented dynamic query index being superior to all static and dynamic query indexes proposed so far. It addresses explicitly high-dimensional and discrete spaces. Additionally, it is intended to be used in highly dynamic application domains in which index updates occur as often as or even more often than new data items. This is true for many application domains such as publish/subscribe. For example, an e-commerce system with products as data items can allow customers to keep informed of new products according to their interests via user-defined subscriptions [SJ11, SJ13, ZCT14]. In this case, new data items are rare, subscriptions change very often (every new customer leads to a set of new subscriptions and existing customers constantly update their subscriptions), data items have `NULL` values for most attributes [CBN07], and the cardinalities of attribute domains are low. For applications having all these properties the BE-tree is perfectly suited. However, none of the properties is true for event processing applications. In this chapter, we focus on improving the BE-tree for a certain class of stream processing applications. Our modified BE-tree, named *BE*$^+$*-tree*, is optimized for EP applications and all stream processing applications that have properties similar to EP.

11.2 Preliminaries

Almost all query indexes are for Boolean expressions in conjunctive form. A Boolean expression in conjunctive form consists of any number of predicates that are all connected by the logical $\wedge$ operator. Furthermore, each predicate must refer to exactly one data attribute and specify all relevant values for it (all data attributes span the space, i.e., each data attribute corresponds to one dimension). In other words, every predicate must restrict the domain of exactly one data attribute to a subset of values in the domain. Note that the described form of predicates is special, because one operand must be a data attribute and the other must be a fixed subset of values in the attribute domain. Obviously, predicates with only data attributes as operands (e.g., $width = height$, with *height* and *with* being numeric data attributes) cannot be supported by an index and must be evaluated for each data item individually.

Definition 4 (Indexable Predicate)**.** *An* indexable predicate $p_{attr,\mathbb{V}}(x)$ *is a Boolean-valued function having a fixed data attribute attr and a fixed set of constant values* $\mathbb{V}$ *being the parameters of the predicate. For every value x in the domain of attr, an indexable predicate evaluates to either* `TRUE` *or* `FALSE`*.*

Definition 4 requires a predicate to restrict one data attribute to a set of constant values for being indexable. Note that JEPC natively supports the indexable predicates $<_{attr,\{v\}}$, $\leq_{attr,\{v\}}$, $=_{attr,\{v\}}$, $\neq_{attr,\{v\}}$, $\geq_{attr,\{v\}}$ and $>_{attr,\{v\}}$.

Definition 5 (Indexable Boolean Expression)**.** *An* indexable Boolean expression *is a Boolean expression having the form* $\bigwedge_{i=1,2,\ldots,n}\left(p^i\right)$*. Every predicate* p^i *is an indexable predicate and for each data attribute there is at most one predicate.*

According to Definition 5, a Boolean expression is indexable only if all its predicates are indexable and connected by logical $\wedge$ operators. Furthermore, every data attribute may not be restricted by more than one predicate.

Complex Boolean expressions being not in conjunctive form can be decomposed into a set of Boolean expressions being in conjunctive form (see Section 11.6.2). Through rewriting, a conjunctive subexpression containing multiple predicates for a data attribute can often be reduced (e.g., $<_{price,\{25\}} \wedge <_{price,\{50\}}$ reduces to $<_{price,\{25\}}$). The additional rewriting rules presented in the following are generally applicable and essential in the case of JEPC that only supports a limited set of predicates natively. They replace subexpressions used to express one of the unsupported predicates *between*, $\in$ or $\notin$ by the corresponding unsupported predicate. Our implementation automatically applies all these rewriting rules before indexing.

Unsupported predicate	*Substitution expression*
$between_{attr,\{v_1,v_2\}}$	$\geq_{attr,\{min(v_1,v_2)\}} \wedge \leq_{attr,\{max(v_1,v_2)\}}$
$\in_{attr,\{v_1,v_2,\ldots,v_n\}}$	$\bigvee_{i=1,2,\ldots,n} (=_{attr,\{v_i\}})$
$\notin_{attr,\{v_1,v_2,\ldots,v_n\}}$	$\bigwedge_{i=1,2,\ldots,n} (\neq_{attr,\{v_i\}})$

Table 11.1: Substitution of unsupported predicates

The predicates *between*, $\in$ and $\notin$ are supported by our native EP provider but not by JEPC. However, they can be substituted by the expressions shown in the second column of Table 11.1. Before indexing, the native EP provider searches for substitution expressions in a Boolean expression and exchanges them for the corresponding predicates that are not part of the JEPC specification. For example, the valid but non-indexable Boolean expression $<_{price,\{25\}} \wedge \geq_{weight,\{100\}} \wedge \leq_{weight,\{200\}} \wedge (=_{size,\{32\}} \vee =_{size,\{42\}} \vee =_{size,\{55\}})$ becomes the indexable Boolean expression $<_{price,\{25\}} \wedge between_{weight,\{100,200\}} \wedge \in_{size,\{32,42,55\}}$ after applying the rules of Table 11.1.

11.3 BE-Tree

The BE-tree is a tree data structure for indexing a set of indexable Boolean expressions. Its internal structure consists of different types of nodes. This makes the BE-tree more complex in comparison to tree data structures that have a homogenous internal structure such as, for example, the B-tree and the R-tree.

11.3.1 Predicate Mapping

For the purpose of indexing a set of indexable predicates that refer to the same data attribute in a uniform way (i.e., independent of the types of predicates), the BE-tree maps every predicate $p_{attr,\mathbb{V}}(x)$ to a single one-dimensional interval that covers all values x in the domain of *attr* for which the predicate evaluates to `TRUE`. Table 11.2 shows for each type of indexable predicate its corresponding mapping to an interval (let v_{min} be the minimum value in the attribute domain and v_{max} be the maximum value in the attribute domain). Note that a resulting interval may also cover values for which the corresponding predicate evaluates to `FALSE`. In some cases, even the entire attribute domain may be covered (e.g., $\neq_{attr,\{v\}}$ and $\notin_{attr,\{v_1,v_2,\ldots,v_n\}}$). The mapping of indexable predicates to intervals leads to the advantage that all types of indexable predicates can be indexed in the same way and together in a single data structure.

Indexable Predicate	*Interval*
$<_{attr,\{v\}}$	$[v_{min} : v - 1]$
$\leq_{attr,\{v\}}$	$[v_{min} : v]$
$=_{attr,\{v\}}$	$[v : v]$
$\neq_{attr,\{v\}}$	$[v_{min} : v_{max}]$
$\geq_{attr,\{v\}}$	$[v : v_{max}]$
$>_{attr,\{v\}}$	$[v + 1 : v_{max}]$
$between_{attr,\{v_1,v_2\}}$	$[\min\{v_1, v_2\} : \max\{v_1, v_2\}]$
$\in_{attr,\{v_1,v_2,\ldots,v_n\}}$	$[\min\{v_1, v_2, \ldots, v_n\} : \max\{v_1, v_2, \ldots, v_n\}]$
$\notin_{attr,\{v_1,v_2,\ldots,v_n\}}$	$[v_{min} : v_{max}]$

Table 11.2: Mapping of indexable predicates to intervals

11.3.2 Structure of BE-Tree

A BE-tree is an unbalanced *n*-ary tree data structure that is supposed to reside completely and permanently in main memory. All indexed Boolean expressions are stored in the leaf nodes. There are two different types of internal nodes that route to the leaf nodes. So-called *partition nodes* are used to partition the space. Per partition, all indexed Boolean expressions are clustered via so-called *cluster nodes*. In a BE-tree, partition nodes and cluster nodes alternate along a path. In particular, every partition node is followed by one or more cluster nodes and every cluster node is followed by any number of partition nodes. Along every path, no two partition nodes refer to the same data attribute. Thus, every path can have at most as many partition nodes as there are data attributes. As a consequence, the maximum possible height of a BE-tree is limited by the total number of data attributes. All partition nodes and all cluster nodes that have the same parent node are maintained in a directory.

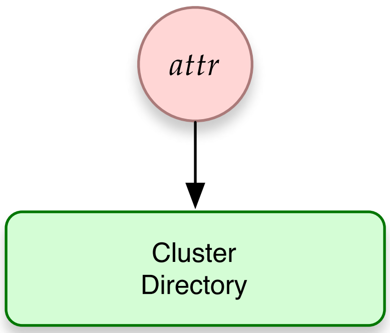


Figure 11.2: Partition node

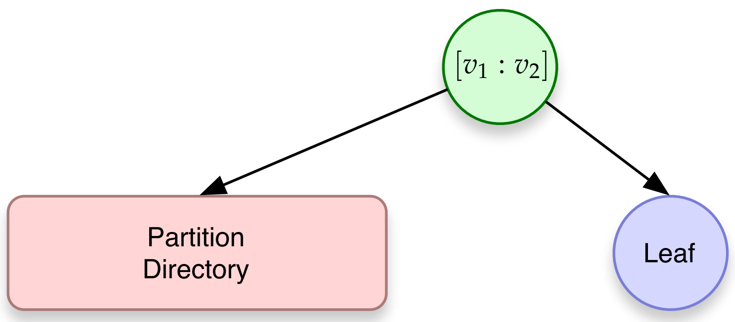


Figure 11.3: Cluster node

A partition node (*p-node* for short) is associated with exactly one data attribute (i.e., one dimension of space). All Boolean expressions stored in the subtree of a p-node have an indexable predicate on the associated data attribute. Every p-node is followed by at least one cluster node. The cluster nodes of a p-node are organized in a single cluster directory that is the only child node of a p-node. Figure 11.2 illustrates a p-node. The shown p-node is associated with the data attribute *attr* and has a pointer to its cluster directory that manages all subsequent cluster nodes.

Each cluster node (*c-node* for short) in a BE-tree (except the root node that is always a c-node) has an assigned one-dimensional interval and is associated with the data attribute of its parent p-node. The interval of a c-node represents a range of values in the domain of the associated data attribute. All Boolean expressions stored in the subtree of a c-node have an indexable predicate which restricts the associated attribute domain to a set of values that is completely covered by the interval of the c-node. Every c-node is followed by exactly one leaf node as well as any number of p-nodes that are organized in a single partition directory. Figure 11.3 illustrates a c-node. It shows a c-node with its assigned interval $[v_1 : v_2]$ covering all values between v_1 and v_2. Also, the subsequent leaf node and partition directory are shown.

Every path from the root node of a BE-tree to a leaf node is always an alternating sequence of p-nodes and c-nodes. Leaf nodes (*l-nodes* for short) in a BE-tree store the indexed Boolean expression that belong to the preceding partitions and clusters. A stored Boolean expression is directly associated with the query it belongs to. Each l-node in a BE-tree has the following property.

Definition 6 (Leaf Node Property)**.** *Every leaf node in a BE-tree stores at most max_{cap} Boolean expressions in total. Otherwise, a leaf node is considered to be overflowing. A leaf node is exceptionally allowed to overflow permanently if and only if there is no further clustering possible and there is no or no suitable data attribute for partitioning available.*

The maximum capacity max_{cap} of l-nodes is a constant and user-defined parameter of the BE-tree. Overflowing l-nodes must be split by further clustering or partitioning in order to eliminate the overflow. There are two exceptional cases where a (reasonable) partitioning of an overflowing l-node that cannot be further clustered may not be possible. First, all data attributes have already been used for partitioning on the path from the root node to the overflowing l-node. Second, there are data attributes left for partitioning, but only those that are rarely used in the stored Boolean expressions (i.e., all available data attributes have low popularity). In the first case further partitioning is simply not possible and in the second case further partitioning would not improve the matching performance of the BE-tree.

A partition directory (*p-directory* for short) is just a map for efficiently looking up p-nodes. It maps each data attribute that is used for partitioning the neighboring l-node to its corresponding p-node. Figure 11.4 illustrates a p-directory.

Every cluster directory (*c-directory* for short) is a grid-based index for one-dimensional intervals. Each cell of a c-directory covers a certain range of values of the associated data attribute and represents a cluster. A cell also points to a c-node that keeps all Boolean expressions of the cluster. Figure 11.5 shows a c-directory in detail. The root cell covers always the entire domain of the associated data attribute (see Definition 7). In a c-directory, the next level is created by cutting one or more cells at the current level into halves. This process can be recursively repeated until the cells cannot be further split. Cells that cannot be split anymore are called *atomic cells*. The grid of a c-directory is expanded dynamically on demand. All cells of a grid that do not have child cells are called *leaf cells* independent of whether they are atomic cells. C-directories are used to index the predicates of indexable Boolean expressions and allow for efficient lookup operations.

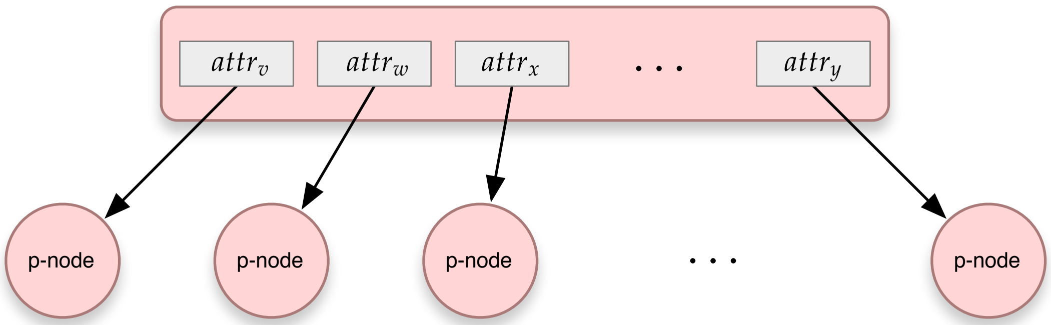
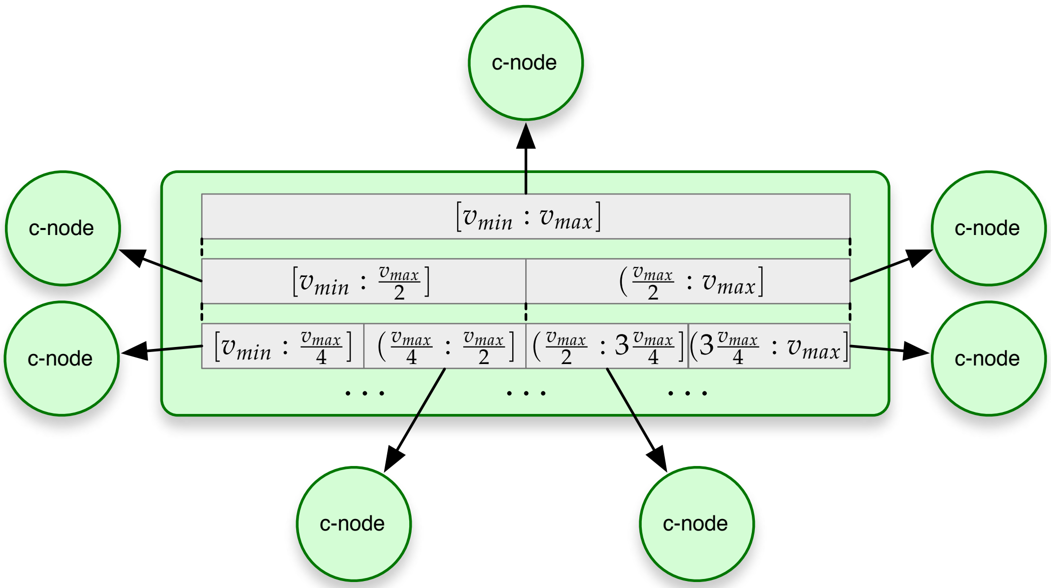


Figure 11.4: Partition directory



c-node
$[v_{min} : v_{max}]$
$[v_{min} : \frac{v_{max}}{2}]$
$(\frac{v_{max}}{2} : v_{max}]$
$[v_{min} : \frac{v_{max}}{4}]$
$(\frac{v_{max}}{4} : \frac{v_{max}}{2}]$
$(\frac{v_{max}}{2} : 3\frac{v_{max}}{4}]$
$(3\frac{v_{max}}{4} : v_{max}]$
. . .

Figure 11.5: Cluster directory

Definition 7 (C-Directory Property)**.** *The root cell of every c-directory in a BE-tree covers the entire domain of its associated data attribute. This ensures the support of insertions of indexable predicates at any time.*

A new c-directory consists only of an empty root cell that covers the entire domain. Every interval being inserted must be always put into the smallest cell that encloses it (see Definition 8). By inserting an additional interval, a cell of a c-directory may overflow (i.e., the l-node of the corresponding c-node overflows). An overflowing cell that is a non-atomic leaf cell can be cut into halves to allow for pushing down some of its stored intervals. Note that the grid of a c-directory is unbalanced in general.

Definition 8 (C-Directory Insertion Property)**.** *An interval must be inserted into the smallest cell of a c-directory that encloses it. This is the cell that encloses the interval and has no child cells at all or no child cell that encloses the interval.*

Definition 8 leads to an algorithm to insert Boolean expressions into c-directories. Algorithm 1 shows the details. The algorithm requires a Boolean expression being inserted and a cell of a c-directory. Initially, it is called with the root cell. A Boolean expression being inserted into a c-directory must have an indexable predicate on the associated data attribute. Then, that predicate is mapped to an interval according to Table 11.2. The resulting interval represents the Boolean expression and is used for insertion. Thus, the method ENCLOSES checks whether a cell encloses the corresponding interval that represents the Boolean expression. The algorithm traverses recursively the c-directory until the correct cell has been found.

Lookups are performed similarly in c-directories as Algorithm 2 shows. The algorithm needs a data item for which all qualifying Boolean expressions are looked up, a cell of a c-directory and a set to which all qualifying Boolean expressions are added. Initially, the algorithm is called with the root cell of a c-directory. The data item is represented by its value of the data attribute the c-directory is associated with. Then, the algorithm recursively traverses the c-directory and visits all cells that cover the data item. The cluster node of each visited cell qualifies and must be checked further (the algorithm MATCHBETREE is presented later in this chapter). Note that lookups in c-directories are limited to a single path.

Algorithm 1: INSERTCDIRECTORY(*be*, *cell*)

```
Input: Boolean Expression: be, C-Directory Cell: cell
Output: C-Node: cNode
1 if cell.leftChild $\neq$ NULL then
2     if cell.leftChild.ENCLOSES(be) then
3         C-Node cNode $\leftarrow$ INSERTCDIRECTORY(be, cell.leftChild);
4         return cNode;
5 if cell.rightChild $\neq$ NULL then
6     if cell.rightChild.ENCLOSES(be) then
7         C-Node cNode $\leftarrow$ INSERTCDIRECTORY(be, cell.rightChild);
8         return cNode;
9 return cell.cNode;
```

Algorithm 2: MATCHCDIRECTORY(*dataItem*, *cell*, *resultSet*)

```
Input: Data Item: dataItem, C-Directory Cell: cell, Set of Boolean Expressions: resultSet
1 if cell.ENCLOSES(dataItem) then
2     MATCHBETREE(dataItem, cell.cNode, resultSet);
3     if cell.leftChild $\neq$ NULL then
4         MATCHCDIRECTORY(dataItem, cell.leftChild, resultSet);
5     if cell.rightChild $\neq$ NULL then
6         MATCHCDIRECTORY(dataItem, cell.rightChild, resultSet);
```

11.3.3 Local Reorganization

During insertion of indexable Boolean expressions, l-nodes might overflow and require the BE-tree to be reorganized. The reorganization of a BE-tree comprises two phases that are executed one another but as an atomic unit. In the first phase the space is partitioned and in the second phase the space is clustered.

11.3.3.1 Space Partitioning

Space partitioning is performed for a single c-node having an overflowing l-node. Algorithm 3 shows the steps of space partitioning in detail. To resolve the overflow, a data attribute for splitting the overflowing l-node is selected and all stored Boolean expressions which have a predicate on that data attribute (RESTRICTS) are moved into a new partition. This procedure is repeatedly performed until the l-node is no longer overflowing or no further partitioning is possible. The algorithm GETPARTITIONINGATTRIBUTE returns for a l-node (thus, a set of Boolean expressions) the best data attribute for partitioning. Therefore, each available data attribute is scored and the available data attribute with the highest score is selected. It might be possible that there is no suitable data attribute available for partitioning. Then, GETPARTITIONINGATTRIBUTE returns NULL. In this case, the space partitioning stops and the l-node remains overflowing (see Definition 6). The exact meanings of the terms *available* and *suitable* are given in Definition 9.

Algorithm 3: SPACEPARTITIONING(*cNode*, *path*)

```
Input: C-Node: cNode, Set of Attributes: path
while SIZEOF(cNode.lNode) > $max_{cap}$ do
    Attribute attr ← GETPARTITIONINGATTRIBUTE(cNode.lNode, path);
    if attr = NULL then
        return;
    P-Node pNode ← cNode.pDirectory.NEWPARTITION(attr);
    for Boolean Expression be ∈ cNode.lNode do
        if be.RESTRICTS(attr) then
            C-Node targetNode ← INSERTCDIRECTORY(be, pNode.cDirectory.root)
            cNode.lNode.REMOVE(be);
            targetNode.lNode.ADD(be);
    SPACECLUSTERING(pNode.cDirectory.root, path ∪ {attr});
```

Definition 9 (Space Partitioning Property)**.** *A split attribute used for space partitioning must be a data attribute that is available and suitable. A data attribute is* available *if and only if it has not already been chosen as split attribute on the path from the root node to the overflowing l-node. This ensures that every data attribute occurs on every path in a BE-tree at most once. A data attribute is* suitable *if and only if there are at least $min_{support}$ Boolean expressions with a predicate on it in the overflowing l-node. This ensures that a new partition always starts with a certain amount of Boolean expressions.*

According to the space partitioning property, a data attribute that has already been used to partition the space cannot be used as split attribute again. This is, because all Boolean expressions in the overflowing l-node have an indexable predicate on it and, thus, all Boolean expressions would be moved into the new partition. Moreover, the space partitioning property ensures this way that infinite loops do not occur during space partitioning. A data attribute used for space partitioning must also have a minimum support among all Boolean expressions of the overflowing l-node. In particular, a split attribute must be restricted by at least $min_{support}$ Boolean expressions of the overflowing l-node. The parameter $min_{support}$ is constant and user-defined.

If there is a data attribute which can be used for space partitioning, Algorithm 3 creates a new p-node for that data attribute (line 5) and moves all Boolean expressions of the overflowing l-node with a predicate on the selected data attribute into the new partition (lines 6–9). This means, all qualifying Boolean expressions are inserted into the root cell (that is the only existing cell in new partitions) of the c-directory of the new partition. Then, space clustering is performed for the c-directory. This ensures not only that a potential overflow of the root cell is eliminated, but also that space partitioning is directly followed by space clustering.

Up to now, we explained the conditions a data attribute must fulfill for being considered as split attribute. In the following, we discuss the scoring of data attributes. Intuitively, the partitioning of a set of Boolean expressions by some data attribute is the better, the more often that data attribute is used by predicates of the Boolean expressions (popularity) and the more selective the corresponding predicates are (selectivity) [SJ11, SJ13, YG94]. Data attributes with high popularity and low selectivity can be used to identify large sets of Boolean expressions (i.e., continuous queries) that do not match a given data item with high probability. Therefore, data attributes with the highest popularity and lowest selectivity should be used for partitioning to optimize the matching performance of the BE-tree. For scoring data attributes with respect to popularity and selectivity, the original BE-tree uses a scoring function that takes into account the actual value distributions of data attributes. Thus, it is an online approach that requires monitoring at runtime. This online approach allows the BE-tree to be adaptive to changing data stream characteristics. In particular, the online scoring function monitors false positive evaluations in order to estimate the selectivity of attributes. We do not use the original online scoring function, because it requires monitoring of the input data stream that imposes overhead at runtime. Furthermore, the behavior of the online scoring function is hard to predict. For all these reasons, we developed a simpler offline scoring function for our BE$^+$-tree. To achieve comparability in benchmarks, we decided to use our simpler offline scoring function also for the original BE-tree. Our offline scoring function also takes into account popularity and selectivity of an attribute but estimates the selectivity via a heuristic as it is done often by query compilers of database systems [Sel79]. Algorithm 4 shows our offline scoring function in detail. It returns for a data attribute and a set of Boolean expressions the corresponding score. For each predicate on the given attribute contained in the set of Boolean expressions the score is increased. The constant value that is added depends on the selectivity which is determined by the relative selectivity of the operator of a predicate for uniformly distributed values. Because both popularity and selectivity influence the score, it is an appropriate function for scoring data attributes.

Algorithm 4: GETSCORE(*attr*, *booleanExpressions*)

```
Input: Attribute: attr, Set of Boolean Expressions: booleanExpressions
Output: Number: score
Number score ← 0;
for Boolean Expression be ∈ booleanExpressions do
    for Predicate predicate ∈ be do
        if predicate.attribute = attr then
            if predicate.operator ∈ {=} then
                score ← score + 10000;
            else if predicate.operator ∈ {between, ∈} then
                score ← score + 1000;
            else if predicate.operator ∈ {<, ≤, ≥, >} then
                score ← score + 10;
            else
                score ← score + 1;
return score;
```

11.3.3.2 Space Clustering

Space clustering is the second phase of local reorganization and handles an overflowing l-node by splitting its corresponding cluster (i.e., the c-directory cell of its parent c-node) and trying to move enough Boolean expressions into the new clusters. To ensure the strict execution order of the two reorganization phases, space clustering must always be performed first (see Definition 10). This is respected by the algorithms of the BE-tree. Algorithm 5 shows the details of the space clustering phase.

Definition 10 (Space Clustering Property)**.** *Space partitioning is considered for splitting an overflowing leaf node only if further clustering is not possible (i.e., the corresponding cell in the c-directory is atomic or has already been split) or further clustering cannot eliminate the overflow (i.e., not enough Boolean expressions can be moved into the new clusters). In other words, to eliminate the overflow of a leaf node, it must first be tried to split the overflowing leaf node by performing space clustering.*

Algorithm 5: SPACECLUSTERING(*cell*, *path*)

```
Input: C-Directory Cell: cell, Set of Attributes: path
if SIZEOF(cell.cNode.lNode) $\leq max_{cap}$ then
    return;
if (not ISLEAF(cell) or ISATOMIC(cell)) then
    SPACEPARTITIONING(cell.cNode, path);
else
    Endpoint middle $\leftarrow$ cell.GETMINVALUE()/2 + cell.GETMAXVALUE()/2;
    cell.leftChild $\leftarrow$ NEWCELL([cell.GETMINVALUE() : middle]);
    cell.rightChild $\leftarrow$ NEWCELL([middle + 1 : cell.GETMAXVALUE()]);
    for Boolean Expression be $\in$ cell.cNode.lNode do
        if cell.leftChild.ENCLOSES(be) then
            cell.leftChild.cNode.lNode.ADD(be);
            cell.cNode.lNode.REMOVE(be);
        if cell.rightChild.ENCLOSES(be) then
            cell.rightChild.cNode.lNode.ADD(be);
            cell.cNode.lNode.REMOVE(be);
    SPACEPARTITIONING(cell.cNode, path);
    SPACECLUSTERING(cell.leftChild, path);
    SPACECLUSTERING(cell.rightChild, path);
```

Space clustering can be performed only for overflowing cells (lines 1–2) that are neither atomic (cannot be split any further) nor already have been split. In all other cases, the algorithm switches to space partitioning (lines 3–4). If space clustering can be performed, it splits the overflowing cell (lines 6–8). Every Boolean expression of the overflowing cell that is enclosed by one of the new child cells is pushed down to the next level of the c-directory (lines 9–15). As a result, the Boolean expressions of the overflowing cell are distributed across the original cell (all Boolean expressions that cannot be moved remain in the original cell) and its two child cells. Then, the reorganization switches back to space partitioning for the original cell while space clustering is continued for the new child cells (lines 16–18).

11.3.4 Insert, Delete and Match

The basic operations of a query index are insertion and deletion of indexable Boolean expressions as well as performing lookups for data items. In this section, we present the insertion and lookup operations of the original BE-tree. A Boolean expression is deleted from a BE-tree by simply removing it from its l-node.

11.3.4.1 Insert

New BE-trees start empty consisting of the root node only. The root node is always a c-node that is not associated with a data attribute and a range of values. Initially, it has an empty l-node and an empty p-directory. Boolean expressions being inserted are pushed down as long as there are matching partitions. They are finally stored in the l-node of the last c-node that does not have a matching partition in its p-directory. Because l-nodes can store at most max_{cap} Boolean expressions, l-nodes may overflow eventually and trigger the creation of new clusters and partitions.

An indexable Boolean expressions *be* is inserted into a BE-tree via Algorithm 6 that is initially called with the root node (parameter *cNode*). At first, the best partition in the p-directory is determined. For each data attribute on that the indexable Boolean expression has a predicate and that has not been used on the path so far, the partition is looked up. Because there can be multiple partitions into which the indexable Boolean expression could be inserted, the partition with the highest score is selected (lines 1–9). Note that there might be no partition at all. In this case, the indexable Boolean expression is stored in the l-node of the current c-node (lines 10–15). In all other cases, the insertion of the indexable Boolean expression is continued for the corresponding cluster of the selected partition (lines 17–19).

Algorithm 6: InsertBETree(*be*, *cNode*, *cDirectory*, *path*)

```
Input: Boolean Expression: be, C-Node: cNode, C-Directory: cDirectory, Set of
       Attributes: path
1  P-Node bestPartition ← NULL;
2  Number bestScore ← $-\infty$;
3  for Predicate predicate $\in$ be do
4      if predicate.attribute $\notin$ path then
5          P-Node pNode ← cNode.pDirectory.LookUp(predicate.attribute);
6          if pNode $\neq$ NULL then
7              if pNode.score > bestScore then
8                  bestScore ← pNode.score;
9                  bestPartition ← pNode;
10 if bestPartition = NULL then
11     cNode.lNode.Add(f);
12     if cDirectory $\neq$ NULL then
13         SpaceClustering(cDirectory.root, path);
14     else
15         SpacePartitioning(cNode, path);
16 else
17     C-Node cNodeInsert ← InsertCDirectory(be, bestPartition.cDirectory.root);
18     InsertBETree(be, cNodeInsert, bestPartition.cDirectory,
       path $\cup$ {bestPartition.attribute});
19     UpdateScore(bestPartition);
```

11.3.4.2 Match

Data items can be interpreted as Boolean expressions in conjunctive form. Each element $(attribute, value)$ of a data item maps to an indexable equality predicate $=_{attribute,\{value\}}$ with the attribute name and the attribute value of the data item as operands. Therefore, data items are indexable Boolean expressions and the lookup operation shown in Algorithm 7 is similar to the insertion operation. The algorithm starts initially at the root node (parameter *cNode*). While a data item traverses through a BE-tree, every matching partition of visited p-directories must be searched for results. Thus, this operation is not limited to a single path in a BE-tree. Recall that within every c-directory the lookup operation is limited to a single path and all cells on that path must be visited (see Algorithm 2).

Algorithm 7: MATCHBETREE(*dataItem*, *cNode*, *resultSet*)

```
Input: Data Item: $dataItem$, C-Node: $cNode$, Set of Boolean Expressions: $resultSet$
1 $resultSet$.ADDALL($cNode.lNode$.LOOKUP($dataItem$));
2 for Predicate $predicate \in dataItem$ do
3     P-Node $pNode \leftarrow cNode.pDirectory$.LOOKUP($predicate.attribute$);
4     if $pNode \neq$ NULL then
5         MATCHCDIRECTORY($dataItem$, $pNode.cDirectory.root$, $resultSet$)
```

11.4 Shortcomings in Event Processing Applications

In this section, we study the use of the BE-tree in event processing applications. At first, we present an example application that is used throughout this section and in our evaluation. The example application consists of an event stream with 100 attributes $A_0, A_1, \ldots, A_{99}$ each of type 32-bit integer number. Note that the total number of attributes is high for an EP application but quite low for the BE-tree that has been designed for spaces with hundreds of dimensions. Also note that the attribute domains have large cardinalities (2^{32} to be precise). This also conflicts some assumptions made by the BE-tree. In the evaluation of the original BE-tree (see [SJ11, SJ13]), attribute domains had cardinalities of at most 50 in all experiments except one. The only experiment with larger domain cardinalities tested attribute domains with cardinalities up to 150,000. On basis of the example event stream, we define a list of 10,000 continuous queries $query_i$ ($i = 0, \ldots, 9999$) having the following filter conditions:

$$query_i := between_{attr_1(i),\{v_1,v_2\}} \wedge between_{attr_2(i),\{v_3,v_4\}}$$

Each filter condition is in conjunctive form and consists of two indexable *between* predicates. A *between* predicate allows to exactly control the size and position of the mapped one-dimensional interval being indexed in c-directories (see Table 11.2). The attribute of the first *between* predicate is set as follows:

$$attr_1(i) := A_j, \text{ with } j = (i \bmod 3)$$

In other words, the filter conditions iterate in a round-robin fashion over the first three attributes A_0, A_1 and A_2 of the event stream. The attribute of the second *between* predicate is set as follows:

$$attr_2(i) := A_j, \text{ with } j = 3 + (\lfloor i/100 \rfloor \bmod 99)$$

The filter conditions of the first 100 queries have A_3 as attribute for the second predicate, the filter conditions of the next 100 queries have A_4, the filter conditions of the next 100 queries have A_5, and so on. The values of v_1 and v_3 are uniformly drawn from the range $[0 : 1,000,000]$ and the values of v_2 and v_4 are set to $v_1 + 1,000,000$ and $v_3 + 1,000,000$ respectively. As a result, every *between* predicate maps to an interval that has a size of 1,000,000 and is completely in $[0 : 2,000,000]$.

11.4.1 Globally Non-Optimal Space Partitioning

The BE-tree is a dynamic index structure that can be updated at runtime. It makes no assumptions about frequencies of data items and index updates. Thus, index updates with high frequency must be supported. The BE-tree achieves this by fast local reorganization. Space clustering affects only single partitions and is deterministic (see Definition 8). Consequently, local reorganization is sufficient to achieve an optimal space clustering of each partition. In contrast, space partitioning affects the entire BE-tree and is not deterministic. But to achieve fast adaption, space partitioning is also performed only locally by splitting overflowing leaf nodes. A globally optimal space partitioning (according to the scoring function) is not achieved by locally optimal space partitioning in general. In fact, the global space partitioning depends on the insertion order of indexable Boolean expressions. To handle this problem, the original BE-tree proposes a self-adjustment technique. In this approach, leaf nodes are continuously scored. If the score of a leaf node drops below some threshold, then a leaf node is entirely or partly deleted and all affected indexable Boolean expressions are reinserted. The problems of this approach are as follows. First, it requires additional monitoring at runtime (insertions, deletions and matches of each l-node must be monitored in detail) which contradicts the strict performance requirements of EP. Second, there are no guarantees whether and when a globally optimal partitioning is achieved. Third, the reinsertion might happen in a suboptimal insertion order again.

The problem of the space partitioning of the BE-tree comes to light when the queries $query_i$ are inserted into a BE-tree ordered by the index i. Let max_{cap} be set to three and $min_{support}$ be set to one. The first four filter conditions being inserted are:

$$query_0 = between_{A_0,\{v_1,v_2\}} \wedge between_{A_3,\{v_3,v_4\}}$$

$$query_1 = between_{A_1,\{\hat{v}_1,\hat{v}_2\}} \wedge between_{A_3,\{\hat{v}_3,\hat{v}_4\}}$$

$$query_2 = between_{A_2,\{\bar{v}_1,\bar{v}_2\}} \wedge between_{A_3,\{\bar{v}_3,\bar{v}_4\}}$$

$$query_3 = between_{A_0,\{\tilde{v}_1,\tilde{v}_2\}} \wedge between_{A_3,\{\tilde{v}_3,\tilde{v}_4\}}$$

After the insertion of the first four filter conditions in an empty BE-tree, the l-node of the root node overflows and must be split. Because all indexable predicates have the same selectivity, the popularity of the attributes becomes the only criteria determining the split attribute. Attribute A_3 is most popular among the first four filter conditions and chosen as split attribute therefore. During the insertion of the remaining filter conditions, the l-node of the root node must be split several times subsequently. Due to the characteristics of the filter conditions and their order, the exact same situation arises again and again so that at the end the attributes A_i with $i = 3, \ldots, 99$ are used to split the topmost l-node. At the second level, the space must be further partitioned, because each partition gets 100 filter conditions in total. Here, possible split attributes are limited to A_0, A_1 and A_2. Since each filter condition consists of exactly two indexable predicates, the height of the BE-tree is limited to three levels and, thus, all l-nodes at the third level cannot be further partitioned (see Definition 6). The final space partitioning is illustrated in Figure 11.6 (p-nodes are not shown).

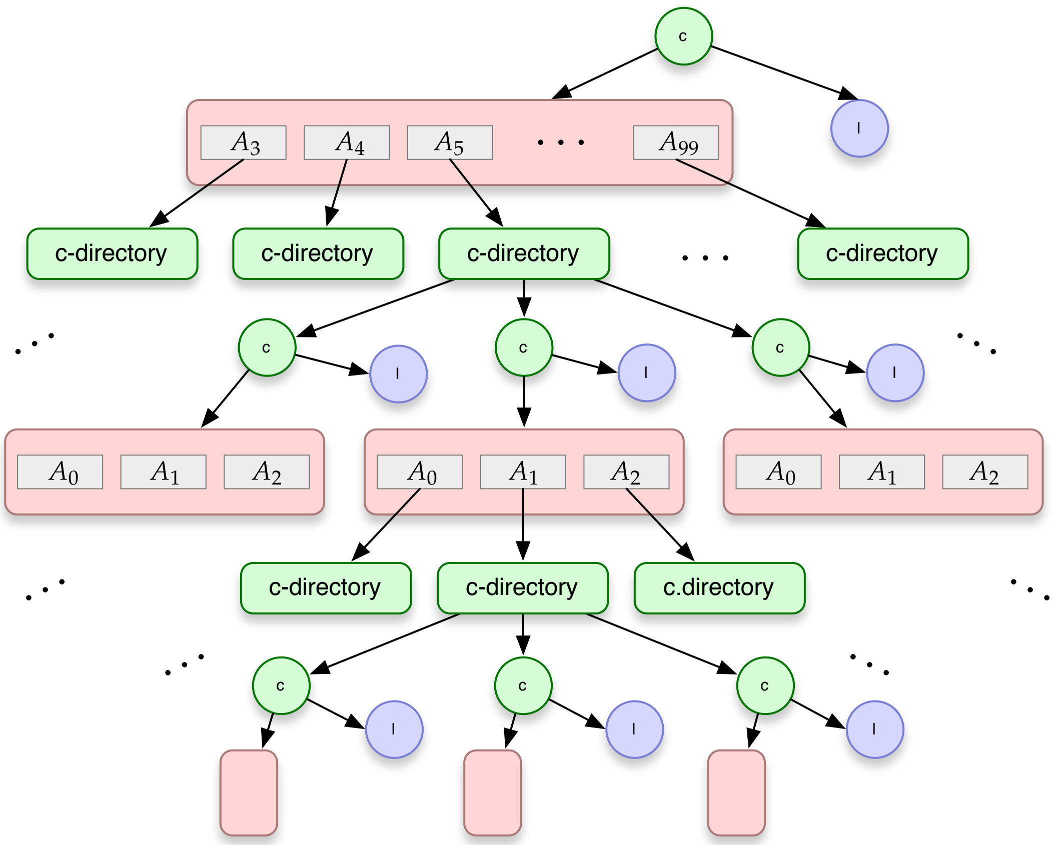


Figure 11.6: Suboptimal space partitioning

The illustrated BE-tree is poorly partitioned. This is because A_0, A_1 and A_2 are the most popular attributes among all filter conditions and, thus, the best attributes for partitioning at the root level. Analogues, the attributes A_i with $i = 3, \ldots, 99$ are less popular among all filter conditions and no good split attributes therefore. A globally optimal space partitioning (according to the scoring function) would use the attributes A_0, A_1 and A_2 for partitioning at the first level which is the most important one and the attributes A_i with $i = 3, \ldots, 99$ for partitioning at the second level. This globally optimal space partitioning is depicted in Figure 11.7 (p-nodes are not shown). A data item must now be probed against only three big partitions instead of 97 small partitions as a first step. Whenever a data item does not match any filter condition with respect to one (two, or three) of the popular attributes A_0, A_1 and A_2, one (two, or three) thirds of all continuous queries can be skipped at the topmost level. This substantially saves false positive computations and speeds up matching time. In our experiments, the performances of the two BE-trees differed by many times.

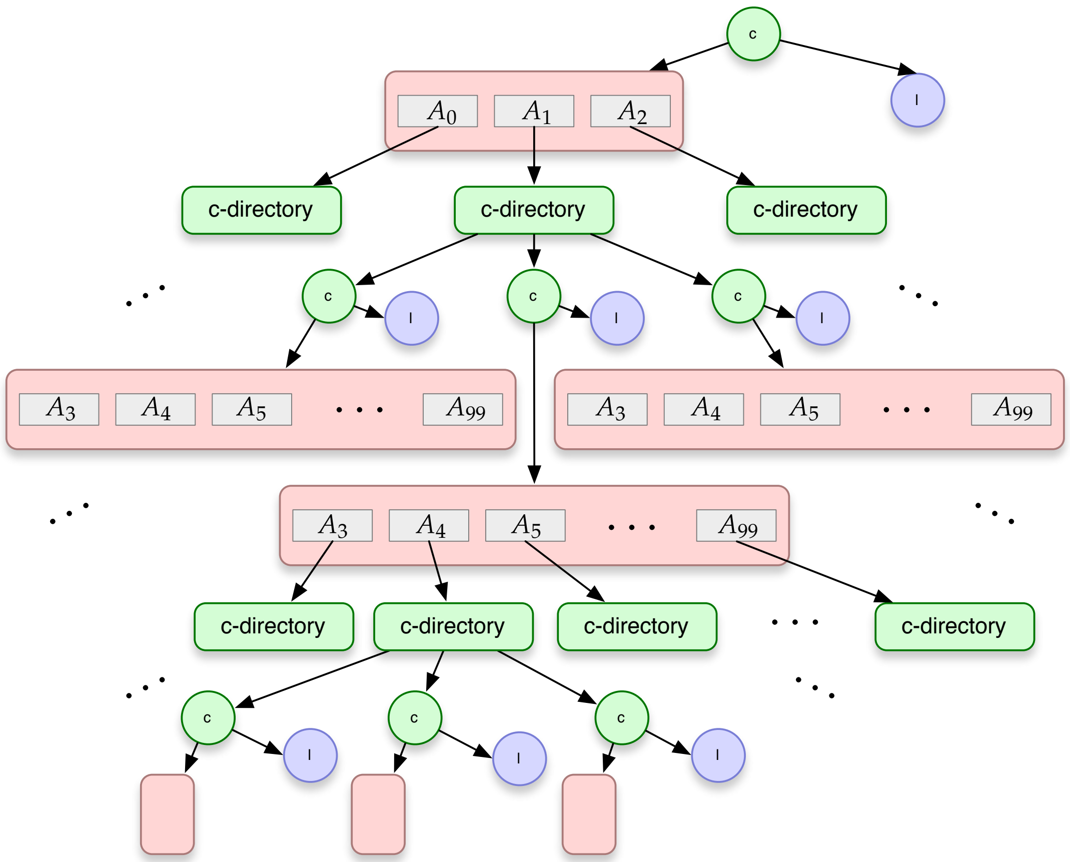


Figure 11.7: Globally optimal space partitioning

11.4.2 Inefficient Clustering

The strength of the BE-tree lies not only in the space partitioning, but also in the space clustering [Fab01]. In fact, our evaluation (see Section 11.8) reveals that for real-world EP applications space clustering is a much more important issue than space partitioning, because a random insertion order of filter conditions leads almost never to a worst case space partitioning. In contrast to the space partitioning, the shortcomings of the space clustering of the BE-tree lead to significantly decreased matching performance in not only synthetic EP applications, but also real-world EP applications. In our example EP application of this section, the c-directories are affected from all shortcomings we present and discuss in the following. Figure 11.8 shows how the c-directories of the BE-tree are structured in the case of our example EP application.

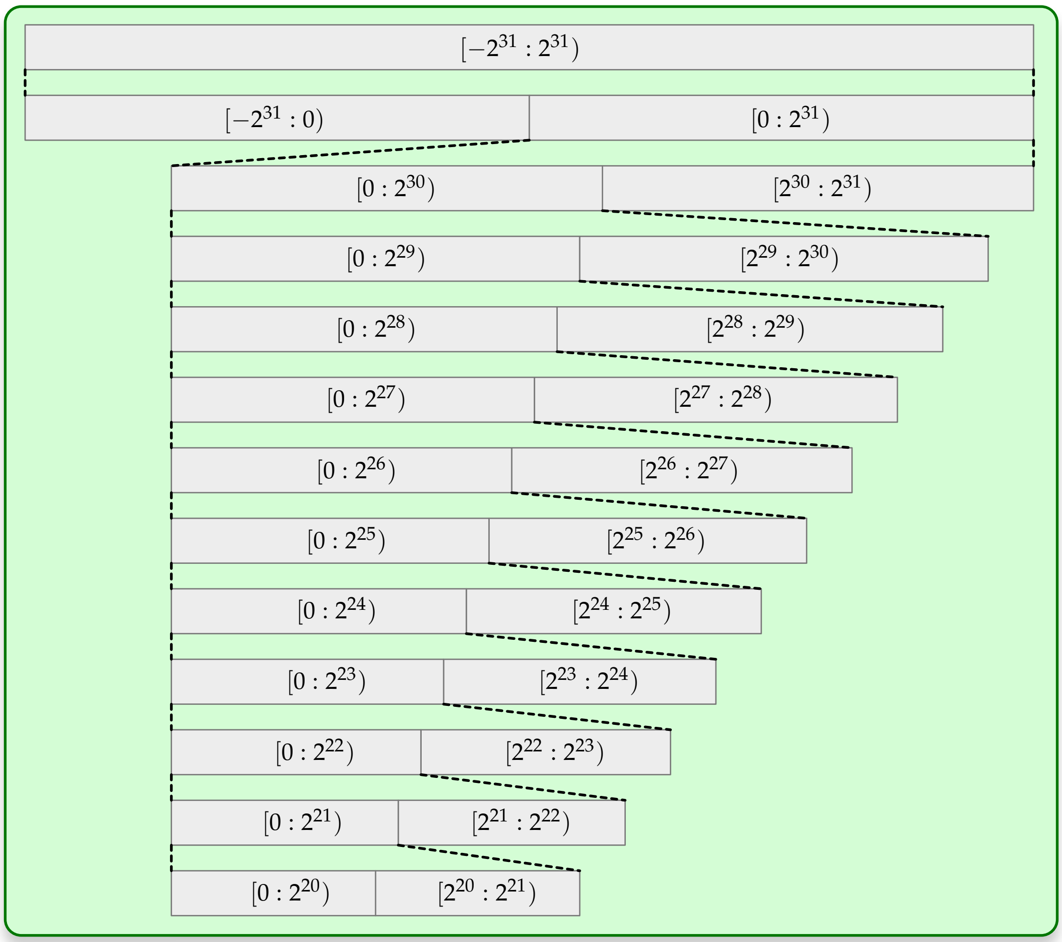


Figure 11.8: Inefficient cluster directory

Note that in Figure 11.8 the relative sizes and the arrangement of the c-directory cells are not correct for the sake of presentability. However, every c-directory cell is labeled with its exact and correct endpoints and connected via dashed lines with its child cells and parent cell to highlight the relationships. The shortcomings of the shown c-directory structure are discussed in the following paragraphs.

11.4.2.1 Oversized Coverage

As already mentioned earlier, attributes have domains with relatively large cardinalities in event processing. But based on the semantics of concrete applications, only small parts of the attribute domains are covered by events and, if carefully specified, by filter conditions in general. For instance, if an attribute of type 32-bit integer number models the temperature of an outdoor sensor, the price of a product in a supermarket, or the altitude of a moving object on earth, then only values close to zero are used. Therefore, the corresponding intervals of all indexable predicates likely cover only a small range of values in the domain. This is particularly true for our example EP application. As a consequence, all Boolean expressions are stored in a smaller subgrid so that every c-directory has an unnecessarily large number of levels. In our example EP application, each Boolean expression is stored in one of the cells $[0 : 2^{21})$, $[0 : 2^{20})$ and $[2^{20} : 2^{21})$. Every event with a value covered by or close to the subgrid where all non-empty clusters are located (irrespective wether or not it matches) must traverse through a high number of empty grid cells.

11.4.2.2 Steadily Empty Cells

When a non-atomic leaf cell of a c-directory overflows for the first time, it is split and all Boolean expressions of the overflowing cell as well as all Boolean expressions being inserted in the future that fit into one of the new child cells are stored there. However, it is possible that a child cell never receives any Boolean expression. In our example EP application, there is one steadily empty child cell after almost every split. In Figure 11.8, this is true for the c-directory cells $[-2^{31} : 0)$, $[2^{30} : 2^{31})$, $[2^{29} : 2^{30})$, $[2^{28} : 2^{29})$, $[2^{27} : 2^{28})$, $[2^{26} : 2^{27})$, $[2^{25} : 2^{26})$, $[2^{24} : 2^{25})$, $[2^{23} : 2^{24})$, $[2^{22} : 2^{23})$ and $[2^{21} : 2^{22})$. Although these c-directory cells are steadily empty, they were still created (with an associated c-node) and linked to the corresponding parent cell. For every event with a value in the range of a steadily empty c-directory cell an empty cluster must unnecessarily be checked.

11.4.2.3 Stuck Intervals

The space clustering of the BE-tree may lead to inefficient assignments of Boolean expressions to clusters and not fully exploit the inherent selectivity of indexable predicates therefore. This problem increases the rate of false positives and has the following simple root cause. An overflowing cell of a cluster directory is cut into halves. The middle of a split cell becomes the right endpoint of its left child cell and the left endpoint of its right child cell. Every interval that covers the middle of a cell gets stuck there forever. Stuck intervals can be quite different from each other in terms of size and position. Assume two intervals being stuck in the same cell (e.g., the first and last intervals in Figure 11.9). One interval covers the left half of the cell almost completely and the right half only a little bit. The other interval covers the right half almost completely and the left half only a little bit. Besides the fact that the selectivity of the associated predicates is exploited poorly, the intervals are different from each other and have only a very small value range in common. But according to the space clustering of the BE-tree, both intervals are assigned to the same cluster.

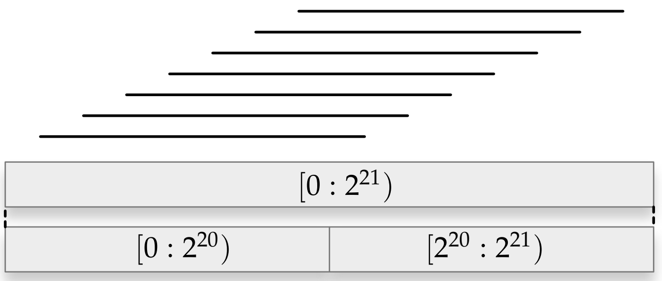


Figure 11.9: Stuck intervals

Figure 11.9 illustrates the situation in every c-directory of the BE-tree for our example EP application. Recall that all intervals have a size of 1,000,000 and are uniformly distributed within the range $[0 : 2,000,000]$. Hence, the figure depicts the subgrid that stores all Boolean expressions. Above the subgrid, the figure shows seven intervals of our example EP application that fit into the topmost cell. This cell is split because seven filter conditions exceed the maximum capacity. However, all intervals are stuck and cannot be moved into the child cells therefore. Every incoming event whose corresponding attribute value is covered by the subgrid successfully matches all these intervals despite the fact that only few of them are likely true positives.

11.5 BE$^+$-Tree

The problems that are presented and discussed in the previous section on basis of an example EP application are partly due to the fact that the BE-tree is a fully dynamic index structure supporting fast index updates at runtime. Most of the problems may have a strong negative effect on the matching performance which is the top priority in EP applications and many other kinds of stream processing applications. Typically, index updates occur with significantly lower frequency than events in EP applications [Agu99] and many EP applications even require no index updates at all.

Query workload	*Index update rate : data rate*	*Recommended query index*
Static	No index updates	BE$^+$-tree
Almost static	Index update rate $\ll$ Data rate	BE$^+$-tree
Dynamic	Everything else	BE-tree

Table 11.3: Application classes

Every stream processing application that potentially benefits from a query index falls into exactly one of the three classes listed in Table 11.3. Our classification is based on the index update rate and the data rate. If there are no index updates at runtime (i.e., all CQs are known in advance), then a stream processing application is classified as *static* and if there are index updates but at a significant lower rate than the data rate, a stream processing application is classified as *almost static*. Lastly, all stream processing applications that are neither *static* nor *almost static* are classified as *dynamic*. Event processing applications are *static* or *almost static*. In high-performance EP applications hundreds of thousands of new events occur every second and even non-static EP applications have update rates that by no means come close to these data rates.

The BE-tree supports fast index updates at the cost of maximum matching performance. For all stream processing applications of the class *dynamic* fast index updates are at least as important as fast processing of data items. Therefore, the BE-tree is the currently best suited query index for those stream processing applications. However, for stream processing applications of the other classes (fast) index updates are dispensable. Index updates are infrequent or do not occur at all and high matching performance is of utmost importance so that higher update costs are acceptable. Recall that query indexes in general and the BE-tree in particular are main memory data structures whose creation costs are orders of magnitudes lower than the creation costs of index structures for external memory. Altogether, this allows us to recreate an entire query index on updates in order to achieve high matching performance.

In this section, we introduce a modified version of the BE-tree named *BE$^+$-tree* that provides several optimizations to avoid the problems the original BE-tree suffers from. The overall structure of a BE$^+$-tree is still the structure of a BE-tree and existing implementations of the BE-tree can be easily extended therefore. Furthermore, a BE$^+$-tree can have all properties of the original BE-tree. Based on the applied optimization techniques, the dynamic property might get lost in a BE$^+$-tree. However, index updates are still supported in every configuration of the BE$^+$-tree (see Section 11.7).

11.5.1 Bulk Loading of BE-Trees

In the following, we assume a fixed set of continuous queries for that a new query index must be created from scratch. Therefore, BE$^+$-trees are created using a tailor-made bulk loading technique in contrast to original BE-trees that must be created via query-by-query insertion.[1]

11.5.1.1 Optimization of Space Partitioning

Since all Boolean expressions are known in advance, we can create a BE-tree so that it is globally optimally partitioned (according to the used scoring function). In case of the BE$^+$-tree, the optimal space partitioning is simply achieved via bulk loading of c-nodes. Every c-node is the root node of a (sub-)tree. If we put all Boolean expressions of a (sub-)tree into a new c-node and apply the scoring function, the globally best split attribute is returned. This step can be repeated in order to create entire and optimally partitioned (sub-)trees.

Algorithm 8 shows the bulk loading of c-nodes. Besides a set containing all indexable Boolean expressions to insert and an empty c-node being loaded, another set containing all attributes that were already used for partitioning must be given. The algorithm repeatedly partitions the set of indexable Boolean expressions until its size is equal to or less than the user-defined maximum capacity of l-nodes max_{cap} (line 1). During partitioning, always the best attribute among all remaining indexable Boolean expressions is selected (lines 2–4). The space partitioning property (Definition 9) is completely respected by reusing GETPARTITIONINGATTRIBUTE of the original BE-tree. This means, all attributes already used for partitioning (contained in the set *path*) and attributes which do not have the user defined minimum support $min_{support}$ are excluded. Therefore, the partitioning stops when there are no (more) suitable attributes available. Analogous to the space partitioning of the original BE-tree, every partition

[1]Note that BE$^+$-trees having the dynamic property also support query-by-query insertions.

Algorithm 8: LOADCNODE($\mathcal{Q}$, *cNode*, *path*)

```
Input: Set of Boolean Expressions: $\mathcal{Q}$, C-Node: $cNode$, Set of Attributes: $path$
1  while SIZEOF($\mathcal{Q}$) > $max_{cap}$ do
2      Attribute $attr \leftarrow$ GETPARTITIONINGATTRIBUTE($\mathcal{Q}$, $path$);
3      if $attr$ = NULL then
4          break;
5      Set of Boolean Expressions $\mathcal{Q}' \leftarrow \emptyset$;
6      foreach Boolean Expression $be \in \mathcal{Q}$ do
7          if $be$.RESTRICTS($attr$) then
8              $\mathcal{Q}$.REMOVE($be$);
9              $\mathcal{Q}'$.ADD($be$);
10     P-Node $pNode \leftarrow cNode.pDirectory$.NEWPARTITION($attr$);
11     LOADPNODE($\mathcal{Q}'$, $pNode$, $path \cup \{pNode.attribute\}$);
12 $cNode.lNode$.ADDALL($\mathcal{Q}$);
```

results in a new p-node that is added to the p-directory of the c-node being loaded. Then, all indexable Boolean expressions of a new partition are moved from the input set into the new partition also in the form of a bulk operation (lines 5–11). For creating the new p-node another bulk loading algorithm of the BE$^+$-tree (LOADPNODE) that we present in the next section is used. After the partitioning has stopped, all remaining indexable Boolean expressions are put into the l-node of the c-node being loaded (line 12). Every l-node created via the algorithm respects the leaf node property (Defintion 6). This means that a leaf node of a BE$^+$-tree is allowed to overflow only if there is no suitable attribute for partitioning available. The creation of a new BE$^+$-tree starts via Algorithm 8 called with an empty c-node, the entire set of indexable Boolean expressions and an empty set for *path*.

11.5.1.2 Optimization of C-Directories

According to Definition 7, the root cell of a new cluster directory must cover the entire domain of its associated attribute. As a consequence, every arbitrarily expanded cluster directory entirely covers its associated attribute domain too and leads to the problems of oversized coverage (see Section 11.4.2.1) and of steadily empty cells (see Section 11.4.2.2). The BE$^+$-tree allows to avoid the problem of oversized coverage via the user-defined parameter `GRID_TIGHTENING`. Algorithm 9 shows how this parameter affects the creation of new c-directories.

Algorithm 9: LoadPNode($\mathcal{Q}$, *pNode*, *path*)

```
Input: Set of Boolean Expressions: $\mathcal{Q}$, P-Node: $pNode$, Set of Attributes: $path$
1  if IsEnabled(GRID_TIGHTENING) then
2      Endpoint $left \leftarrow$ GetDomain($pNode.attribute$).GetMaxValue();
3      Endpoint $right \leftarrow$ GetDomain($pNode.attribute$).GetMinValue();
4      for Boolean Expression $be \in \mathcal{Q}$ do
5          if $be$.GetLeftEndpoint($pNode.attribute$) < $left$ then
6              $left \leftarrow be$.GetLeftEndpoint($pNode.attribute$);
7          if $be$.GetRightEndpoint($pNode.attribute$) > $right$ then
8              $right \leftarrow be$.GetRightEndpoint($pNode.attribute$);
9      $pNode.cDirectory.root \leftarrow [left : right]$;
10 else
11     Endpoint $left \leftarrow$ GetDomain($pNode.attribute$).GetMinValue();
12     Endpoint $right \leftarrow$ GetDomain($pNode.attribute$).GetMaxValue();
13     $pNode.cDirectory.root \leftarrow [left : right]$;
14 LoadCDirectory($\mathcal{Q}$, $pNode.cDirectoy.root$, $path$, $pNode.attribute$);
```

If the parameter `GRID_TIGHTENING` is disabled, the algorithm creates for each new partition a c-directory that covers the entire domain of the associated attribute exactly as the original BE-tree (lines 10–13). As a consequence, Definition 7 is respected so that the resulting BE$^+$-tree has the dynamic property and, thus, is a completely valid BE-tree. But if the parameter `GRID_TIGHTENING` is enabled, the algorithm creates c-directories having tightened grids that are of minimal size (lines 1–9). This means that the grid of each c-directory minimally covers the corresponding intervals of all predicates. For this purpose, the algorithm computes the minimum and maximum endpoints among the corresponding intervals of all predicates and creates a c-directory with a root cell that only covers all values between these endpoints. Note that the tightening of grids violates Definition 7 and leads to the loss of the dynamic property. Tightened grids prevent the insertion of new indexable Boolean expressions into an existing BE$^+$-tree for the following reason. Let the domain of an attribute start with some value x_1 and end with some greater value x_2. A tightened grid covering a range $[x'_1 : x'_2]$ with $x_1 < x'_1 < x'_2 < x_2$ does not cover the ranges $[x_1 : x'_1)$ and $(x'_2 : x_2]$ of the attribute domain. Each predicate whose corresponding interval overlaps or falls completely into a range uncovered by the grid cannot be inserted. According to Definition 8 there exists no cell to which it could be added.

Algorithm 10: LOADCDIRECTORY($\mathcal{Q}$, *cell*, *path*, *attr*)

```
Input: Set of Boolean Expressions: $\mathcal{Q}$, C-Directory Cell: $cell$, Set of Attributes: $path$,
       Attribute: $attr$
if SIZEOF($\mathcal{Q}$) > $max_{cap}$ then
    Number $counter \leftarrow 0$;
    Endpoint $minValue \leftarrow cell$.GETMINVALUE();
    Endpoint $maxValue \leftarrow cell$.GETMAXVALUE();
    Endpoint $middle \leftarrow minValue/2 + maxValue/2$;
    if not ISATOMIC($cell$) then
        if ISENABLED(GRID_TIGHTENING) then
            $counter \leftarrow$ CHECKSPLIT($\mathcal{Q}$, $cell$);
        else
            $counter \leftarrow min_{support}$;
    if $counter \geq min_{support}$ then
        Set of Boolean Expressions $\mathcal{Q}_l \leftarrow$ QUERY($\mathcal{Q}$, $attr$, $[minValue : middle]$);
        Set of Boolean Expressions $\mathcal{Q}_r \leftarrow$ QUERY($\mathcal{Q}$, $attr$, $[middle + 1 : maxValue]$);
        $\mathcal{Q} \leftarrow (\mathcal{Q} \setminus \mathcal{Q}_l) \setminus \mathcal{Q}_r$;
        if ISENABLED(SPLIT_INTERVALS) then
            SPLITINTERVALS($\mathcal{Q}$, $\mathcal{Q}_l$, $\mathcal{Q}_r$, $middle$, $attr$);
        if SIZEOF($\mathcal{Q}_l$) > 0 or not ISENABLED(AVOID_EMPTY_CELLS) then
            $cell.leftChild \leftarrow$ NEWCELL($[minValue : middle]$);
            LOADCDIRECTORY($\mathcal{Q}_l$, $cell.leftChild$, $path$, $attr$);
        if SIZEOF($\mathcal{Q}_r$) > 0 or not ISENABLED (AVOID_EMPTY_CELLS) then
            $cell.rightChild \leftarrow$ NEWCELL($[middle + 1 : maxValue]$);
            LOADCDIRECTORY($\mathcal{Q}_r$, $cell.rightChild$, $path$, $attr$);
if SIZEOF($\mathcal{Q}$) > 0 then
    LOADCNODE($\mathcal{Q}$, $cell.cNode$, $path$)
```

After the creation of a new c-directory, Algorithm 9 is called to bulk load it in a top-down fashion. This is done according to Definition 10 that requires overflowing nodes to be further clustered before space partitioning is allowed to be performed. That requirement is important also for BE$^+$-trees to balance the space partitioning and space clustering phases. Whether to split a non-atomic and overflowing cell is determined by *counter* in the algorithm. If `GRID_TIGHTENING` is disabled, then *counter* is immediately set to $min_{support}$ so that a split is performed definitely (line 10). This implements exactly the behavior of the original BE-tree. A consequence is that new cells are created into which possibly no or only few Boolean expressions can be pushed down. For empty or almost empty clusters it can be expected that future insertions will fill them up. But this assumption is not true if `GRID_TIGHTENING` is enabled. Then, empty or almost empty clusters will remain in their state because there are no future insertions at all. Therefore, a split is performed in case of a non-dynamic BE$^+$-tree if and only if there are enough Boolean expressions that can be pushed down (line 8). The algorithm CHECKSPLIT simply counts all Boolean expressions that can be pushed down if the current cell is split (CHECKSPLIT also takes into account all possible interval splits, see the following section for details). Basically, a new parameter that specifies the threshold would be necessary. But instead of introducing another parameter, we reuse the existing parameter $min_{support}$. This parameter is appropriate, because it already defines the minimum size of new partitions.

The split of a cell during bulk loading is performed as follows (lines 11–22). From the entire set of Boolean expressions to add two new sets are derived. One of the new sets contains all Boolean expressions that can be pushed down into the left child cell and the other contains all Boolean expressions that can be pushed down into the right child cell. For obtaining those Boolean expressions, the algorithm QUERY is used. It takes a set of Boolean expressions, the corresponding attribute as well as a range of values and returns a new set containing all Boolean expressions of the input set that completely fall into the given range of values. All Boolean expressions that can be pushed down are removed from the original set. The parameter `SPLIT_INTERVALS` is for the handling of stuck intervals and discussed in the following section. Finally, the cell can be split by creating the left and the right child cells. For the child cells, bulk loading is recursively performed using the sets $\mathcal{Q}_l$ and $\mathcal{Q}_r$ respectively. Both child cells or one of them might be completely empty. This happens if no Boolean expressions can be pushed down at all or if all Boolean expressions that can be pushed down are inserted into the same child cell (as it is true for almost all child cells in the c-directories of the example EP application). Empty cells can be avoided by enabling the parameter `AVOID_EMPTY_CELLS`. Then, empty cells are not created and the corresponding pointers of the parent cells remain set to `NULL`. There are no invasive consequences (except better matching performance) if there are no insertions in the future. However, an empty cell that has not been created can always be created on demand when the very first Boolean expression is inserted. At the end, the algorithm inserts all remaining Boolean expressions of the input set into the cell currently being processed via bulk loading of its cluster node (lines 23–24).

11.5.2 Handling of Stuck Intervals

The problem of stuck hypercubes is well-known for grid indexes that split cells into equally sized subcells (e.g., quad trees [FB74]), particularly if they are used to index high-dimensional data. Here, the curse of dimensionality [Bis06, Sco92] leads for uniformly and independently distributed data to the phenomenon that the number of stuck hypercubes rapidly grows as the number of dimensions increases. This results in the serious problem that an increasing amount of hypercubes cannot be pushed down into subcells, because they cover the borders of the grid. To overcome this problem, the GESS approach [DS01] proposes to split hypercubes that cover the borders of the grid so that the resulting parts can be further pushed down. Obviously, this introduces redundancy because a hypercube that has been split occurs in the form of copies in multiple cells. Performing every split that is possible may lead to a high degree of redundancy and not every split improves the index performance. Therefore, splits should be performed in a controlled way.

The c-directories of BE-trees are grid indexes in only one-dimensional space so that the curse of dimensionality has no impact. However, there are other factors that lead to stuck intervals. With respect to the mapping of predicates to intervals (see Table 11.2), there are several types of predicates that are mapped to large intervals in general. For example, the open-ended predicates $<$, $\leq$, $\geq$, $>$ may cover a large part of the domain. Also, the predicates *between* and $\in$ can result in an interval of any size.[2] Furthermore, the distribution of intervals within a cell (e.g., normal distribution) might encourage intervals to get stuck. The BE$^+$-tree allows to split stuck intervals and to control which and how many intervals are split. In Algorithm 10, the call of a method for splitting intervals is already integrated (line 15). We present this method in the following.

Algorithm 11 shows how and which stuck intervals are split if the user-defined parameter `SPLIT_INTERVALS` is enabled. The loop checks every Boolean expression whether it should be split and performs the split if necessary. Of course, the split technique is only applicable to stuck intervals (line 2). Then, the final decision whether to split a stuck interval is controlled by three user-defined parameters. The total number of splits that can be performed for a Boolean expression can be limited both locally and globally. The global limit is specified by the parameter `TOTAL_SPLIT_LIMIT`. During insertion, a Boolean expression is allowed to be split at most as many times as the parameter defines (line 4). Before the bulk loading of a new tree starts, a global split counter for each Boolean expression is set to zero and never reset. Every single split of

[2]Because the corresponding intervals of the predicates $\neq$ and $\notin$ always cover the entire domain, they are best stored in the root cell and excluded from our listing of affected predicates therefore.

Algorithm 11: SPLITINTERVALS(Q, Q_l, Q_r, *middle*, *attr*)

```
Input: Set of Boolean Expressions: $Q$, $Q_l$, $Q_r$, Endpoint: $middle$, Attribute: $attr$
1  for Boolean Expression $be \in Q$ do
2      if $be$.ENCLOSES($middle$) then
3          if $be$.GETLOCALSPLITCOUNT() < GRID_SPLIT_LIMIT then
4              if $be$.GETTOTALSPLITCOUNT() < TOTAL_SPLIT_LIMIT then
5                  $lc \leftarrow middle - ((middle - cell$.GETMINVALUE())$*$SPLIT_FACTOR);
6                  $rc \leftarrow middle + ((cell$.GETMAXVALUE() $- middle)*$SPLIT_FACTOR);
7                  if $lc \leq be$.GETLEFTENDPOINT($attr$) or
                   $rc \geq be$.GETRIGHTENDPOINT($attr$) then
8                      $be_l \leftarrow$ COPYANDADJUST($be$, $attr$,
                       [$be$.GETLEFTENDPOINT($attr$) : $middle$]);
9                      $Q_l$.ADD($be_l$);
10                     $be_r \leftarrow$ COPYANDADJUST($be$, $attr$,
                       [$middle + 1$ : $be$.GETRIGHTENDPOINT($attr$)]);
11                     $Q_r$.ADD($be_r$);
12                     $Q$.REMOVE($be$);
13                     $be_l$.INCREMENTSPLITCOUNTERS();
14                     $be_r$.INCREMENTSPLITCOUNTERS();
```

a Boolean expression increments its global split counter. The local limit is specified by the parameter GRID_SPLIT_LIMIT and restricts the total number of splits that can be performed within a single c-directory. Before the bulk loading of a c-directory starts, a local split counter for every Boolean expression is set to zero. For each split that is performed for a Boolean expression during bulk loading of a c-directory, its local split counter is incremented. If a Boolean expression is allowed to be split according to the local and global split limits, a third and final condition is checked. In lines 5–6, the degree of coverage is computed for both the left child cell and the right child cell. The differences in the innermost brackets give the sizes of the entire ranges of the left and right child cells. These sizes are scaled down by a third user-defined parameter SPLIT_FACTOR that must be set to a value in $[0 : 1]$. Figure 11.10 illustrates the meaning of the parameter SPLIT_FACTOR (denoted by SF in the figure). It is used to place the endpoints rc and lc. These endpoints have always an equal distance to *middle*. The parameter SF is used to control the size of the distance. If SF is increased then the distance increases. Analogues, if SF is decreased then the distance decreases. In line 7, the endpoints rc and lc are used to only split stuck intervals that have their left endpoint between *lc* and *middle*, or their right endpoint between *middle* and *rc*, or both. In other words, this requires that the size of at least one overlapping part does not exceed a certain threshold. With the help of this parameter, it is possible to perform only splits that have a significantly positive effect on the performance. This issue is discussed intensively in the next paragraphs. But before, we finish the presentation of Algorithm 11 by describing the split of intervals (lines 8–14). At first, two copies be_l and be_r of the corresponding Boolean expression *be* are created via COPYANDADJUST. This algorithm creates an exact copy of *be*, adjusts the interval of the corresponding predicate on the attribute associated with the c-directory and returns the adjusted copy. Then, the copies are added to the sets $\mathcal{Q}_l$ and $\mathcal{Q}_r$ which keep all Boolean expressions for the bulk loading of the child cells. Lastly, *be* is removed from the original set and the split counters of both copies are incremented.

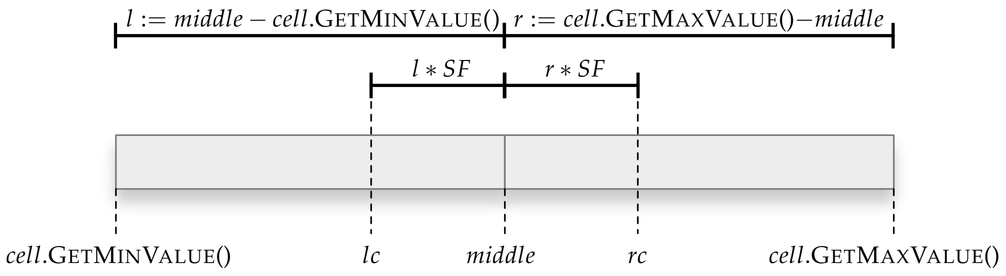


Figure 11.10: Parameter split factor

Each stuck interval is always considered as a matching candidate for every data item that visits its cell. The larger the uncovered range of an interval is, the more likely the interval is a false positive match for visiting data items. Thus, stuck intervals which do not cover a large range should be split in order to reduce false positives and to improve the matching performance.

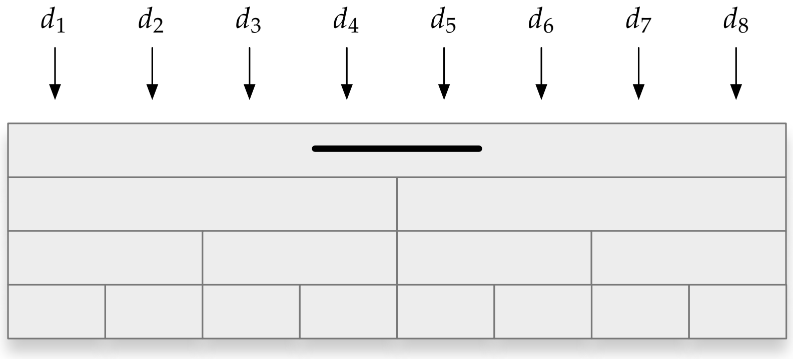


Figure 11.11: Tiny stuck interval

Figure 11.11 shows a stuck interval that is a perfect candidate for being split. It covers only small ranges of both halves while two large ranges are uncovered. Assume for example that Boolean expressions must be looked up for the data items $d_1, d_2, \ldots, d_8$. The arrows in the figure indicate the positions of the data items. Because the stuck interval is located in the topmost cell, it matches every shown data item. However, only in case of the data items d_4 and d_5 this is a correct result. In case of the other data items (i.e., 75 % of all shown data items) the stuck interval is a false positive.

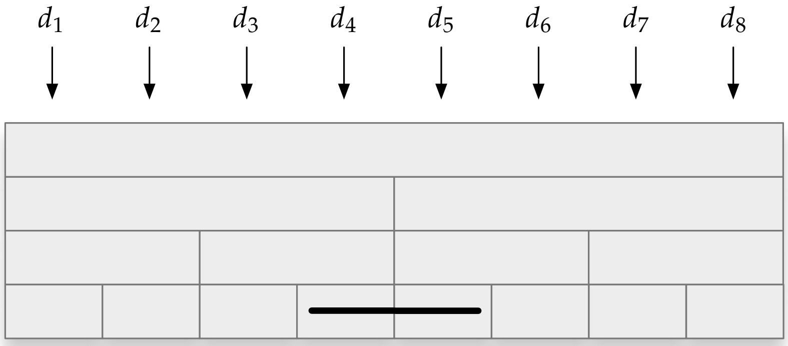


Figure 11.12: Split interval

To improve the matching performance, the stuck interval should be split, because the resulting parts could be pushed down into cells located several levels below the topmost cell. Figure 11.12 shows the situation after the interval has been split and the parts have been pushed down as far as possible. Now, the interval still matches the data items d_4 and d_5, but in case of the other data items it is no false positive anymore.

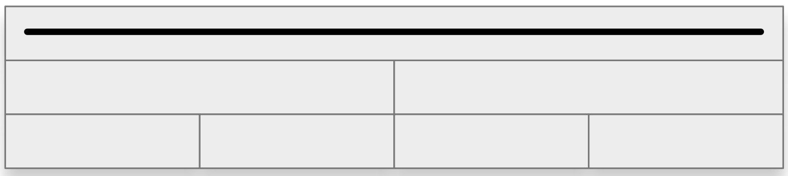


Figure 11.13: Huge stuck interval

As already stated, not every possible split of a stuck interval improves the matching performance. In general, if there are no large ranges being uncovered by a stuck interval, then the interval is already in the best cell. Figure 11.13 shows a stuck interval which should not be split, because the resulting parts could only be pushed down to the next level at which they would get stuck again. If an interval is pushed down only to the next level, the matching performance is not influenced at all. This is because every data item that visits some cell also visits one of its two child cells (see Algorithm 2). Therefore, every data item that visits the topmost cell in Figure 11.13 would still consider the interval as a match regardless whether it is a true positive or a false positive. Based on this observation, we can conclude that all values higher than 0.5 for `SPLIT_FACTOR` have no additional positive effects in general, but increase the degree of redundancy. Therefore, `SPLIT_FACTOR` should be set to a value between 0 and 0.5 in order to reduce false positives. A value of 0.5 ensures that at least one part after a split can be pushed down by at least two levels. If `SPLIT_FACTOR` is decreased further, the number of levels one part can be definitively pushed down increases.

Theorem 1. *Let i be any positive integer number. If the value of* `SPLIT_FACTOR` *is less than or equal to $\frac{1}{2^{i-1}}$, then it is guaranteed that at least one part can be pushed down by at least i levels after every split of a stuck interval.*

Proof. The theorem is proven by mathematical induction. For $i = 1$, every stuck interval that satisfies the split factor condition can be pushed down at least to the next level. The base case of the induction is $i = 2$. For $i = 2$, the split factor condition requires that at least one part of a stuck interval is at most of half the size of a child cell. This part gets not stuck if it is pushed down to the next level. This is because the part is at most half the size of the child cell and one of its endpoints is identical to one endpoint of the child cell so that the middle of the child cell is not covered. Consequently, the part can be further pushed down by at least one more level. For the induction step, we can assume that for a split factor of $\frac{1}{2^{i-1}}$ one part of every stuck interval can be pushed down by at least i levels. If i is now increased by exactly one, the maximum allowed size of at least one part is divided by 2. Then, the exact same argumentation as in the case $i = 2$ can be reused to conclude that for $i = i' + 1$ at least one part can be pushed down by at least one more level than in case of i'. $\square$

Note that there are some special situations in which a higher value for `SPLIT_FACTOR` can improve the matching performance. Values between 0.5 and 1.0 could be chosen if many cascading interval splits are possible. This means, an interval is recursively split several times in order to push it down by multiple levels. Of course, this requires that there are multiple splits per grid and in total allowed. Furthermore, the distribution of data items is of utmost importance. Cascading splits are beneficial only if a high rate of data items falls into the small uncovered ranges of the cell an interval initially got stuck. Therefore, values greater than 0.5 for `SPLIT_FACTOR` are only recommended for applications with the special characteristics described above.

Since each split duplicates the corresponding Boolean expression so that two elements instead of one must be stored, it is important to know the relationship between the number of splits and the number of copies. In the following, we derive a formula for the (theoretical) upper bound of the maximum total number of Boolean expressions a BE$^+$-tree with interval splits enabled must store in the worst case. The first theorem gives the maximum total number of splits of a single Boolean expression.

Theorem 2. *The maximum possible total number of interval splits max_{splits} that can be performed for a single Boolean expression is:*

$$max_{splits} = \min\{\texttt{TOTAL\_SPLIT\_LIMIT}, \#attributes * \texttt{GRID\_SPLIT\_LIMIT}\}$$

Proof. Obviously, max_{splits} is limited by `TOTAL_SPLIT_LIMIT`. For each performed interval split, a counter per Boolean expression is incremented by one. This counter is never reset and interval splits are performed only if this counter has a value less than or equal to `TOTAL_SPLIT_LIMIT`. But it is possible that there is a lower limit due to `GRID_SPLIT_LIMIT`. A Boolean expression and every copy of it can appear in at most *#attributes* many c-directories (see Definition 9). Within each c-directory, `GRID_SPLIT_LIMIT` limits the number of interval splits that are allowed per Boolean expression. If the product of *#attributes* and `GRID_SPLIT_LIMIT` is less than `TOTAL_SPLIT_LIMIT`, then this value defines the maximum possible number of interval splits. Otherwise, the limit is defined by `TOTAL_SPLIT_LIMIT`. □

Note that Theorem 2 assumes that every Boolean expression can potentially be split max_{splits} times (i.e., it must get stuck and satisfy the split factor condition extremely often). Based on max_{splits}, the maximum possible number of copies of a single individual Boolean expression can be determined.

Theorem 3. *The maximum possible total number of copies max_{copies} of a single individual Boolean expression in a BE$^+$-tree is:*

$$max_{copies} = 2^{max_{splits}}$$

Proof. Every interval split replaces a copy of a Boolean expression by two new ones. According to Algorithm 11, each copy gets its own counters which are only incremented when the copy itself is split (and not when other copies are split). Those counters are initialized with the values of the original copy at the time of the split and incremented by one. In the worst case, a single counted interval split can result in the doubling of all copies at the same level of a c-directory. This behavior is described by the function 2^x while the maximum number of doublings x is limited by max_{splits}. □

Our evaluation shows that max_{splits} is a theoretical upper bound that is higher than the average number of splits of a Boolean expression in practice. Consequently, max_{copies} is much higher than the average number of copies of Boolean expressions because of its exponential growth. Therefore, the formula is not appropriate for getting good estimations in practice. But it shows that the number of copies can grow rapidly when the number of splits increases. Therefore, it is important to control the number of splits, because otherwise the number of copies and the size of the corresponding BE$^+$-tree might escalate. The worst case size of a BE$^+$-tree is given by the product of max_{copies} and the total number of input Boolean expressions ($\#expressions$).

Corollary 1. *The maximum possible total number of copies of Boolean expressions max_{size} a BE$^+$-tree with interval splits enabled must store is:*

$$max_{size} = \#expressions * max_{copies}$$

The upper bound max_{size} is virtually never reached in practice. This is not only because max_{copies} is almost always overestimated, but also because of Algorithm 8. Due to space partitioning and space clustering, a set of Boolean expressions is successively divided into smaller sets during bulk loading. As soon as such a set becomes smaller than max_{cap}, it is directly stored in a l-node at an internal level (see line 1 in Algorithm 8). For instance, every interval could be split and actually was split in the first experiment in Section 11.8.2. The theoretical maximum number of splits per Boolean expression was 20 according to Theorem 2, because there were 20 attributes and only one split was allowed per c-directory (the total split limit was set to 100). But in the resulting BE$^+$-tree, the average number of splits was about 6.4 that is significantly lower than the worst case estimation (despite the fact that we used a synthetic query workload that was intended to maximize interval splits). Only a tiny fraction of all Boolean expressions was split 20 times. Most Boolean expressions were stored in l-nodes at internal levels and split significantly less times than stated by max_{splits}.

11.6 Implementation

We implemented both the original BE-tree and the BE$^+$-tree as query index for the native EP provider of JEPC. In fact, both implementations are based on the same data structures. Only the algorithms to create and maintain query indexes differ. The BE-tree and particularly the BE$^+$-tree have many parameters that can be used to configure the algorithms. However, it is also possible to create an instance of the native EP provider without specifying any parameters. Then, a default configuration is used.

```java
public class FilterIndexConfiguration {
        // available evaluators
        public static enum EVALUATOR { CONJUNCTIVE, DNF, COMPLEX }
        // general parameters
        public int          maxCap                                     = 16;
        public int          minSupport                                 = 8;
        public EVALUATOR    evaluator                                  = CONJUNCTIVE;
        // BE+-tree-specific parameters
        public boolean      bulkLoading                                = false;
        public boolean      avoidEmptyCells                            = true;
        public boolean      gridTightening                             = true;
        public boolean      splitIntervals                             = true;
        public int          totalSplitLimit                            = 100;
        public int          gridSplitLimit                             = 1;
        public double       splitFactor                                = 0.0001;
}
```

Listing 11.1: Default configuration of the query index of the native EP provider

Listing 11.1 shows the Java class `FilterIndexConfiguration` that represents the configuration of the query index. It is globally valid for an instance of the native EP provider. The first three parameters `maxCap`, `minSupport` and `evaluator` are shared by the BE-tree and the BE$^+$-tree. In particular, the parameter `maxCap` represents max_{cap} with default value 16 and the parameter `minSupport` represents $min_{support}$ with default value 8. The parameter `evaluator` is explained in Section 11.6.2. All other parameters are used only by the BE$^+$-tree. Which type of query index is used is determined by the parameter `bulkLoading`. If it is enabled, all indexes are BE$^+$-trees. Otherwise, all indexes are BE-trees. Via the parameters `avoidEmptyCells`, `gridTightening` and `splitIntervals` all by the BE$^+$-tree provided optimizations can be individually enabled and disabled. The parameter `avoidEmptyCells` is for the optimization that prevents the creation of empty c-directory cells. If `gridTightening` is enabled, then c-directories are tightened at the cost of the dynamic property. The parameter `splitIntervals` determines whether stuck intervals should be split. Lastly, the three parameters `totalSplitLimit`, `gridSplitLimit` as well as `splitFactor` are used by the interval split technique and self-explaining. Users are allowed to create their own configurations. In case the BE$^+$-tree is used, new filter EPAs are maintained in lists until they are moved via bulk loading into the corresponding indexes. To ensure semantical correctness, incoming events are forwarded to the corresponding index and to every corresponding filter EPA that has not been indexed yet.

11.6.1 Optimization of Boolean Expressions

For the sake of optimal performance and wide support of the query index, Boolean expressions are rewritten before indexing in our implementation. Some of the obligatorily applied rules are presented in Table 11.1. Every pair of indexable predicates with reversed half-open operators and referring to the same data attribute is replaced by a single indexable *between* predicate. If there are multiple indexable $=$ predicates referring to the same data attribute, then they are replaced by a single indexable $\in$ predicate. Analogous, multiple indexable $\neq$ predicates referring to the same data attribute are replaced by a single indexable $\notin$ predicate. In addition, our optimizer for Boolean expressions is able to detect expressions or subexpressions that will always or never evaluate to `TRUE`. Such (sub-)expressions are replaced by the Boolean constants `TRUE` and `FALSE` respectively. Lastly, our optimizer detects subexpressions that are completely covered by other subexpressions and removes them (e.g., $between_{attr,\{-5,4\}}$ is completely covered by $between_{attr,\{-8,6\}}$).

11.6.2 Indexing of Arbitrary Boolean Expressions

Boolean expressions can consist of both indexable and non-indexable predicates. Moreover, predicates can be connected via any logical connector in Boolean expressions. But the BE-tree and the BE^{+}-tree support only indexable Boolean expressions. However, our implementation extracts non-indexable predicates. If a non-indexable predicate refers to no data attribute, it is evaluated and replaced by the constant result (i.e., either `TRUE` or `FALSE`) and if it refers to two data attributes, it is evaluated for each incoming event separately. All remaining predicates of a Boolean expression are indexable. Such Boolean expressions are handled according to the configured mode (parameter `evaluator`). In our implementation, there are three different modes available. In the mode `CONJUNCTIVE`, only Boolean expressions in conjunctive form are indexed. All others are not indexed and evaluated for every incoming event. In the mode `DNF`, the native EP provider indexes Boolean expressions that are in disjunctive normal form (DNF). At each logical $\vee$ operator, a Boolean expression in DNF is split so that the result is a collection of Boolean expressions in conjunctive form [YG94]. Those Boolean expressions are all in an appropriate form for being indexed. The third mode is named `COMPLEX`. In this mode, arbitrarily complex Boolean expressions can be handled. Based on several rewriting and processing rules, a Boolean expression is decomposed into a set of subexpressions. All elements of the set that can be indexed are indexed and all others are evaluated on per event basis.

11.7 Handling Dynamic Sets of Continuous Queries

In general, the BE$^+$-tree is a static index structure that cannot be updated. Thus, an updated set of Boolean expressions requires the entire recreation of the corresponding BE$^+$-tree. Our experiments proved that the creation of even large BE$^+$-trees needs only a few seconds. The BE$^+$-tree is a main memory data structure and can be created very fast in comparison to the creation times of data structures that reside on external memory. Moreover, the creation of a BE$^+$-tree from scratch is significantly faster than the creation of a BE-tree from scratch. This is because BE$^+$-trees are created efficiently via bulk loading while BE-trees are created via costly query-by-query insertion. For applications with relatively low update rates such as, for example, EP applications the BE$^+$-tree is usable without any restrictions and recommended because of its better matching performance. However, the recreation of a BE$^+$-tree is a blocking operation. This means, the processing of data items must be paused until the recreation has finished. This blocking behavior might not be acceptable in some applications.

Since filter operators are stateless, it is easily possible to update BE$^+$-trees without blocking the processing of data items. Our idea is to maintain a second BE$^+$-tree for each data stream. This way one BE$^+$-tree can be created concurrently in the background while the other is used for lookup operations. Note that the active BE$^+$-tree is always up-to-date because of its insertion list that maintains all recently added filter operators. As soon as the creation of the other BE$^+$-tree in the background finishes, lookup operations can be answered by the new tree. Because there is no state that must be migrated, the handover can happen between any two data items without losing semantical correctness. This approach is similar to the double buffer technique [Bis94] known from computer graphics. Here, interactive images are rendered using two buffers. One buffer keeps the last frame to be shown on a screen and the other is used for drawing the next frame in the background. When the next frame is complete, the buffers interchange their roles. Thus, there are only complete images displayed. Flicker and tearing do not occur. But in contrast to the double buffer technique, our double tree approach must not be performed continuously in order to save computing resources. Depending on the application, the creation of a new tree in the background can be triggered after a certain amount of data items has been processed or after a certain amount of new filter operators has been added.

11.8 Evaluation

In this section, we present our evaluation of the BE$^+$-tree that comprised a comprehensive set of experiments and benchmarks. The setups of the experiments and benchmarks consisted of synthetic filter EPAs and synthetic events (we made intensive use of our evaluation framework, see Appendix A for details). Only this way all parameters could be fully controlled. But even using synthetic workloads, it was still quite hard to create setups with certain parameter settings. Particularly the setting of the matching rate was difficult to be kept constant because it is influenced by almost all other parameters of an experiment. This problem already occurred in the evaluation of the BE-tree [SJ11, SJ13]. Furthermore, we used BE-trees as well as BE$^+$-trees via the native EP provider of JEPC. Therefore, our results give the real performance of the index structures built into a stream processing engine. The use of the native EP provider also led to the fact that all experiments were purely single-threaded (see Chapter 20 for parallelization in the context of JEPC). Nevertheless, because lookups require only read operations, the BE-tree and the BE$^+$-tree could also process multiple events in parallel without any problems. All experiments and benchmarks were done on a commodity machine with an Intel i7-2600 CPU and 8 GiB of main memory running 64-bit Oracle Java HotSpot VM (1.7.0_13). Because the parameter space is large due to numerous parameters (especially in the case of the BE$^+$-tree), we had to run a lot of experiments and benchmarks. In this section, we report and discuss only the most interesting ones. For the original BE-tree, we could reproduce all results from its evaluation in [SJ11, SJ13]. This not only is a strong point for the original work, but also verifies that our implementation is consistent with the original one.

11.8.1 Globally Optimal Partitioning, Grid Tightening and Avoidance of Empty Cells

BE$^+$-trees are created via bulk loading by default. Then, a globally optimal space partitioning is always achieved. The tightening of grids, the avoidance of empty grid cells, and the splitting of stuck intervals are optional optimization techniques and can be enabled or disabled via parameters independently of each other. Because the interval split technique has several parameters in addition, it is studied in depth in the next subsection. Throughout this subsection, it has been disabled in every configuration of the BE$^+$-tree. For the remaining two optional optimization techniques there is one configuration for every possible combination of them. Table 11.4 shows the four different configurations of the BE$^+$-tree used in the experiments of this subsection.

Parameter	*BE$^+$-tree 1*	*BE$^+$-tree 2*	*BE$^+$-tree 3*	*BE$^+$-tree 4*
`GRID_TIGHTENING`	`FALSE`	`TRUE`	`FALSE`	`TRUE`
`AVOID_EMPTY_CELLS`	`FALSE`	`FALSE`	`TRUE`	`TRUE`
`SPLIT_INTERVALS`	`FALSE`	`FALSE`	`FALSE`	`FALSE`

Table 11.4: Used configurations of the BE$^+$-tree

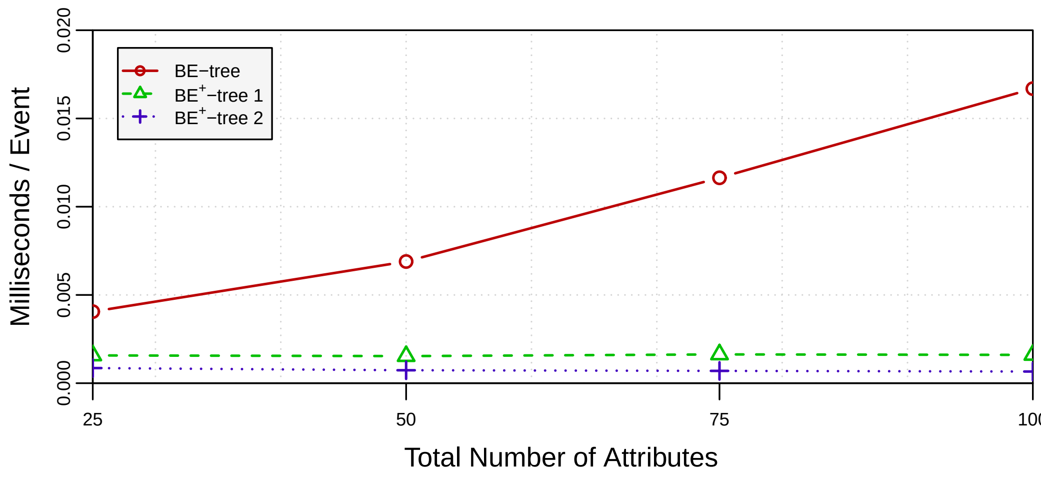


Figure 11.14: Effects of space partitioning and grid tightening

In the first experiments we ran, we used the event stream, the filter conditions and the insertion order of our example EP application (see Section 11.4.1 for details). In total, there were 20,000 filter EPAs. Each filter condition consisted of two *between* predicates that followed the pattern of the example EP application. The predicates were defined so that all intervals of a c-directory had a size of 1,000,000 and were uniformly distributed within [0:2,000,000]. The other parameters were set as follows. All events were uniformly distributed and the matching rate was kept constant at 1 %. The parameters max_{cap} and $min_{support}$ were set to 16 and 8 respectively.

For the BE-tree as well as for the BE$^+$-tree the resulting trees had the overall structures that are presented in Section 11.4.1. At the first level of the BE$^+$-trees the space was partitioned by the first three attributes of the event stream schema and at the second level it was partitioned by all remaining attributes. In contrast, the partitioning of the BE-tree was precisely the opposite. Figure 11.14 shows the event matching times of the original BE-tree and the first two configurations of the BE$^+$-tree for different total numbers of attributes of the underlying event stream. The BE-tree and BE$^+$-tree 1 performed and scaled significantly differently. Recall that both only dif-

	25	50	75	100	∅
BE$^+$-tree 1	0.0015735	0.0015344	0.0016313	0.0016032	0.0015856
BE$^+$-tree 2	0.0008566	0.0007336	0.0006798	0.0006609	0.0007327
BE$^+$-tree 3	0.0014411	0.0013110	0.0013952	0.0014793	0.0014067
BE$^+$-tree 4	0.0008378	0.0006669	0.0006624	0.0006493	0.0007041

Table 11.5: Effect of the avoidance of empty grid cells

fered in space partitioning in this experiment. Already for 25 attributes the globally optimal partitioning of BE$^+$-tree 1 led to a matching performance that was about three times better than the matching performance of the suboptimal partitioning of the BE-tree. While the matching time of the BE$^+$-tree did not change when the number of attributes increased, the matching time of the BE-tree increased noticeably. For 100 attributes BE$^+$-tree 1 performed about ten times better than the BE-tree.

Table 11.5 shows the average matching times in milliseconds per event for all tested configurations of the BE$^+$-tree and for different total numbers of attributes. When the grid tightening optimization was enabled (BE$^+$-tree 2), the performance of BE$^+$-tree 1 could be further improved by factor 2. Although the avoidance of empty grid cells is more an implementation detail, its effect could be noticed. BE$^+$-tree 3 had an average matching time that was 13 % better in comparison to BE$^+$-tree 1. When the avoidance of empty grid cells was used in combination with grid tightening (BE$^+$-tree 4), the matching time was 4 % better in comparison to grid tightening only (BE$^+$-tree 2). The reason why the improvement was less than 13 % this time is that grid tightening influences the internal structure of c-directories. With grid tightening there were simply less empty cells than without it. In fact, the tightening of grids influences not only the avoidance of empty grid cells, but also the interval split technique.

With respect to the examined optimization techniques, the query workload and the insertion order were chosen so that the potential of the optimization techniques could be effectively demonstrated. While the grid tightening and interval split techniques also significantly improve the performance of real-world EP applications, this is not true for the space partitioning of the BE$^+$-tree (see benchmarks in Section 11.8.3). However, very disadvantageous insertion orders of Boolean expressions may occur in practice, particularly if they are created automatically by machines following some pattern (as in our setup). In such cases, the original BE-tree would perform poorly and not scale well with an increasing number of attributes while the BE$^+$-tree would have an optimal space partitioning and, thus, perform and scale well.

Parameter	*Value*	*Parameter*	*Value*
Total number of attributes	20	`AVOID_EMPTY_CELLS`	`TRUE`
Total number of filter EPAs	20,000	`GRID_TIGHTENING`	`TRUE`
Matching rate	1 %	`TOTAL_SPLIT_LIMIT`	100
max_{cap}	16	`GRID_SPLIT_LIMIT`	1
$min_{support}$	8	`SPLIT_FACTOR`	0.0001

Table 11.6: Default configuration used in experiments of interval split technique

11.8.2 Interval Splits

In this subsection, we present our evaluation of the interval split technique of the BE$^+$-tree. We studied not only the effects of this optimization technique, but also how the different parameters influence the resulting BE$^+$-trees and how to set them properly. The default configuration of parameters is shown in Table 11.6. If not stated otherwise, these are the values used in the experiments of this subsection.

By default, the test event stream consisted in total of 20 attributes each of type 32-bit integer number. The total number of filter EPAs was 20,000. With respect to the corresponding query workload, our event generator emitted events so that 1 % of them matched one or more filter conditions. For max_{cap} and $min_{support}$, the default values were 16 and 8 respectively. The BE$^+$-tree optimization techniques which avoid empty grid cells and tighten the grids of c-directories were enabled by default. Each filter condition was allowed to be split at most 100 times in total and at most once per cluster directory by default. Lastly, the default split factor was 0.0001.

For the first experiment, we generated a query workload including only filter conditions that theoretically could be split max_{split} times in order to demonstrate the potential of the interval split technique. In this extreme case scenario, the advantages as well as the disadvantages of the interval split technique showed up at high scale. The filter conditions were generated as follows. Every filter condition had one indexable predicate defined for each event attribute. Only *between* predicates were used, because they allow to generate predicates whose corresponding intervals can be arbitrarily set with respect to size and position. The constant values of the predicates were generated so that the corresponding intervals covered one half of the attribute domain completely, the middle of the attribute domain, and a very small range of the other half. Intervals that covered the left half completely are called *left-handed intervals* and intervals that covered the right half completely are called *right-handed intervals*. Our generator created left-handed intervals and right-handed intervals each with proba-

	Interval splits disabled	*Interval splits enabled*	*Factor*
Matching time	0.7608 ms/event	0.00308 ms/event	247
#Levels per c-directory	1	9.11	9.11
#Cells per #c-directory	1	14.76	14.76
Stored filter conditions	20,000	1,701,276	850
#p-nodes, #p-, #c-directories	20	43,830	2,191
#c-Nodes	21	646,740	28,119
#l-nodes (non-empty)	1	173,136	57,712

Table 11.7: Effects of interval split technique

bility of 50 %. This way, two notable effects were achieved. First, a set of intervals covered the entire domain so that grid tightening could not be performed. Second, at each c-directory the set of input filter conditions was divided into two sets each containing half of the filter conditions on average, when the interval split technique was enabled. One resulting set contained all copies of filter conditions with left-handed intervals for the associated attribute and the other set contained all copies of filter conditions with right-handed intervals. The maximum size of the part of intervals that covered only a small range of one half of the domain was determined by an additional parameter $OverlapSize$ that defined the maximum distance from the middle of the domain. Within the ranges $(middle : middle + OverlapSize]$ and $[middle - OverlapSize : middle)$ respectively, the corresponding endpoints of intervals were uniformly distributed generated. Note that the intervals of this setup can be produced by almost all supported indexable predicates. The only exceptions are the indexable predicates $=$, $\neq$ and $\notin$. For example, applications that filter events on basis of thresholds which are close to the middle of the domain (i.e., close to zero for numeric attribute domains) via the half-open predicates $<$, $\leq$, $\geq$ and $>$ produce such query workloads.

Table 11.7 shows the averaged results for multiple runs of the experiment with $OverlapSize$ set to 5,000. The first column gives the units of measurement. While the second column shows the numbers for a BE$^+$-tree that did not split intervals, the third column shows the numbers for a BE$^+$-tree that did. Although the BE$^+$-trees only differed in whether to split intervals, the numbers of nearly every unit of measurement differed by a large factor that is shown in the fourth column. The first row shows that the interval split technique improved the matching time significantly by factor 247. In the configuration without interval splits, there was absolutely no space clustering. Every c-directory kept all filter conditions in its root cell. But in the setup with interval splits, the c-directories had more than 9 levels and more than 14 non-empty grid cells

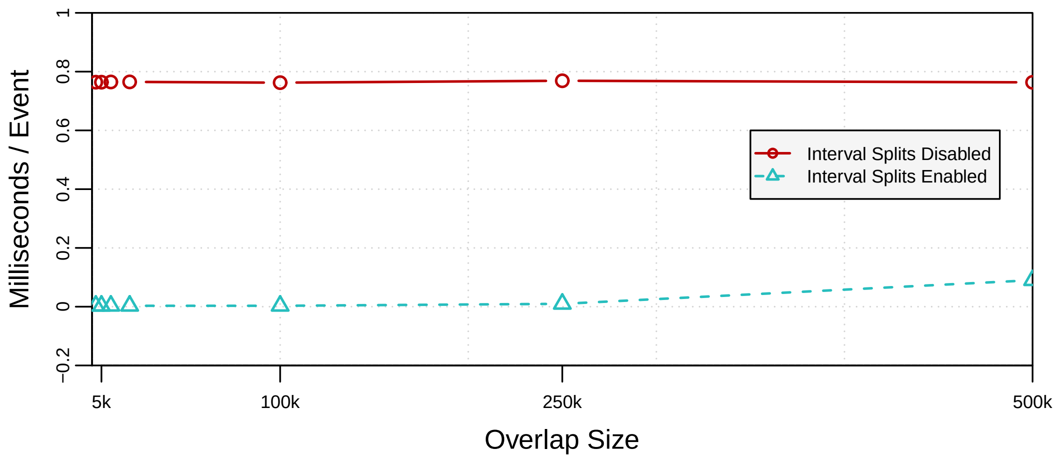


Figure 11.15: Effect of parameter *OverlapSize* on matching time

on average. As already theoretically discussed, the better performance through interval splits comes with additional costs. The BE$^+$-tree that did not split any intervals stored exactly the 20,000 input filter conditions. In contrast, the BE$^+$-tree with interval splits enabled stored about 1.7 million filter conditions. Of course, this larger number of filter conditions in combination with finer space clustering led to significant more tree nodes and directories as shown by the other rows.

Taking this basic setup of the experiment as a starting point, we changed different parameters. First of all, we repeated the experiment for different values of *OverlapSize*. Note that this led to exactly the same effects that would have been occurred if we had changed the split factor instead. Figure 11.15 shows the results for both configurations of the BE$^+$-tree. With interval splits disabled, the average matching time was constant at about 0.76 milliseconds per event. The parameter *OverlapSize* had no effect, because in the chosen setup every interval covered the middle of the root cell of every c-directory and got stuck there. Thus, exactly the same BE$^+$-tree having the same structure and performance was created for every value of *OverlapSize*. Consequently, not only the matching time but also all other values in the second column of Table 11.7 were constant. For the BE$^+$-tree with interval splits enabled, the values were not constant. Instead, the matching time increased slightly when *OverlapSize* was increased. For example, at 250k the average matching time was 0.009699 milliseconds per event (i.e., about three times higher than in the default setup) and at 500k it was already 0.090203 milliseconds per event (i.e., about 30 times higher than in the default setup). However, the BE$^+$-tree with interval splits enabled still performed significantly better than the BE$^+$-tree with interval splits disabled.

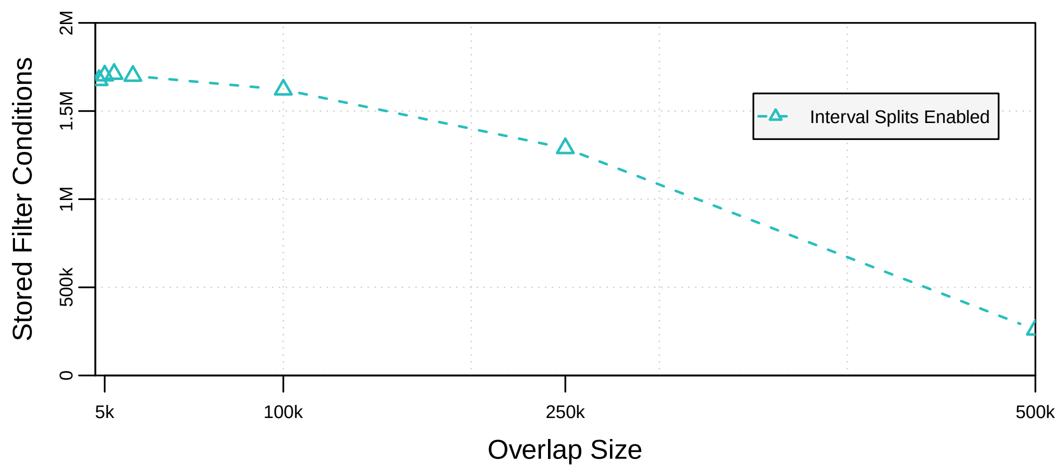


Figure 11.16: Effect of parameter *OverlapSize* on redundancy

For the BE$^+$-tree with interval splits enabled, we also studied other properties than the matching time under changing parameters. At first, we had a special focus on the total number of stored filter conditions as well as on the resolution of the c-directories. Figure 11.16 shows the effect of *OverlapSize* on the redundancy. With increasing values of *OverlapSize* the total number of stored filter conditions decreased noticeably. For example, at 250k there were about 1.3 million filter conditions stored on average and at 500k there were only about 250 thousand filter conditions stored on average. The effect was due to the split factor condition. The higher the value of *OverlapSize* the more stuck intervals were not split, because they violated the split factor condition.

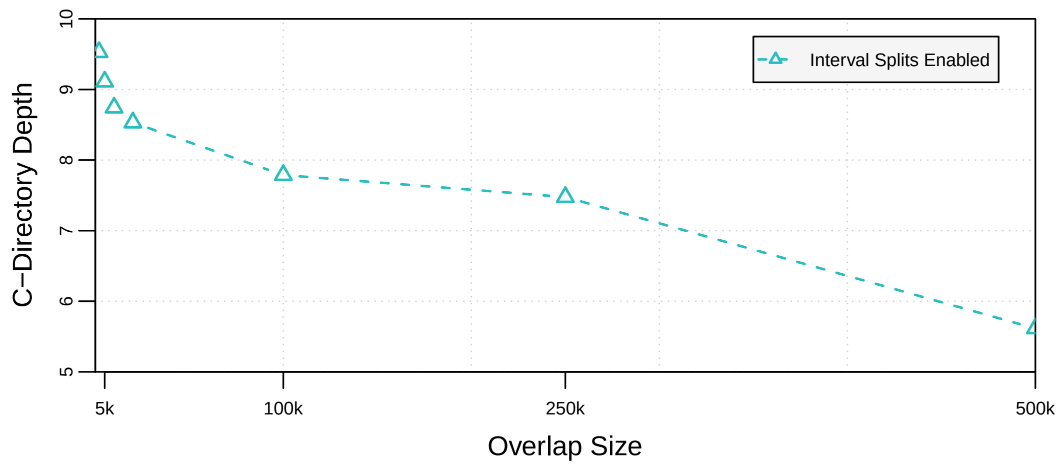


Figure 11.17: Effect of parameter *OverlapSize* on c-directory depth

Figure 11.17 shows the effect on the depth of c-directories. The average depth of the c-directories decreased from about 9.5 levels to about 5.6 levels within the shown range of values of *OverlapSize*. When *OverlapSize* was increased, the average depth of c-directories decreased. Hence, the average matching time increased.

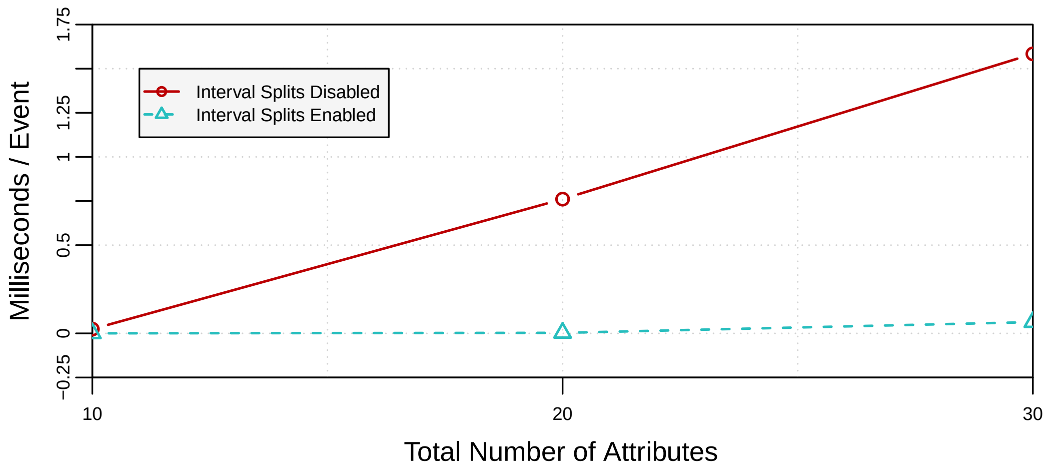


Figure 11.18: Effect of total number of attributes on matching time

In the next experiment, the total number of attributes was changed. Figure 11.18 shows that the matching time of the BE$^+$-tree with interval splits disabled significantly increased with an increasing number of attributes. To study this effect in more detail, Table 11.8 shows the structure of the resulting BE$^+$-trees.

	10 Attributes	*20 Attributes*	*30 Attributes*
#c-Nodes	11	21	31
#p-Nodes	10	20	30
#c-Directories	10	20	30
#l-nodes (non-empty)	1	1	1

Table 11.8: Structures of BE$^+$-trees with interval splits disabled

With interval splits disabled, a resulting BE$^+$-tree was a totally linearly, fully expanded chain of nodes and directories according to Table 11.8. For N attributes, there were $N + 1$ c-nodes, N p-nodes, N c-directories and always exactly one l-node that was not empty. Because every filter condition had predicates on all available attributes, every partition contained all filter conditions. The same was true for cluster directories that only consisted of exactly one cluster. Because the set of filter conditions was never divided (neither through partitioning nor through clustering), the entire set was pushed

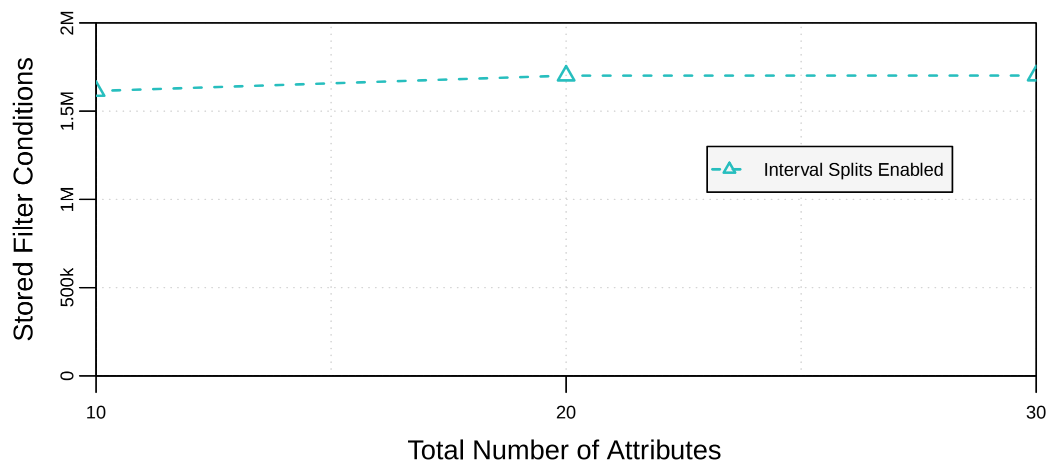


Figure 11.19: Effect of total number of attributes on redundancy

down to the lowest possible level where all filter conditions were stored in a single l-node. When the number of attributes increased, the height of the BE$^+$-tree increased too (in a linear fashion). Every event was routed from node to node until it finally reached the leaf node which stored all filter conditions. Along its entire way through the tree, absolutely no disqualifying filter conditions were detected.

The BE$^+$-tree with interval splits enabled was also influenced by the total number of attributes, but by far less than the BE$^+$-tree with interval splits disabled. Particularly when the number of attributes increased from 20 to 30, an increase in the matching time could be observed (see Figure 11.18). But the reasons for this effect were different from the reasons of the BE$^+$-tree with interval splits disabled. Figure 11.19 shows that the total number of attributes had nearly no impact on the redundancy. In particular, the numbers of stored filter conditions were equal for 20 and 30 attributes. Due to clustering all filter conditions could already been stored in l-nodes at internal levels so that the lower levels of a tree were not created. Therefore, the resulting BE$^+$-trees were almost identical (independent of the number of attributes). Because at each level the number of filter conditions was reduced by a factor of 2 during clustering, there are theoretically only about 9.8 filter conditions per set at the 12th level. Because this was below the defined l-node capacity max_{cap}, no further partitioning and clustering were performed. In our experiment, we actually measured an average size of non-empty l-nodes of 9.8 filter conditions. Therefore, the BE$^+$-trees were only created up to the 12th level (theoretically there can be as many levels as there are attributes). When the number of attributes increased, then the matching time increased too, because for the additional attributes there were no new partitions and clusters created.

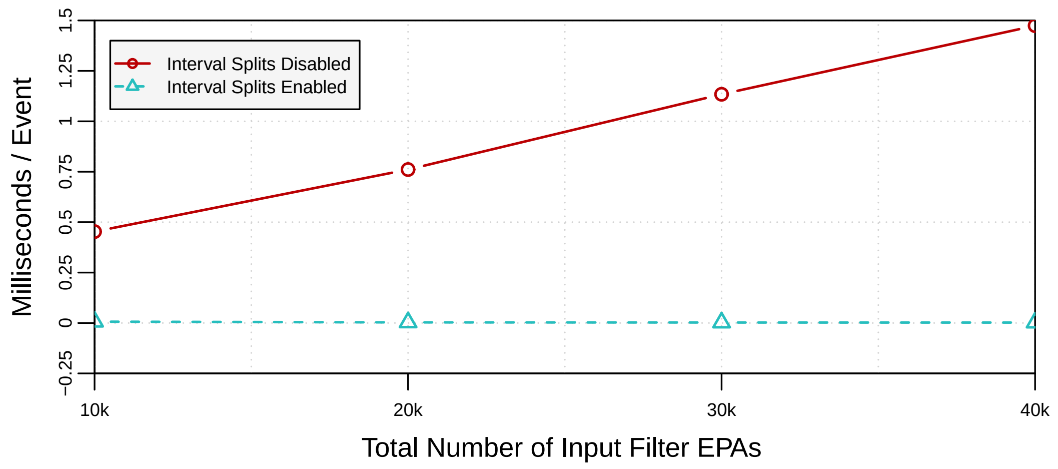


Figure 11.20: Effect of total number of input filter EPAs on matching time

The last setup of this particular experiment comprised changing the total number of filter EPAs. Figure 11.20 shows that the matching time scaled linearly with the total number of filter EPAs in the case of the BE$^+$-tree with interval splits disabled. This result was no surprise, because the resulting trees were linear chains keeping all filter conditions in a single leaf node at the lowest possible level. In the case of the BE$^+$-tree with interval splits enabled, there was no change of the event matching time. This is due to the fact that the resulting trees were better clustered and, thus, widely branched. This experiment showed how important it is for BE-trees and BE$^+$-trees to be able to partition and cluster the space. Moreover, the experiment proved that our interval split technique can significantly improve the quality of space clustering.

As stated by the theoretical analysis, the total number of copies grows exponentially with the number of interval splits in the worst case. However, because an interval split has no negative impact on the matching time (every event being processed visits only one cell at each level of a c-directory and, thus, sees at most one copy of a filter condition), we can generally recommend to perform as many interval splits as possible whenever memory consumption is not an issue. However, memory might be quite limited in certain applications or there might be extremely many event streams that require a query index. In total, there are three possibilities to reduce the number of interval splits. First, the parameter `GRID_SPLIT_LIMIT` can be decreased. This is an option if it is set to two or a greater value and if there are cascading splits possible. Second, `SPLIT_FACTOR` can be decreased. If this parameter is decreased, some splits might no longer be performed because the corresponding intervals violate the split

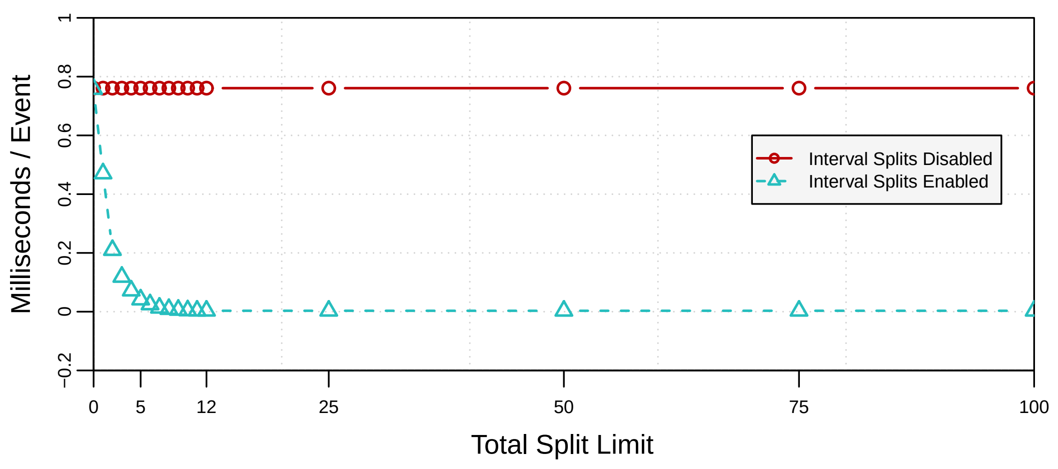


Figure 11.21: Effect of total split limit on matching time

factor condition now. Third, the parameter `TOTAL_SPLIT_LIMIT` can be decreased. This is always an option to reduce the number of interval splits and it is the only option that is independent of the query workload. In the default configuration of the experiment, `GRID_SPLIT_LIMIT` was already set to one and `SPLIT_FACTOR` was chosen extremely small. Therefore, we decreased the parameter `TOTAL_SPLIT_LIMIT` in order to reduce the number of performed interval splits.

The parameter `TOTAL_SPLIT_LIMIT` was set to 100 in the default configuration. This value allowed many more interval splits than possible. Figure 11.21 clearly shows that reducing the total split limit down to 12 had no effect on the matching time. As explained on page 134, the sets of filter conditions became so small during bulk loading that they could be stored completely in l-nodes at the 12th level. Therefore, every additionally allowed interval split could not be performed and, thus, influence the resulting tree. But from this point on, a further reduction of the total split limit led the matching time increase rapidly. For a total split limit of zero, exactly the same tree was created that would have been created if interval splits had been disabled. Of course, then the performances of both configurations should be identical, which is shown by the graphs. The graphs also reveal that the very first interval splits (that are located at the topmost part of a tree) gave the highest improvement in performance. Therefore, we recommend to enable the interval split technique also in case of limited memory. But then the number of interval splits should be severely limited. We also studied the redundancy in this experiment. Figure 11.22 shows that the redundancy increased in the same way as the matching time decreased. The figure also verifies that no more than 12 splits per filter condition were performed on average.

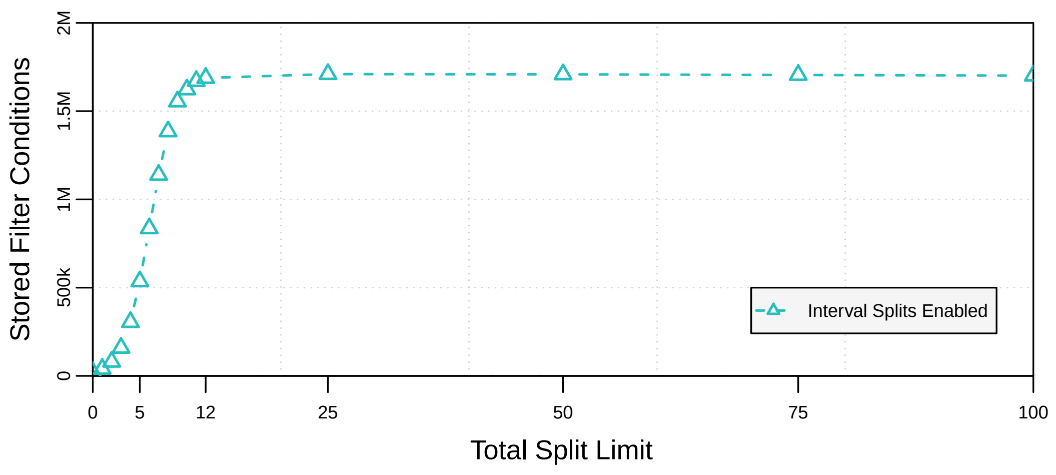


Figure 11.22: Effect of total split limit on redundancy

Up to now, all experiments of this section had a specially prepared synthetic query workload in order to investigate the interval split technique in an extreme case scenario. In particular, every interval got stuck in the root cell and did not cover a large range of the attribute domain. The experiments presented in the following had the goal to simulate real-world applications by dealing with intervals whose positions and sizes were randomly chosen. By default, a specific setup was as follows. Every filter condition had an indexable *between* predicate on every available attribute. All attributes were of the type 32-bit integer number. Again, we considered a set of filter conditions that could not be divided via space partitioning, because every filter condition restricted all event attributes. We explicitly wanted to study the effects of the interval split technique only (for more realistic workloads see our benchmarks in Section 11.8.3 which drew a random subset of all available attributes for the placement of predicates). The constant values of the *between* predicates were derived from randomly generated intervals. In particular, the sizes of those intervals were randomly chosen from the range $[2 : 2^{31}]$. Note that the maximum possible interval size of 2^{31} is half the size of the 32-bit integer number domain. The positions of the intervals were randomly distributed across the entire domain. As a consequence, grid tightening was not possible at all or had only little effect. In total, there were three probability distributions involved for generating the random intervals. First, the probability distribution for generating the sizes of intervals. Second, the probability distribution for generating the positions of intervals. Third, the probability distribution for generating the values of events. In our first setup, all generators used the normal distribution (i.e., the preset `NORMAL`; see Appendix A for detailed parameter settings).

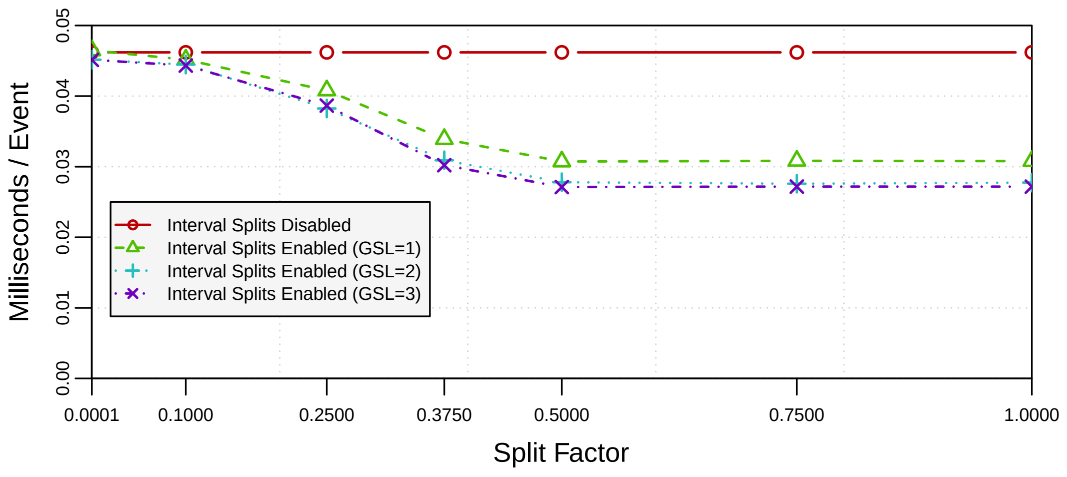


Figure 11.23: Effects of split factor and grid split limit on matching time

Figure 11.23 shows the matching performance for different split factors and different grid split limits (GSL). As theoretically discussed, a split factor higher than 0.5 does not speed up the matching time in general, because it is not guaranteed that every split is able to push down at least one part by more than one level. The results clearly show that at 0.5 the optimum was reached for the generated query workloads. Note that this was true not only for single allowed splits per c-directory (GSL=1), but also for multiple allowed splits per c-directory (GSL=2 and GSL=3). We also got the same result for almost all other tested probability distributions. Therefore, a value higher than 0.5 is definitively not recommended in general. In comparison to allowing one split per c-directory, the performance only slightly improved when GSL was set to two. And from GSL=2 to GSL=3 there was no further improvement at all.

Although the matching time was constant for a split factor higher than 0.5, the redundancy substantially increased. Figure 11.24 shows the degree of redundancy for each configuration of the experiment. As expected, a reduced matching time led to an increased degree of redundancy. For all configurations with GSL=1 and for all configurations that had a split factor smaller than or equal to 0.5, the degree of redundancy was relatively moderate, controllable and predictable. But for larger split factors it escalated. For `SPLIT_FACTOR` set to 1.0 (i.e., every stuck interval was split if the split limits had not been reached yet) and `GRID_SPLIT_LIMIT` set to 2 there were more than 9 million filter conditions stored in a tree and for `GRID_SPLIT_LIMIT` set to 3 even more than 220 million. However, for split factors limited to the range $[0 : 0.5]$ redundancy and matching performance could be well balanced.

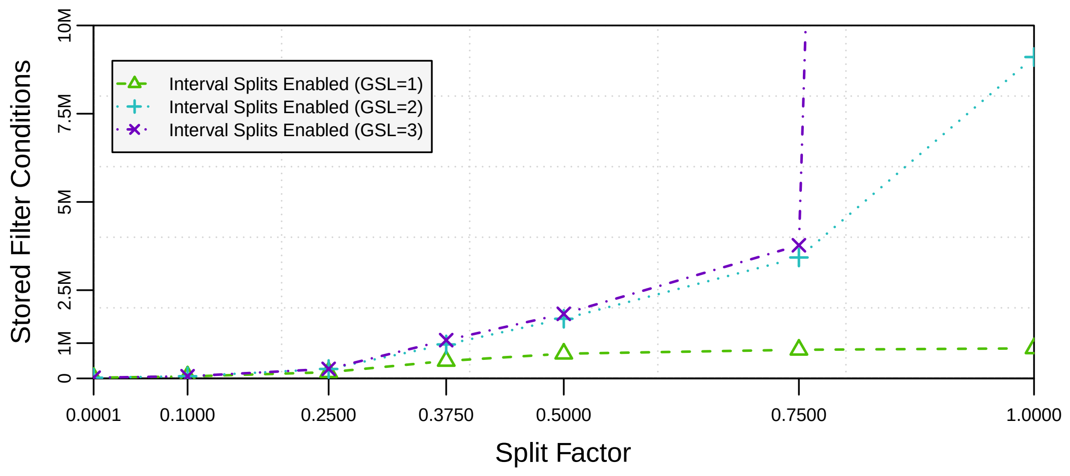


Figure 11.24: Effects of split factor and grid split limit on redundancy

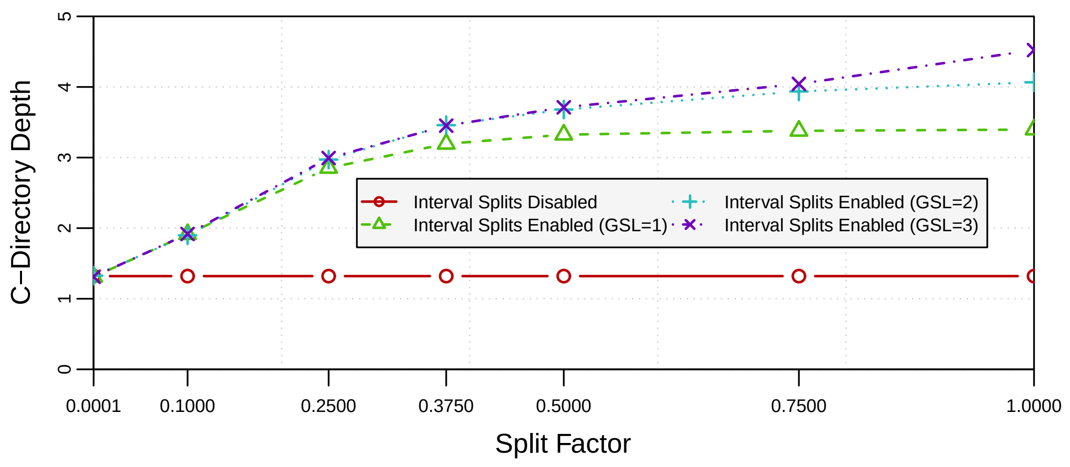


Figure 11.25: Effect of split factor on c-directory depth

For this particular setup of the experiment, we also report other results in the following. All observed effects occurred not only for normally distributed workloads, but also all other tested probability distributions. Because the intensities of the effects varied only slightly, we discuss the effects only for this setup. Figure 11.25 shows the average c-directory depths. Again, a higher degree of redundancy led to better developed grids. However, in the case of split factors greater than 0.5, the rapidly growing redundancy only slightly increased the average depth of c-directories. The figure also shows the average c-directory depths for BE-$^{+}$-trees with interval splits disabled. In contrast to the previous experiments, not all intervals got stuck in the root cell this

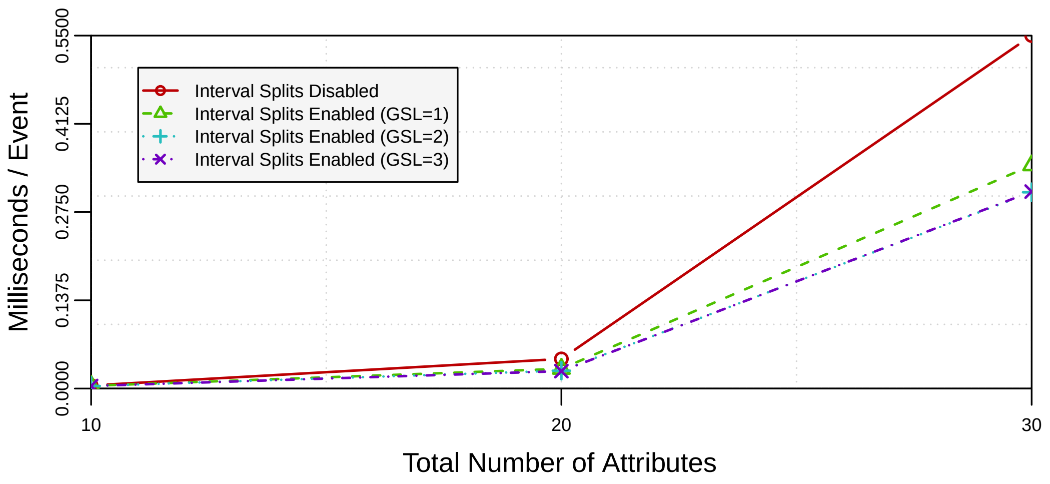


Figure 11.26: Effect of total number of attributes on matching time

time. On average, a c-directory had a depth of 1.32 levels. However, space clustering was still quite limited. This shows that the BE-tree as well as the BE$^+$-tree with interval splits disabled are highly sensitive for normally distributed intervals, because many intervals get stuck in the root cell. These results have not been reported before, because normally distributed intervals were not considered in the original evaluation of the BE-tree [SJ11, SJ13].

If we compare Figure 11.23 with Figure 11.24, again it becomes clear that the very first interval splits resulted in the highest performance improvements. In order to achieve the same intensity after the first few interval splits again, many more intervals had to be split in addition. Fortunately, the bulk loading of BE$^+$-trees is performed top-down. This means, all interval splits are also performed top-town until the lowest level or the maximum number of allowed interval splits has been reached. Therefore, all allowed interval splits are always performed in the topmost part of a tree where they have the highest beneficial impact.

Figure 11.26 plots the matching time as a function of the total number of attributes. All configurations of the BE$^+$-tree had a significantly increased matching time when the number of attributes increased. This effect was due to the fact that there was no space partitioning possible as being discussed on page 133. Figure 11.27 shows the matching time for different total numbers of filter EPAs. Note that BE-trees scale linearly with the total number of matched filter EPAs [SJ11, SJ13]. Thus, all shown configurations scaled equally. But the configurations with interval splits enabled were superior with respect to the absolute matching time.

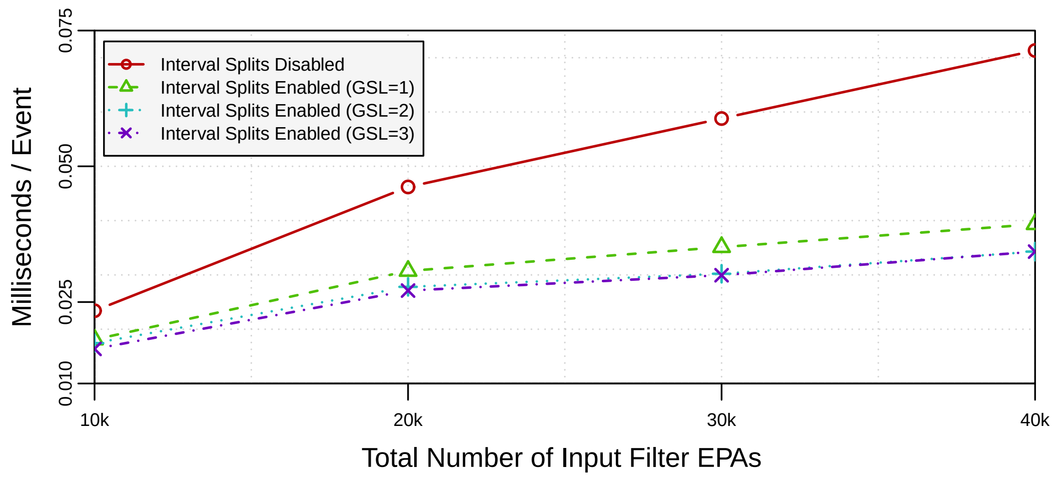


Figure 11.27: Effect of total number of input filter EPAs on matching time

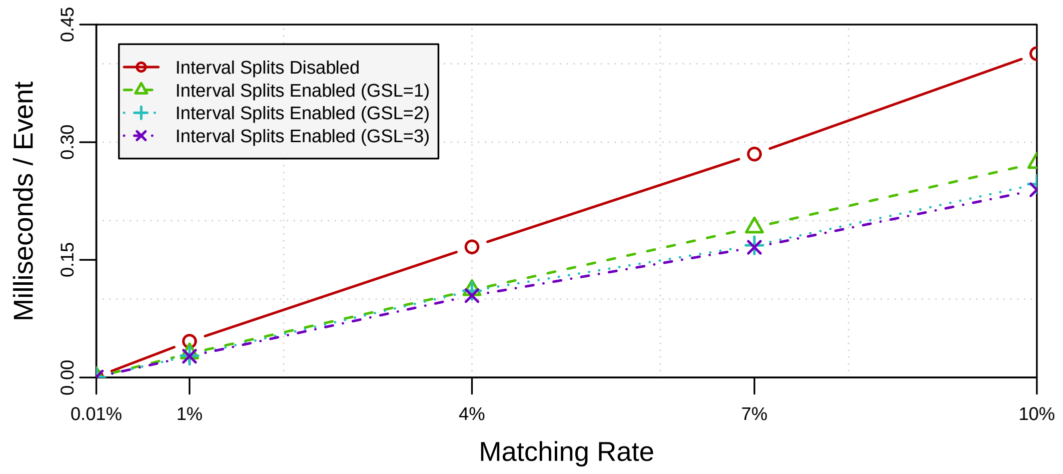


Figure 11.28: Effect of matching rate on matching time

Figure 11.28 shows that all configurations of the BE$^+$-tree performed similarly when the matching rate changed. Altogether, the matching times of all configurations of the BE$^+$-tree increased linearly with an increasing matching rate. But again, the BE$^+$-trees with the interval split technique enabled achieved significantly lower absolute matching times. This result is also consistent with the evaluation of the original BE-tree. For higher matching rates there were relatively less mismatching filter EPAs that could have been skipped. So, the linear increase of the matching times is expected.

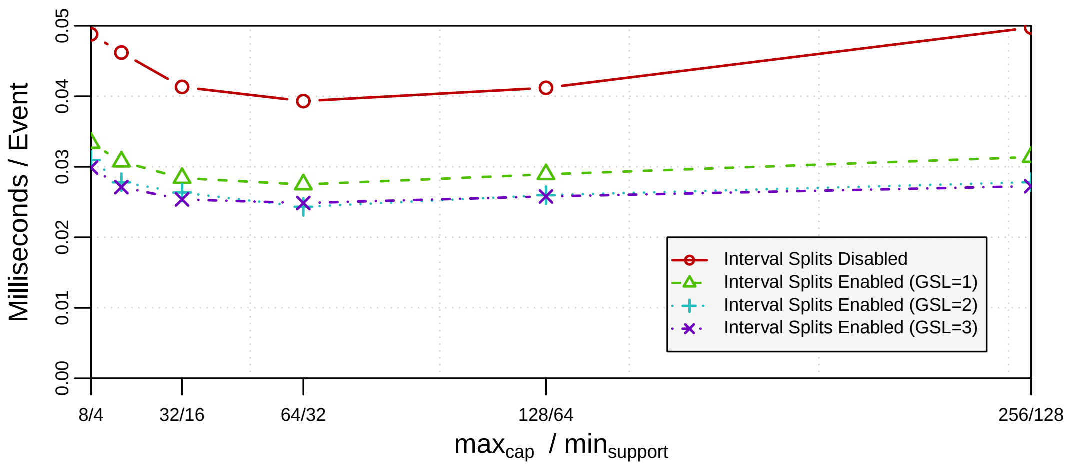


Figure 11.29: Effects of max_{cap} and $min_{support}$ on matching time

Figure 11.29 shows the matching time for different values of max_{cap} and $min_{support}$. On the x-axis, the first number gives the value of max_{cap} and the second number gives the value of $min_{support}$ used in the corresponding runs of the experiment. The parameter $min_{support}$ was always set to $0.5 * max_{cap}$ so that an overflowing l-node was partitioned only if at least half of the filter conditions could be moved into a new partition. Again, this experiment not only confirmed the original BE-tree evaluation, but also showed that the parameters influence BE$^+$-trees exactly in the same way as BE-trees. The matching time changed slightly when the parameters were changed. As proposed by the original work on the BE-tree, these parameters can be used to fine-tune the matching performance. Obviously, our used default values 16/8 were not optimal but close to. Because their optimal settings depend also on other parameters as in the case of BE-trees, these parameters can be used to fine-tune BE$^+$-trees too.

As already stated, we also tested workloads generated on basis of other probability distributions without getting other results. Exactly the same effects appeared, but of course with different intensities. The next figures show the results for workloads which had uniformly distributed interval sizes and positions as well as uniformly distributed events. Because uniformly distributed intervals get stuck in the root cells less often than normally distributed intervals, less interval splits occurred. Therefore, we observed a much lower degree of redundancy in this experiment (see Figure 11.31). The BE$^+$-tree without interval splits benefited from the uniform distribution. This was proven by a significantly higher depth of c-directories (see Figure 11.32). Even though there were less interval splits and the basic space clustering was of higher quality, the BE$^+$-tree with interval splits enabled was still superior (see Figure 11.30).

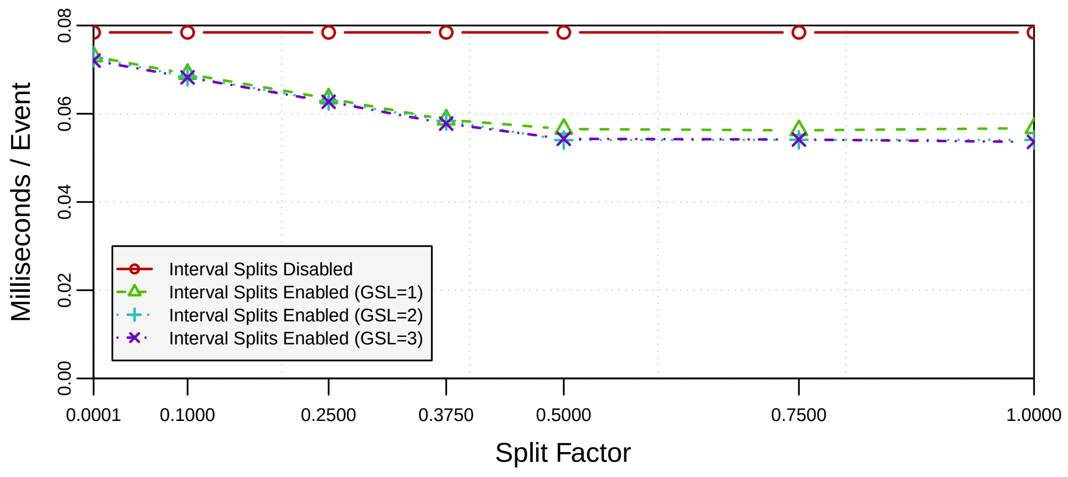


Figure 11.30: Effect of split factor on matching time

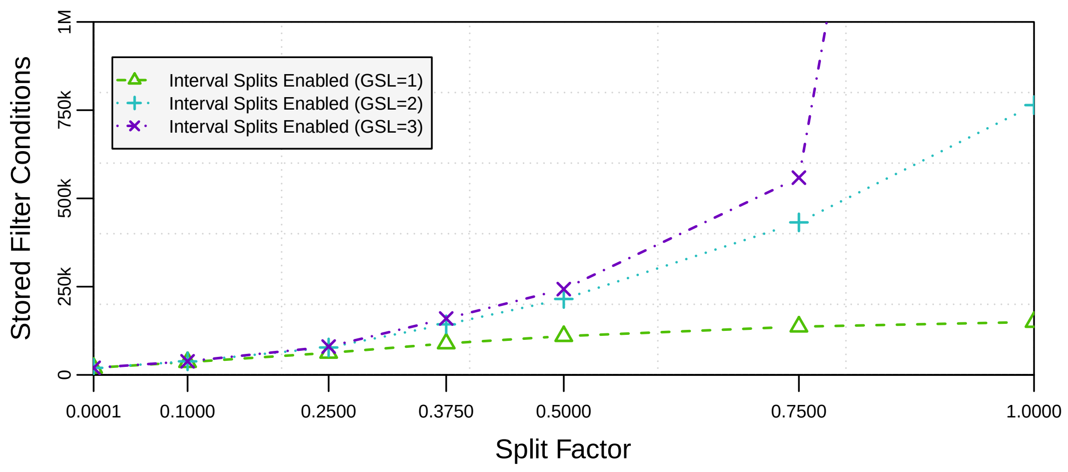


Figure 11.31: Effect of split factor on redundancy

This particular experiment again confirmed that the very first interval splits are the most effective ones. For the parameter `SPLIT_FACTOR` set to its default value 0.0001 there were only up to 20,015 filter conditions stored among all runs of the experiment. This means that due to interval splits the sizes of the BE$^+$-trees increased by only up to 15 additional filter conditions which is less than 0.1 %. But there already was a significantly lower event matching time as shown in Figure 11.30. Because this was true in all our experiments, we chose the very small number of 0.0001 as default value for `SPLIT_FACTOR`. This setting noticeably improves the matching performance for real-world workloads, but does not lead to high redundancy.

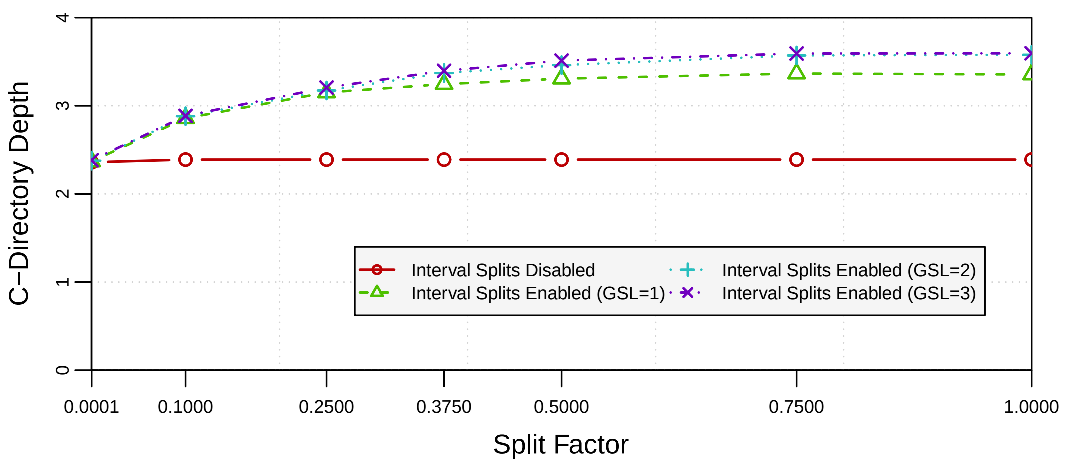


Figure 11.32: Effect of split factor on c-directory depth

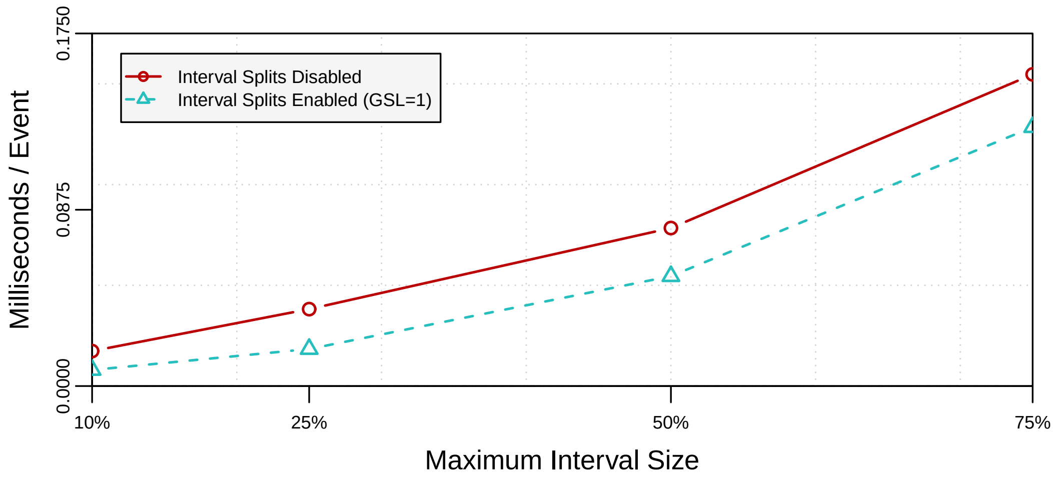


Figure 11.33: Effect of interval size on matching time

For the workloads generated on basis of uniform distributions, we report the results of an experiment that was not presented yet. In this experiment, we examined the matching time for different average interval sizes. For this purpose we extended and reduced the value range $[2 : max_{intervalSize}]$ from which the sizes of intervals were drawn under uniform distribution. While the minimum value was kept at 2, the maximum value $max_{intervalSize}$ was changed. The x-axis in Figure 11.33 gives the ratio of the maximum interval size to the size of the entire domain. Obviously, smaller intervals can be indexed better than larger ones. Large intervals naturally have less potential to prune the space than small intervals.

To complete the presentation of the evaluation of the interval split technique, we derive some general advices on basis of the outcomes of the experiments. The good news is that there is virtually no wrong configuration of parameters with respect to matching performance. But recall that a very aggressive configuration may result in a high degree of redundancy. To achieve optimal matching performance while uncontrolled redundancy is avoided, parameters must be fine-tuned according to the workload. However, it is possible to recommend a basic configuration of parameters which comes relatively close to the optimal configuration for most real-world workloads. This basic configuration is as follows. The parameter `SPLIT_FACTOR` should be set to 0.5. Then, it is guaranteed that every advantageous interval split is performed and every interval split that potentially only increases redundancy is not performed. The parameter `GRID_SPLIT_LIMIT` should be set to one. As the experiments showed, an increased value led to high redundancy very often and not always reduced the matching time. In contrast, the parameter `TOTAL_SPLIT_LIMIT` should be set large enough. To be sure to not prevent beneficial interval splits, it should be set to any value that is higher than the total number of attributes.

11.8.3 Benchmarks

In this section, we compare the performances of the BE-tree and the BE$^+$-tree for randomly generated workloads that consisted of 50 attributes and 50k filter EPAs by default. We tested a great number of probability distributions and different combinations of them. Here, we report the results that are most important or cover a wide range of real-world applications. For the interval split technique the recommended configuration was used instead of the more conservative default configuration. Recall that in the first part of the evaluation every filter condition always consisted of two predicates following a certain pattern and that in the second part every filter condition had a predicate on each attribute. In this last part, attributes were selected randomly per filter condition according to a specific probability distribution. Besides the uniform distribution, we also report the results for normal and exponential distributions that model many real-world applications in which some hot attributes appear in many filter conditions while other attributes are almost never restricted. The sizes and positions of intervals as well as events were randomly generated as in the experiments before. The range from which interval sizes were drawn was fixed so that an interval covered between 2 % and 25 % of the entire 32-bit integer domain. For the uniform and the normal distribution this led to an average interval size of 12 %. Note that this value was the default value in the evaluation of the original BE-tree [SJ11, SJ13].

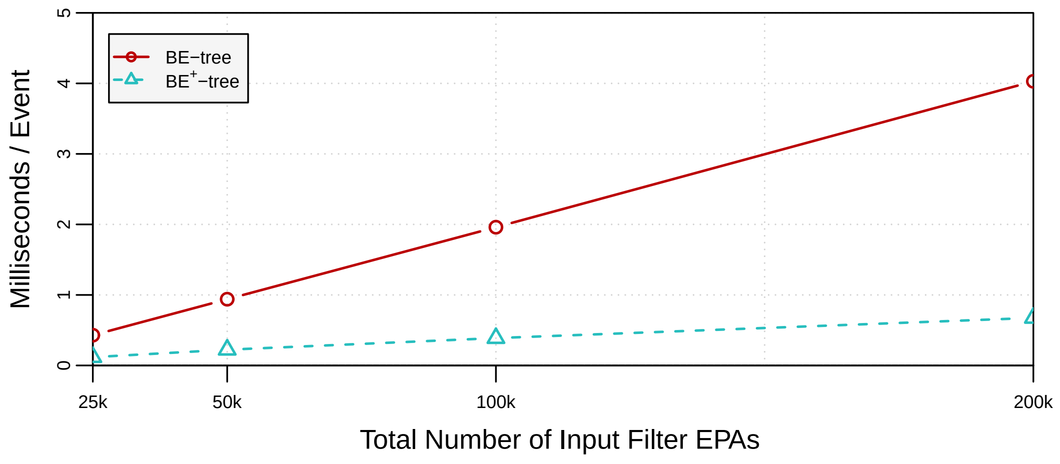


Figure 11.34: Effect of total number of input filter EPAs (NORMAL distributions)

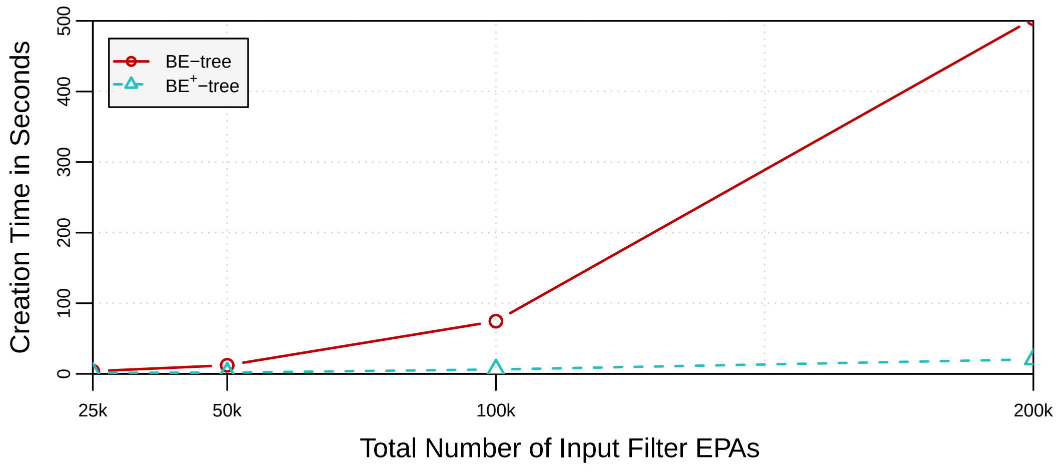


Figure 11.35: Creation times (NORMAL distributions)

The results we present first are for a setup in which all generators (i.e., the generators for attribute selection, interval sizes, interval positions, and event positions) followed the normal distribution. Note that the insertion order of filter EPAs was according to their creation order and, thus, random in all following experiments. Figure 11.34 shows the matching time for different total numbers of filter EPAs. The BE$^+$-tree clearly outperformed the BE-tree by many times. Also for all other tested setups of the experiment, the BE$^+$-tree was always superior to the BE-tree. Differently configured generators only led to different distances between the resulting graphs. We present the worst cases (for the BE$^+$-tree) later in this section.

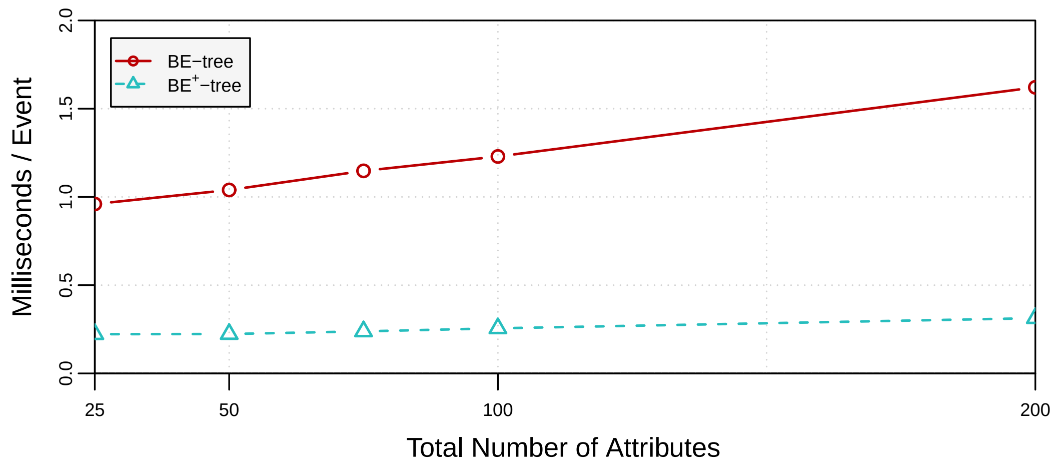


Figure 11.36: Effect of total number of attributes (NORMAL distributions)

We also measured the total time that was needed to create an entire query index from scratch for a given set of filter EPAs. A typical property of bulk loading techniques for index structures is a significantly better creation time. While a BE$^+$-tree can be created top-down in one go, a BE-tree must be created by calling the insertion algorithm N times for N filter EPAs. Figure 11.35 shows the creation times of both approaches. The bulk loading of the BE-$^+$-tree scaled substantially better than the query-by-query insertion of the BE-tree. Therefore, even for large sets of filter EPAs it is not costly to (re-)create a BE$^+$-tree. Note that the BE$^+$-trees in this experiment had to cope with many more filter expressions that had to be stored, because the interval split technique was always enabled. For 200k filter EPAs, the corresponding BE-trees stored exactly 200k filter conditions, but the corresponding BE$^+$-trees stored more than three million filter conditions on average. Despite this fact, the creation times of the BE$^+$-trees were substantially better than those of the original BE-trees.

Next, we present and discuss the matching time for different total numbers of attributes. Figure 11.36 shows that the BE$^+$-tree performed significantly better than the original BE-tree. However, both were relatively insensitive to the total number of attributes. The slope of the graph of the BE$^+$-tree is quite similar to the slope of the graph of the original BE-tree. This fact proves that the original BE-tree produced a good space partitioning for a random insertion order of filter EPAs. As mentioned earlier, the globally optimal space partitioning of the BE$^+$-tree is a less effective optimization technique compared to the interval split technique. It has a crucial effect only if the insertion order of filter conditions is disadvantageous.

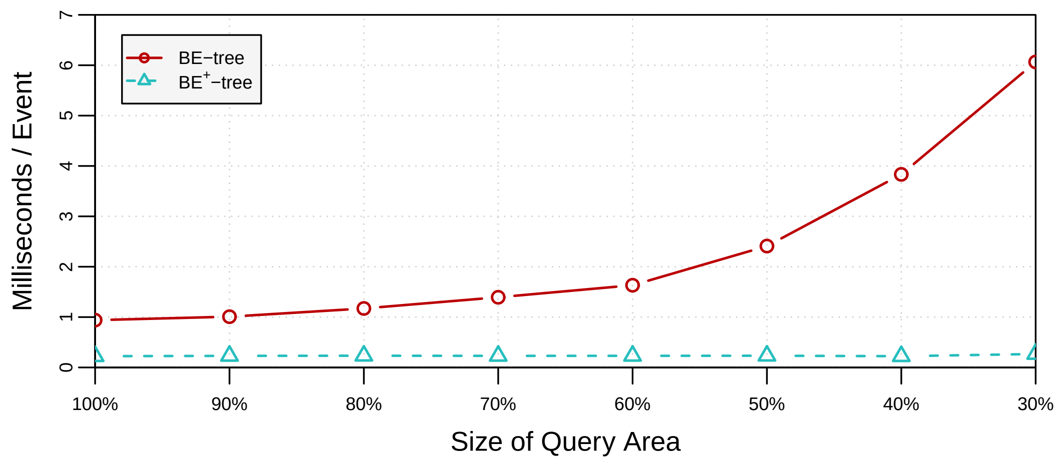


Figure 11.37: Effect of query area size on matching time (NORMAL distributions)

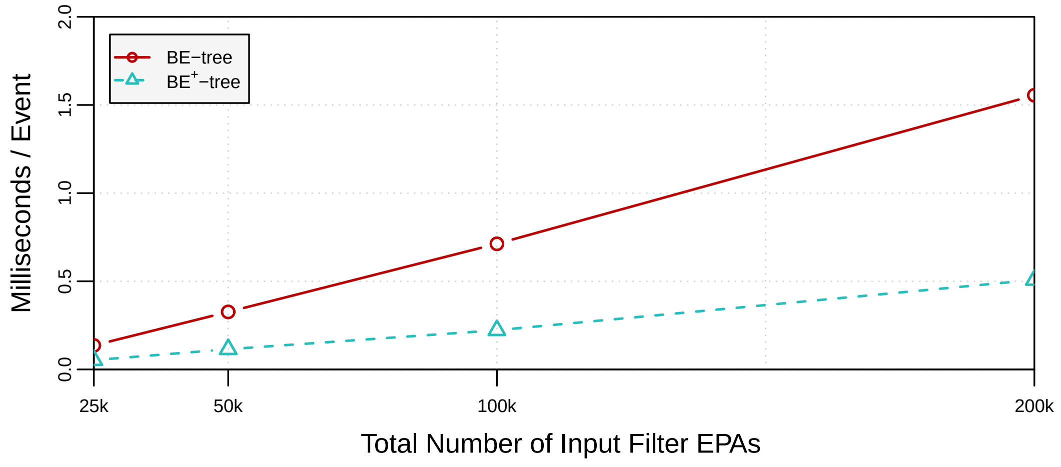


Figure 11.38: Effect of total number of input filter EPAs (UNIFORM distributions)

The next experiment that we present and discuss is only reported for this particular setup of the generators, because it held also for all other setups. Up to now, intervals were distributed across the entire domain of their associated attribute. In this experiment, the area of the domain that was covered by all intervals (query area) was successively shrunken symmetrically. Figure 11.37 shows the effects on the matching time. The labels on the x-axis give the ratio of the size of the query area to the size of the entire domain. It shows that the performance of the original BE-tree decreased significantly when the intervals covered only a smaller part of the domain. However, due to grid tightening and interval splits the performance of the BE$^+$-tree kept constant.

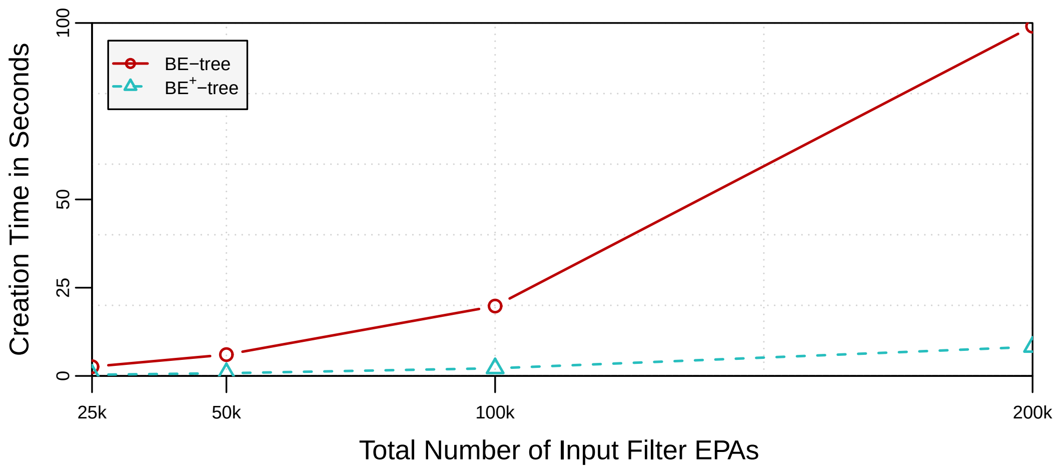


Figure 11.39: Creation times (UNIFORM distributions)

The next figures show the results of the same experiments but for slightly differently generated workloads. All generators that created the workloads followed the uniform distribution this time. Besides the normal distribution, the uniform distribution is one of the most important and most widely used probability distributions to describe and model real-world applications. Because there occurred no new effects, we do not discuss the results in detail. Figure 11.38 shows the matching time for different total numbers of filter EPAs. Also in this setup the BE$^+$-tree was superior to the original BE-tree. But note that the distance between the two graphs is smaller than in the setup before. This is because there got significantly less intervals stuck. Consequently, there were less interval splits possible (and necessary) so that the original BE-tree had a performance that came closer to the performance of the BE$^+$-tree. As a side effect, the degree of redundancy of the BE$^+$-tree was considerably lower this time.

Figure 11.39 gives the corresponding creation times for both the BE-tree and the BE$^+$-tree. At a first glance, there is nothing new to discuss. Therefore, creation times are not reported anymore, because the difference between the BE-tree and the BE$^+$-tree and the scaling were the same for every tested setup. However, it should be mentioned that the concrete numbers of the BE$^+$-tree were almost identical to the concrete numbers of the BE$^+$-tree in the setup before. It is somewhat surprising that the creation times of the BE$^+$-trees were almost identical, because the workload of the previous setup led to many more interval splits and, thus, significantly more copies of filter conditions that had to be stored in addition. This means, the total number of interval splits has only little impact on the creation time.

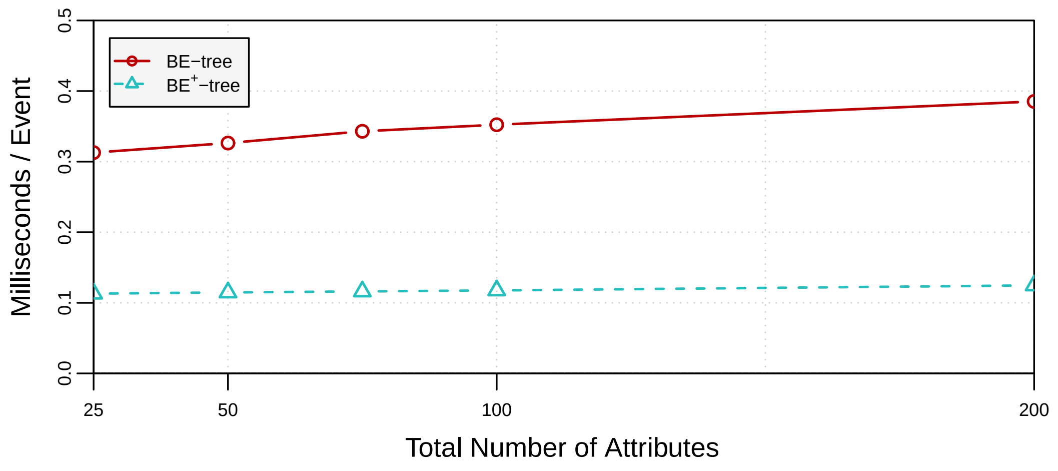


Figure 11.40: Effect of total number of attributes (UNIFORM distributions)

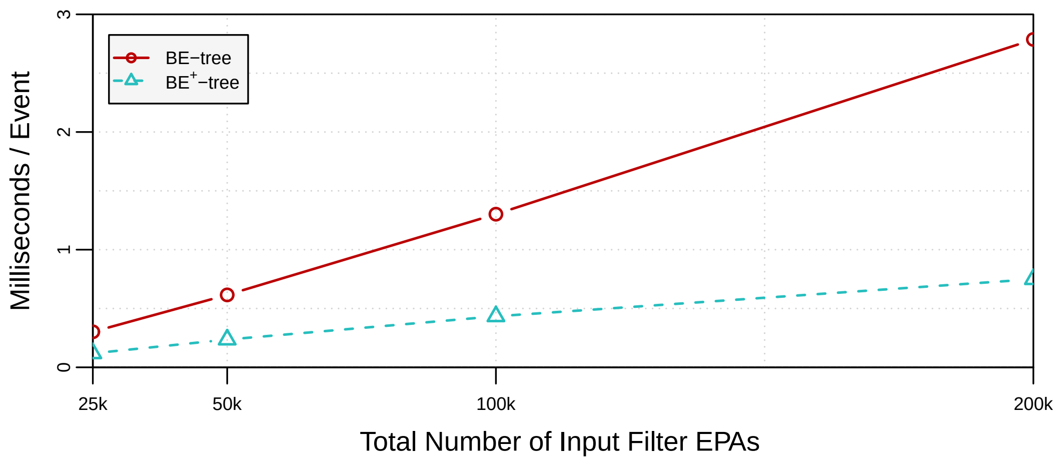


Figure 11.41: Effect of BETA1 distributed interval sizes

Similar to the results of the previous setup, Figure 11.40 shows that both the BE-tree and the BE$^+$-tree were again relatively insensitive to the total number of attributes and that the BE$^+$-tree achieved again a substantially better matching time than the original BE-tree. Once more, a random insertion order of filter conditions led to a good space partitioning in the case of the original BE-tree. The experiment showed that exchanging the probability distribution of the generator that selects the attributes on which a filter condition places indexable predicates had no significant effect. For this reason, the attributes on which a filter condition places indexable predicates were selected on basis of the normal distribution in all following experiments.

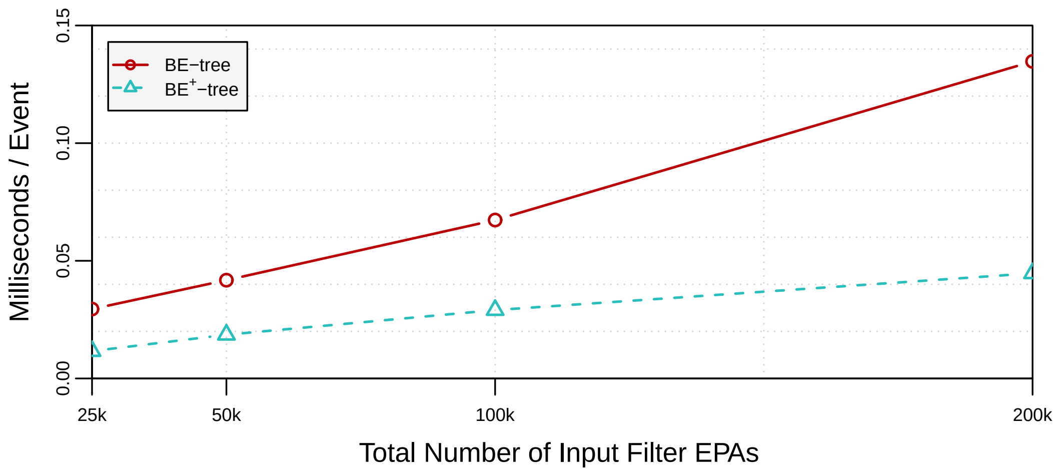


Figure 11.42: Effect of CHI_SQUARED1 distributed interval sizes

Even though the exchange of the probability distributions of the other workload generators did not lead to an overall outcome that was different from the main results presented so far, the final figures show the matching times for probability distributions that are quite different from the normal and the uniform distributions. In particular, we present the results of four different setups in which the positions or the sizes of intervals were generated on basis of the `BETA1` or the `CHI_SQUARE1` distributions (see Section A.3.1 in the appendix for details), while all other generators followed the normal distribution. We chose these particular distributions for presentation, because they are different from bell-shaped probability distributions and often the underlying distributions in real-world applications [Gra94].

The next two figures (Figure 11.41 and Figure 11.42) show the matching times for setups in which the underlying distribution of the generator of interval sizes has been exchanged. Figure 11.41 gives the results for workloads with `BETA1` distributed interval sizes. In these workloads, half of the intervals were tiny and the other half of intervals were huge on average. As shown by Figure 11.41, the performance of the BE$^+$-tree was almost identical to its performance in the setup with normally distributed interval sizes. This was because tiny intervals are per se good to index by all types of BE-trees (tiny intervals tend to not get stuck at the upper levels of c-directories) and huge intervals can be handled via the interval split technique. But the performance of the BE-tree improved significantly in comparison to its performance in the setup with normally distributed interval sizes. In this experiment, the created BE-trees had a much better space clustering because of the tiny intervals.

Figure 11.42 shows the matching times for workloads with `CHI_SQUARED1` distributed interval sizes. Note that the distribution `CHI_SQUARED1` is a special case being identical to the exponential distribution. Almost all intervals were of tiny size in this experiment. The previous experiments showed that the interval split technique is the most effective optimization technique of the BE$^+$-tree. But because there were almost all intervals tiny, they got not stuck very often and, thus, interval splits were neither possible nor necessary very often. The BE-tree benefited from the tiny intervals and its graph comes close to the graph of the BE$^+$-tree therefore. However, the BE$^+$-tree still outperformed the BE-tree in this disadvantageous setup.

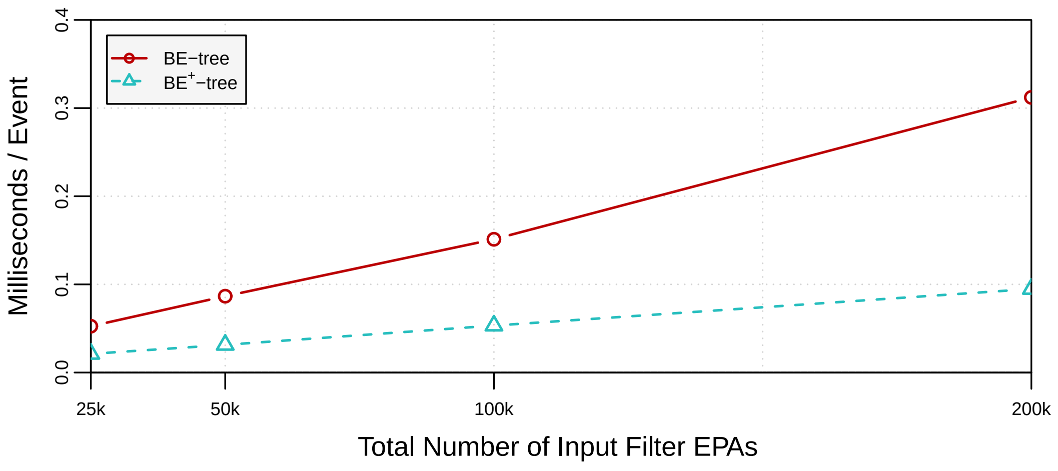


Figure 11.43: Effect of BETA1 distributed interval positions

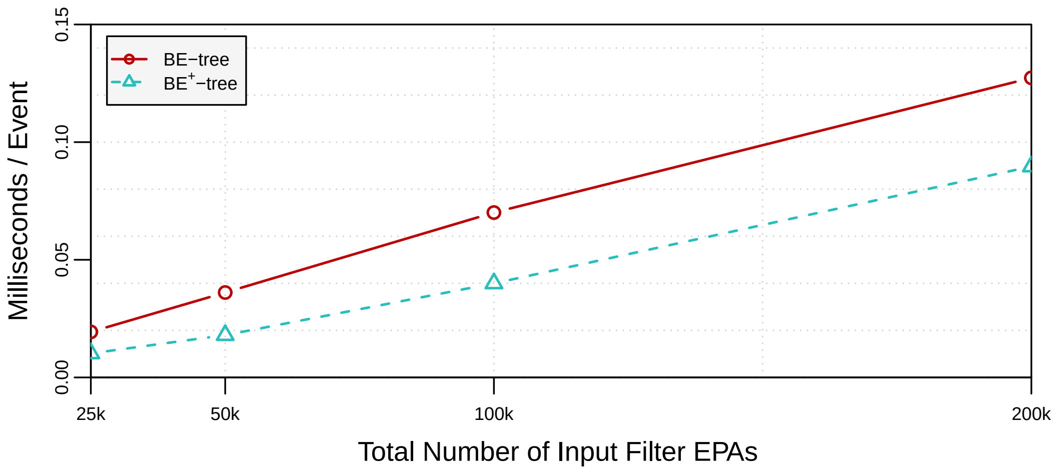


Figure 11.44: Effect of CHI_SQUARED1 distributed interval positions

The last two figures plot the matching times for workloads in which the positions of intervals were generated according to other probability distributions. Figure 11.43 gives the results of the setup with `BETA1` distributed interval positions. Here, there is nothing new to discuss. Figure 11.44 shows the matching times of the setup with `CHI_SQUARED1` distributed interval positions. In this setup, almost all intervals were located at the left border of the domain. This particular setup was the one among all tested setups for that the difference between the BE$^+$-tree and BE-tree was smallest. At a closer look, the original BE-tree took full advantage of the `CHI_SQUARED1` distributed intervals. Because almost all intervals where at the left border of the domain and had a size of at most 25 % of the domain size, they did not get stuck at the top levels in the case of the BE-tree. However, the BE$^+$-tree still outperformed the BE-tree. For 200k filter EPAs the BE$^+$-tree achieved a 75 % faster matching time.

The benchmarks proved that the BE$^+$-tree achieves a substantially better matching performance than the BE-tree in virtually every situation. Applications with no or infrequent index updates and very high data rates should always prefer the BE$^+$-tree therefore. Note that we used our recommended values of parameters belonging to the interval split technique in all benchmarks. The BE-tree has only two parameters for fine-tuning that are also parameters of the BE$^+$tree. But the BE$^+$-tree offers several more options for further tuning its performance according to a specific workload.

11.9 Related Work

Of course, previous work on BE-tree [SJ11, SJ13] and GESS [DS01] is related to the BE$^+$-tree. But because BE-tree and GESS already have been presented and discussed throughout this chapter, they are excluded from this section. Instead, we discuss other important indexing approaches. Note that those were not considered in our benchmarks, because already the BE-tree is superior to all approaches proposed so far. The following approaches are not competitive in at least one important category and not discussed in detail therefore. The count-based index Propagation [Fab01] is designed for event matching in publish/subscribe systems. However, this approach does not support multi-valued predicates that restrict an attribute to multiple possible non-contiguous values. This low expressiveness is unacceptable for many of today's applications. The tree-based index structures for ranked publish/subscribe systems proposed in [Mac08] do scale very poorly with the total number of attributes. But scalability of an index is of utmost importance for most applications.

SIFT. The work presented in [YG94] introduces and discusses several approaches, called *SIFT methods* in the following, on basis of the inverted list data structure [ZM06] for matching profiles submitted to a document database against incoming documents. This work laid the foundation for subsequent work done in the area of event matching and is mentioned here therefore. The presented approaches comprise count-based methods as well as tree-based methods. All SIFT methods only support equality predicates (and via an extension also negation predicates), which is sufficient for document matching. However, for event matching also the efficient support of multi-valued predicates (including range predciates) is very important. This shortcoming of the SIFT methods is addressed and fixed by k-index [Wha09].

K-Index. The index k-index [Wha09] is static, supports multi-valued predicates, and is based on the inverted list data structure [ZM06]. However, k-index only supports single-valued predicates natively. A multi-valued predicate must be translated into a set that contains a corresponding single-valued predicate for each value of the multi-valued predicate. This is adequate only if there are very few values. But for attributes with high cardinality domains (e.g., all 32-bit integer numbers) as they are present in EP applications, range predicates cannot be supported efficiently. To overcome this problem, k-index proposes to organize attribute domains in hierarchies of different granularities. However, this proposal is not specified in detail. As an additional feature, k-index also allows to find and return only the top-k matching Boolean expressions for an incoming data item according to some scoring function [Mac08].

Gryphon. The static index presented in [Agu99] and developed for the publish/subscribe system Gryphon [Ban99] is a tree data structure. Actually, the Gryphon index is dynamic and created by inserting subscriptions one another. Every subscription is fully described by a path in the corresponding tree based on its predicates. Each tree node is associated with an attribute and a predicate type. On basis of the different values demanded by the corresponding predicate instances, a tree node points to different child nodes. All this is identical to the Esper index. For subscriptions that do not restrict a specific attribute there are catch-all edges to the respective next tree levels. At the end of a path, a corresponding subscription is stored in a leaf node. After the dynamic creation of a Gryphon tree for a set of subscriptions, several optimization techniques that prevent the insertion of additional subscriptions are performed. In contrast to the BE^{+}-tree, an a priori knowledge of all subscriptions is not directly exploited and there is no efficient bulk loading technique provided.

Esper Index. The SPE Esper [Esp] has its own implementation of a dynamic query index. All observations described here are extracted from the source code of version 5.0.0. As in our implementation, Esper optimizes filter conditions through rewriting. The Esper index consists of so-called *filter handle set nodes* (FHSN) that are organized in a tree. Each path in that tree represents a set of indexed filter conditions. In terms of BE-tree, every FHSN is a leaf, a partition and a cluster node at the same time. It can store any number of filter conditions. There is no maximum capacity. Furthermore, there can be any number of partitions within a FHSN. Each partition is associated with an attribute. But in contrast to BE-tree and just as Gryphon, partitions are also associated with a certain type of predicate. For example, if a set of filter conditions has $=$ predicates as well as $>$ predicates on a certain attribute defined, then there are two separate partitions. One partition contains all filter conditions having a $=$ predicate on that attribute, the other partition contains those with a $>$ predicate on it. Partitions are called *index entries* and organized differently in comparison to the BE-tree. They route on basis of the values demanded by all corresponding predicates to different FHSNs at the next level. While in the case of the BE-tree the clustering is always done via a grid index for intervals, the Esper index selects an appropriate Java data structure depending on the predicate type. For instance, an index entry is a hash map for $=$ predicates and a tree map for $>$ predicates. Note that the Esper index is similar to the (dynamic) Gryphon index that is clearly outperformed by the BE-tree [SJ11, SJ13]. Because Esper was not considered in any evaluation of query indexes yet, we ran some of our benchmarks using Esper. However, in every benchmark both the BE-tree and the BE$^+$-tree were very significantly superior to the Esper index.

Summary

This chapter presents the BE$^+$-tree, a novel index structure for large sets of continuous queries. It is based on the BE-tree which is the most efficient query index so far. The BE$^+$-tree extends the BE-tree by several optimization techniques that are integrated into a bulk loading method for BE-trees. First, due to bulk loading a globally optimal partitioning (according to some scoring function) can be achieved. Second, attribute domains are reduced to the range that is actually used by the filter conditions of all CQs. Third, through the fixed grid structure used by the BE-tree to index predicates in the form of intervals, it is possible that intervals get stuck at the upper levels. The BE$^+$-tree handles stuck intervals by splitting them so that the resulting parts can be pushed down to lower levels. An experimental evaluation shows that all proposed optimization techniques have a strong potential to improve the overall performance. Moreover, BE$^+$-trees can be created noticeably faster from scratch than BE-trees because of the bulk loading approach. The disadvantages of the BE$^+$-tree are an increased memory consumption caused by interval splits and the loss of the dynamic property. The former might not be relevant since the sizes of main memory are high today. But if it does, several parameters allow to control the required additional memory while still getting a significantly improved performance. The latter can be compensated in many applications by periodically recreating the query index.

12

Efficient Aggregation over Event Streams with Time-Interval Semantics

Outline

12.1 Introduction

Aggregation is important for almost every kind of data analytics and, thus, has gained much attention in the past. Different processing paradigms have different requirements and need tailor-made approaches to aggregation therefore. In the stream processing paradigm, the aggregation operator must support the concept of windows and should continuously update its output as efficiently as possible with regard to the high performance demands of applications. In the particular case of a data model with time-interval semantics as it is used in our implementation of JEPC, the design and implementation of the aggregation operator is more complicated and operators for data models with time-instant semantics cannot be taken over. By now, there has been proposed only one appropriate implementation so far. Unfortunately, this implementation scales linearly with the size of the operator state. In this chapter, we present an alternative implementation on basis of balanced trees, named *Agg-2-3-tree*, that scales logarithmically with the size of the operator state. An experimental evaluation confirms the significantly better performance of the Agg-2-3-tree.

The outline of this chapter is as follows. At first, we present the standard implementation of the aggregation operator for event streams with time-interval semantics. This is not only for demonstrating its shortcomings, but also for explaining the principles of windowed aggregation over event streams with time-interval semantics. Then, we introduce the Agg-2-3-tree which is a 2-3-tree augmented with the segment tree. Hereafter, we present experiments that examined its performance in comparison to the standard implementation. Finally, related work is discussed.

12.2 Standard Implementation

The snapshot-reducible data stream algebra [KS09] on which JEPC is based also comes with a default implementation that has been adopted by several systems which implement this algebra (e.g., PIPES [KS04], Odysseus [App12]). Algorithm 12 shows the proposed implementation of the aggregation operator. Note that the state of the aggregation operator does not consist of events of the input event stream as in the case of other stateful operators (e.g., join, union, difference). Instead, so-called *partial aggregates* that are also events with time-interval semantics are kept. For each point in time that is covered by the current state of the aggregation operator there is exactly one partial aggregate valid. This requires that within the state of the aggregation operator all partial aggregates have disjoint time intervals and that there are no gaps between

Algorithm 12: ScalarAggregation($S_{in}, f_{init}, f_{merge}, f_{eval}$)

```
Input: Stream: $S_{in}$, Functions: $f_{init}, f_{merge}, f_{eval}$
Output: Stream: $S_{out}$
SweepArea $sa \leftarrow$ NewSweepArea($\leq_{t_s}$, $[t_s : t_e) \cap [\hat{t}_s : \hat{t}_e) \neq \emptyset, t_s \geq \hat{t}_e$);
for $event := (p, [t_s : t_e)) \hookleftarrow S_{in}$ do
    Iterator $qualifies \leftarrow sa$.Query($event$);
    if $qualifies = \emptyset$ then
        $sa$.Insert(($f_{init}(p), [t_s : t_e)$));
    else
        Timestamp $last_{t_e} \leftarrow t_s$;
        while $qualifies$.HasNext() do
            Element $(\hat{p}, [\hat{t}_s, \hat{t}_e)) \leftarrow qualifies$.Next();
            $qualifies$.Remove();
            if $\hat{t}_s < t_s$ then
                $sa$.Insert(($\hat{p}, [\hat{t}_s, t_s)$));
                if $t_e < \hat{t}_e$ then
                    $sa$.Insert(($f_{merge}(\hat{p}, p), [t_s : t_e) \cap [\hat{t}_s : \hat{t}_e)$));
                    $sa$.Insert(($\hat{p}, [t_e : \hat{t}_e)$));
                else
                    $sa$.Insert(($f_{merge}(\hat{p}, p), [t_s : \hat{t}_e)$));
            else
                if $[\hat{t}_s : \hat{t}_e) = [t_s : t_e) \cap [\hat{t}_s : \hat{t}_e)$ then
                    $sa$.Insert(($f_{merge}(\hat{p}, p), [\hat{t}_s : \hat{t}_e)$));
                else
                    $sa$.Insert(($f_{merge}(\hat{p}, p), [\hat{t}_s : t_e)$));
                    $sa$.Insert(($\hat{p}, [t_e : \hat{t}_e)$));
            $last_{t_e} \leftarrow \hat{t}_e$;
        if $last_{t_e} < t_e$ then
            $sa$.Insert(($f_{init}(p), [last_{t_e} : t_e)$));
    Iterator $results \leftarrow sa$.ExtractElements($event$);
    while $results$.HasNext() do
        Element $(\hat{p}, [\hat{t}_s, \hat{t}_e)) \leftarrow results$.Next();
        $(f_{eval}(\hat{p}), [\hat{t}_s : \hat{t}_e)) \hookrightarrow S_{out}$;
```

the time intervals of two adjacent partial aggregates. The payload of a partial aggregate contains all information that is necessary for the computation of final aggregates and, thus, depends on the specific aggregate function being computed. Aggregate functions are described by three functions f_{init}, f_{merge} and f_{eval} [LWZ04]. While $f_{init}(p)$ is for starting a new partial aggregate based on the payload p of some input event, $f_{merge}(a, p)$ is for updating the payload of an existing partial aggregate a based on the payload p of an input event. The function $f_{eval}(a)$ transforms the payload of a partial aggregate a into the payload of a final aggregate. For example, the computation of the average value of a numeric event attribute *attr* can be achieved by modeling the payload of a partial aggregate as a pair (s, c) where s is the sum of any number of numeric values and c is a counter that gives the total number of values the sum has been computed for. Then, the aggregation function can be defined as follows [KS09]:

$$f_{init}(p) := (p.attr, 1)$$

$$f_{merge}((s, c), p) := (s + p.attr, c + 1)$$

$$f_{eval}((s, c)) := s/c$$

Whenever a new partial aggregate is started by an input event, the sum s is set to the value of the corresponding event attribute *attr* and the counter c is set to 1. On every update of a partial aggregate by an input event, the value of the corresponding event attribute *attr* is added to the sum s and the counter c is incremented. To obtain the average value from a partial aggregate, the sum must be divided by the counter.

Algorithm 12 uses a SweepArea (see Section 8.3.1 on page 69) to keep all partial aggregates that can still be affected by future events. This SweepArea is configured as follows (line 1). All partial aggregates of the SweepArea returned by ITERATOR are ordered by start timestamps ($\leq_{t_s}$). For a given reference event, QUERY returns all partial aggregates with a time interval that intersects the time interval of the reference event ($[t_s : t_e) \cap [\hat{t}_s : \hat{t}_e) \neq \emptyset$). The method EXTRACTELEMENTS gets and removes all partial aggregates with an end timestamp less than or equal to the start timestamp of a given reference event ($t_s \geq \hat{t}_e$). Those partial aggregates cannot be affected by future events anymore and are safe to be extracted, evaluated and reported. Partial aggregates are created and updated on arrival of new events. All existing partial aggregates that temporally intersect with an incoming event must be updated and are obtained therefore (line 3). If there are no partial aggregates to update, the event starts a new partial aggregate (lines 4–5). Otherwise, all partial aggregates that must be updated are iterated. Each of those partial aggregates is removed from the SweepArea (line 10) and one, two or three partial aggregates are inserted in return (lines 11–26). The total number of partial aggregates that are inserted depends on the relationship between the intersecting time intervals. If the time interval of the event starts after the time interval of the partial aggregate being updated ($\hat{t}_s < t_s$), a partial aggregate having the original payload and the time interval $[\hat{t}_s : t_s)$ is inserted into the SweepArea. Similarly, a partial aggregate having the original payload and the time interval $[t_e : \hat{t}_e)$ is inserted into the SweepArea if the time interval of the event ends before the time interval of the partial aggregate being updated ($t_e < \hat{t}_e$). For the range of time a partial aggregate is covered by the event a new partial aggregate is inserted. The payload of that partial aggregate is created by applying f_{merge} to the original payload and the payload of the event. Lastly, an event may cover a range of time which is not covered by any partial aggregate of the SweepArea. For this range of time a new partial aggregate must be initialized (line 26). After all partial aggregates of the SweepArea have been updated, those which cannot be affected by future events anymore are transformed into final aggregates and reported (lines 27–30).

The standard implementation uses an implementation of the SweepArea that is based on an ordered list, because the iterators returned by the methods QUERY and EXTRACTELELMENTS must output partial aggregates ordered by start timestamps [Krä07, KS09]. Since insertion into an ordered list has the time complexity of $\mathcal{O}(N)$ where N is the size of the list, the standard implementation takes linear time. As a consequence, the standard implementation scales poorly and mid-sized as well as large operator states result in insufficient performance. Up to now, there has not been proposed a better implementation yet.

Very often aggregation is used in combination with grouping. The standard implementation efficiently supports grouping as follows (a more detailed description and pseudo code can be found in [Krä07]). Each active group gets its own SweepArea. Every incoming event updates only the partial aggregates of its associated group. If an incoming event lets time progress, each active group might contain expired partial aggregates. But instead of searching every single active group for expired partial aggregates, all active groups are ordered by the start timestamps of their first partial aggregates using a min-priority queue. Thereby, all groups having a first partial aggregate whose start timestamp is less than the current point in time can be efficiently determined. For each of those groups, all expired partial aggregates are extracted and transformed into final aggregates. Because every group has its own clock, all final aggregates are ordered using another min-priority queue before they are put into the output stream of the aggregation operator.

12.3 Tree-Based Implementation

As an alternative to the standard implementation of the aggregation operator, we present an implementation that utilizes a balanced tree data structure instead of an ordered list for maintaining partial aggregates. Within the tree data structure partial aggregates are arranged in the form of a hierarchy. Of course, this requires to delay some merges of partial aggregates with incoming events and partial aggregates with each other, respectively.

12.3.1 The 2-3-Tree

The 2-3-tree is a special case of the B-tree [BM72, Com79] and was introduced unpublished in 1970 [Cor09]. Because there is no source that gives a commonly accepted definition of 2-3-trees, there can be found many and slightly different definitions in literature. Throughout this chapter, we use the following definition of 2-3-trees.

Definition 11 (2-3-Tree)**.** *The* 2-3-tree *is a search tree data structure. Every leaf node contains exactly one data item and one unique key. All leaf nodes are at the same level and totally ordered according to their keys. Every internal node including the root node has either two (left and right) or three child nodes (left, middle and right) and contains exactly one key but no data items. The key of an internal node is the maximum key in the subtree rooted by it.*

According to Definition 11, 2-3-trees inherently limit the number of children per internal node to two to three. This is in contrast to B-trees which allow user-defined limits. As a consequence, 2-3-trees are not widely branched and, thus, well-suited for main memory. Because all leaf nodes are at the same level, a 2-3-tree is always balanced and its height grows logarithmically with the total number of stored data items.

Theorem 4. *The height h of a 2-3-tree storing N data items is at most $\log_2 N$.*

Proof. A 2-3-tree is always balanced and each of its internal nodes has at least two child nodes. Then $N \geq 2^h$ holds for every 2-3-tree. This is equivalent to $h \leq \log_2 N$. □

Because of Theorem 4, all operations that traverse only a single path in the 2-3-tree can be done in logarithmic time. In particular, all basic operations (i.e., search, insert, and delete) have a time complexity of $\mathcal{O}(\log N)$. As an example, Algorithm 13 shows the search operation that requires the root node of a 2-3-tree and the key being looked up. The algorithm visits exactly one node per level until it reaches a leaf node. At each internal node, the relationship between the search key and the keys of the child nodes clearly determines which child node to visit next.

Algorithm 13: 23TreeSearch(*node*, *key*)

```
Input: 2-3-Tree Node: $node$, Key: $key$
Output: Data Item: $dataItem$
if IsLeaf($node.leftChild$) then
    if $node.leftChild.key = key$ then
        return $node.leftChild.dataItem$;
    else if $node.middleChild \neq$ NULL and $node.middleChild.key = key$ then
        return $node.middleChild.dataItem$;
    else if $node.rightChild.key = key$ then
        return $node.rightChild.dataItem$;
else
    if $key \leq node.leftChild.key$ then
        return 23TreeSearch($node.leftChild$, $key$);
    else if $node.middleChild \neq$ NULL and $key \leq node.middleChild.key$ then
        return 23TreeSearch($node.middleChild$, $key$);
    else if $key \leq node.key$ then
        return 23TreeSearch($node.rightChild$, $key$);
return NULL;
```

The insert and delete operations are also similar to those of the B-tree and therefore only roughly outlined in the following. At first, the target leaf node of the data item to insert or to delete is determined by performing a search operation with its key. Then, the data item is inserted into or deleted from the found leaf node. The insert and delete operations may lead to underflowing or overflowing internal nodes. Overflows and underflows are handled via node merges and node splits as in B-trees.

12.3.2 Managing Partial Aggregates in 2-3-Trees

For the purpose of indexing intervals, search trees can be augmented with data structures such as the interval tree [Ber08, Cor09] or the segment tree [Ber08]. In the following, we extend the 2-3-tree by features known from the segment tree. Recall that the 2-3-tree requires a key dimension which can be ordered according to the $<$ relation. But intervals have no natural total order according to the $<$ relation [All83, Moo79]. In order to manage partial aggregates in a 2-3-tree, we force adjacent partial aggregates to have disjoint but meeting time intervals and order them by their end timestamps.

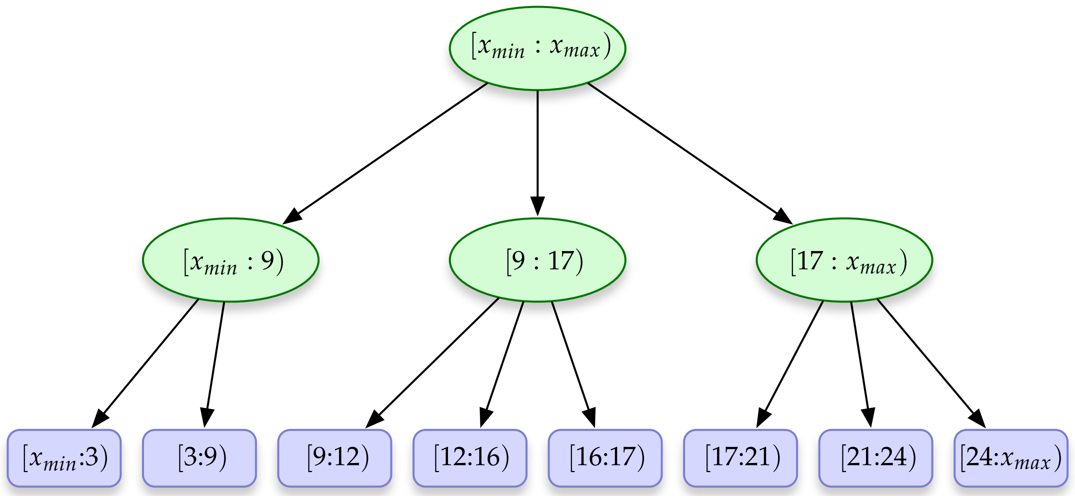


Figure 12.1: 2-3-tree managing partial aggregates

We use the following example to describe the general principle in more detail. Let x_{min} be any integer number with $x_{min} < 3$ and let x_{max} be any integer number with $x_{max} > 24$. Then, Figure 12.1 shows a valid 2-3-tree that manages partial aggregates. Each tree node is labeled with the corresponding time interval of its associated partial aggregate (note that the shown end timestamps are arbitrary and only intended to illustrate our approach). At first, we focus on the nodes at the leaf level. Here, the 2-3-tree keeps the partial aggregates ordered by their end timestamps, because the end timestamps are used as keys. Since we require adjacent partial aggregates to have disjoint but meeting time intervals, the start timestamp of a partial aggregate at the leaf level is identical to the end timestamp of its preceding partial aggregate. However, the very first partial aggregate has no preceding partial aggregate. Therefore, we set its start timestamp to the start timestamp of the very first event. Note that the partial aggregates at the leaf level (which is, in fact, an ordered list) are organized in the same way as in the standard implementation. For the time intervals of internal nodes the end timestamps are already determined by the 2-3-tree (i.e., the end timestamp of an internal node is identical to the maximum end timestamp in the subtree rooted by it). We set the start timestamp of an internal node to the minimum start timestamp in the subtree rooted by it. Then, the time interval of an internal node is the union of the time intervals of all its child nodes. In particular, the root node directly shows that all stored partial aggregates are in $[x_{min} : x_{max})$. This way, every internal tree level has the same properties as the leaf level (i.e., adjacent time intervals are disjoint but meet and all partial aggregates at the same level are ordered by their end timestamps).

12.3.3 The Agg-2-3-Tree

In the following, we introduce the Agg-2-3-tree. It is a 2-3-tree that manages partial aggregates as shown in the last section and comes with a set of tailor-made algorithms for fully supporting windowed aggregation over event streams with time-interval semantics. The data items being stored are the payloads of partial aggregates. In the Agg-2-3-tree, payloads of partial aggregates are stored in not only the leaf nodes, but also the internal nodes. As a consequence, there are multiple partial aggregates covering the same point in time. To get the final aggregate for such a point in time, the payloads of all covering partial aggregates must be merged before evaluation. Recall that in the standard implementation a new event is immediately merged with all partial aggregates which temporally intersect the event. For the new behavior the function f_{merge} must be defined slightly differently than in the standard implementation. In the case of the Agg-2-3-tree, $f_{merge}(a, a')$ combines the payloads of two partial aggregates a and a'. In addition, the modified function f_{merge} has a neutral element e. Whenever the payload of a partial aggregate a is merged with the neutral element, a is returned unchanged. For ease of presentation, an internal tree node can temporarily have a fourth child node that is referred to as $bufferChild$. In a correct order of child nodes, $bufferChild$ is always the rightmost child node.

12.3.3.1 Auxiliary Operations

We created for some tasks being performed by the basic operations of the Agg-2-3-tree auxiliary operations that are presented in the following. Algorithm 14 is used to add an arbitrary node $node2$ to an internal node of an existing Agg-2-3-tree $node1$ as a child node. If $node1$ has only two child nodes, then $node2$ can be simply added to $node1$ as a regular child node (lines 2–3). Otherwise, $node2$ must be added temporarily to $node1$ as a fourth child node (line 5). In every case, it is important to order all child nodes according to their keys (line 6). The algorithm AGG23TREEORDERNODE is not presented since it is a trivial ordering task. If the buffer has been used (line 7), $node1$ overflows and must be split. Therefore, a new node $node'$ that takes over the two rightmost child nodes of $node1$ is created (lines 8–12). In case $node1$ is the root node, a new root node must be created (i.e., the tree grows by one level). The new root node gets $node1$ and $node1'$ as child nodes as well as the data item of $node1$, while the data items of $node1$ and $node1'$ are set to the neutral element. If $node1$ is not the root node, then $node1'$ gets the same data item as $node1$ and is added to the parent node of $node1$ by recursively calling AGG23TREEADDNODE.

Algorithm 14: AGG23TREEADDNODE(*node*1, *node*2)

```
Input: Agg-2-3-Tree Node: $node1$, $node2$
$node2.parentNode \leftarrow node1$;
if $node1.middleChild$ = NULL then
    $node1.middleChild \leftarrow node2$;
else
    $node1.bufferChild \leftarrow node2$;
AGG23TREEORDERNODE($node1$);
if $node1.bufferChild \neq$ NULL then
    Agg-2-3-Tree Node $node1' \leftarrow$ NEWNODE($\{node1.rightChild, node1.bufferChild\}$);
    $node1.rightChild \leftarrow node1.middleChild$;
    $node1.bufferChild \leftarrow$ NULL;
    $node1.middleChild \leftarrow$ NULL;
    $node1.key \leftarrow node1.rightChild.key$;
    if $node1.parentNode$ = NULL then
        Agg-2-3-Tree Node $root \leftarrow$ NEWNODE($\{node1, node1'\}$);
        $root.dataItem \leftarrow node1.dataItem$;
        $node1.dataItem \leftarrow e$; $node1'.dataItem \leftarrow e$;
    else
        $node1'.dataItem \leftarrow node1.dataItem$;
        AGG23TREEADDNODE($node1.parentNode$, $node1'$);
```

Algorithm 15: AGG23TREEMERGEAGGREGATES(*node*)

```
Input: Agg-2-3-Tree Node: $node$
$node.leftChild.dataItem \leftarrow f_{merge}(node.leftChild.dataItem, node.dataItem)$;
if $node.middleChild \neq$ NULL then
    $node.middleChild.dataItem \leftarrow f_{merge}(node.middleChild.dataItem, node.dataItem)$;
$node.rightChild.dataItem \leftarrow f_{merge}(node.rightChild.dataItem, node.dataItem)$;
$node.dataItem \leftarrow e$;
```

In certain situations, it is necessary for reasons of correctness to push down the payload of a partial aggregate before changing its key. Algorithm 15 shows the procedure that is for pushing down the payload of a partial aggregate to the next level. The algorithm requires an internal node as argument and merges the data item of each of its child nodes with its data item. Afterwards, the data item of the given node is set to the neutral element *e* in order to eliminate the content.

```
Algorithm 16: Agg23TreeGetLeftmostLeafNode(*node*)
   Input: Agg-2-3-Tree Node: $node$
   Output: Agg-2-3-Tree Node: $leftmostLeafNode$
 1 Agg-2-3-Tree Node $leftmostLeafNode \leftarrow node$;
 2 while not IsLeaf($leftmostLeafNode$) do
 3     $leftmostLeafNode \leftarrow leftmostLeafNode.leftChild$;
 4 return $leftmostLeafNode$;
```

Algorithm 16 can be used to obtain the leftmost leaf node in a (sub-)tree. The leftmost leaf node in an entire Agg-2-3-tree is special, because it represents the first partial aggregate within the state of the aggregation operator. Starting at the root node of a (sub-)tree *node*, the algorithm traverses the leftmost path until it reaches a leaf node. This leaf node is the leftmost one and returned therefore.

12.3.3.2 Basic Operations

The next paragraphs present the basic operations of the Agg-2-3-tree. We start with the insert operation that is shown in Algorithm 17. Because Agg-2-3-trees can only handle partial aggregates, a new event is directly transformed into a partial aggregate whose payload is derived via f_{init} from the payload of the new event and whose key is set to the end timestamp of the new event. Then, this partial aggregate is put into a new node being inserted in place of the event (line 1). The first node being inserted starts a new Agg-2-3-tree (line 2), while the insertion of a new node into an existing Agg-2-3-tree is more complex. In our approach, an Agg-2-3-tree must have a certain property the insert operation relies on. The insert operation requires the start timestamp of the first partial aggregate x_{min} (see Figure 12.1) of an Agg-2-3-tree to be identical to the start timestamp of a new event. Note that this property is always ensured by the basic operations of the Agg-2-3-tree. Depending on the key of the node being inserted there are three different cases. The key of the node being inserted can be less than, equal to or greater than the maximum key x_{max} (see Figure 12.1) in the entire Agg-2-3-tree. If the keys are identical to each other, then the time interval of the node being inserted is the union of the time intervals of all stored partial aggregates. This means that all stored partial aggregates must be updated. Therefore, the node to insert is simply merged with the root node (line 3). The remaining two cases require each a differentiation whether the tree has only one level or more than one level. In general, an Agg-2-3-tree keeps a large set of partial aggregates and has multiple levels

Algorithm 17: AGG23TREEINSERT(*root*, *event*)

```
Input: Agg-2-3-Tree Node: $root$, Event: $event$
Agg-2-3-Tree Node $node \leftarrow$ NEWAGG23TREENODE($f_{init}(event.p), event.t_e$);
if $root$ = NULL then Start new Agg-2-3-Tree with $node$ as root node;
else if $root.key = node.key$ then $root.dataItem \leftarrow f_{merge}(root.dataItem, node.dataItem)$;
else if ISLEAF($root$) then
    Agg-2-3-Tree Node $node' \leftarrow$ NEWAGG23TREENODE($e$);
    if $root.key < node.key$ then
        AGG23TREEADDNODE($node$, $root$);
        $node'.key \leftarrow node.key$;
        AGG23TREEADDNODE($node$, $node'$);
    else
        AGG23TREEADDNODE($root$, $node$);
        $node'.key \leftarrow root.key$;
        AGG23TREEADDNODE($root$, $node'$);
else
    if $root.key < node.key$ then
        Agg-2-3-Tree Node $rightPathNode \leftarrow root$;
        while not ISLEAF($rightPathNode$) do
            AGG23TREEMERGEAGGREGATES($rightPathNode$);
            $rightPathNode.key \leftarrow node.key$;
            $rightPathNode \leftarrow rightPathNode.rightChild$;
        Agg-2-3-Tree Node $node' \leftarrow$ NEWAGG23TREENODE($e$, $node.key$);
        AGG23TREEADDNODE($rightPathNode.parentNode$, $node'$);
        ROOTNODE().$dataItem \leftarrow f_{merge}$(ROOTNODE().$dataItem, node.dataItem$);
    else
        Agg-2-3-Tree Node $node' \leftarrow root$; Bool $run \leftarrow$ TRUE;
        while $run$ = TRUE do
            foreach $child \in < node'.leftChild, node'.middleChild, node'.rightChild >$ do
                $run \leftarrow$ FALSE;
                if $child.key \leq node.key$ then
                    $child.dataItem \leftarrow f_{merge}(child.dataItem, node.dataItem)$;
                    if $child.key = node.key$ then return ;
                else if ISLEAF($child$) then
                    $node.dataItem \leftarrow f_{merge}(child.dataItem, node.dataItem)$;
                    AGG23TREEADDNODE($node'$, $node$); return;
                else
                    $node' \leftarrow child$; $run \leftarrow$ TRUE; break;
```

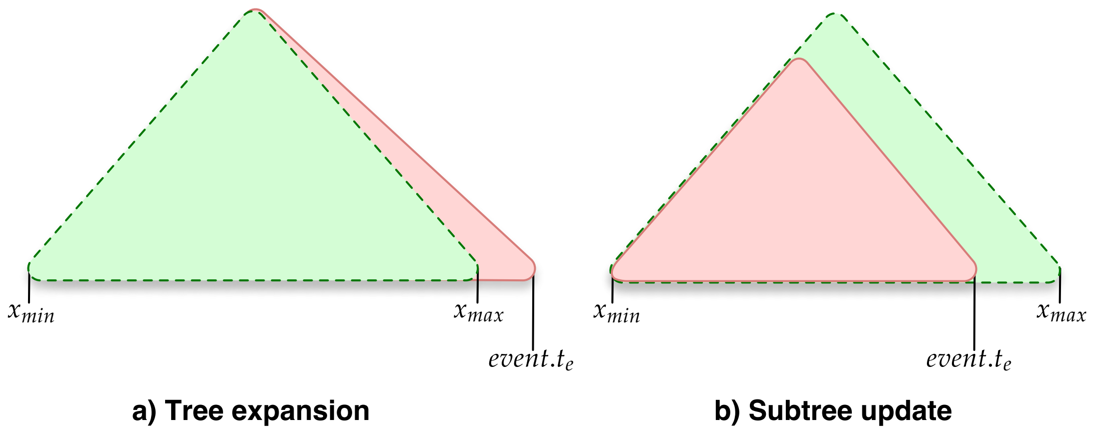


Figure 12.2: Agg-2-3-tree insertion

therefore. So, we discuss this case first. If the key of the node being inserted exceeds x_{max}, the Agg-2-3-tree is expanded at the right hand side and all partial aggregates are updated (see Figure 12.2a and lines 15–23). The tree expansion is done by traversing the rightmost path down to the rightmost node at the leaf level. While on its way down, the algorithm pushes down the payloads of all partial aggregates and updates the key of every visited internal node. At the leaf level, a new node that has the neutral element as data item and that takes over the key of the node being inserted is added. Finally, all partial aggregates are updated by merging the node being inserted with the root node (ROOTNODE simply gets the current root node). In the last case to discuss, the key of the node being inserted ends before x_{max} so that only a part of all partial aggregates must be updated. All affected partial aggregates are located in the subtree that covers a range starting at x_{min} and ending at some point between x_{min} and x_{max} (see Figure 12.2b). Therefore, the algorithm updates top-down all affected partial aggregates (lines 24–36). The node being inserted may have a key that is equal to the key of a visited node. Then, the update terminates at this point (line 31). But if the node to insert has a key that is exceeded by the key of the rightmost affected node at the leaf level, a new node is created at the leaf level for the range of time that is covered by both of them (line 34). In case that the Agg-2-3-tree consists only of its root node, tree expansion and subtree update cannot be performed. Instead of the tree expansion, the node being inserted simply becomes the new root node having the former root node as left child node and a new node keeping the neutral element as right child node (lines 6–9). The subtree update is replaced by a similar procedure. But here, the existing root node gets the node being inserted as left child node and a new node keeping the neutral element as right child node (lines 10–13).

Algorithm 18: AGG23TREEDELETELEFTMOSTLEAF(*root*, *key*)

```
Input: Agg-2-3-Tree Node: root, Key: key
if $root.key \leq key$ then
    SetRootNode(NULL);
else
    Agg-2-3-Tree Node node ← Agg23TreeGetLeftmostLeafNode(root);
    if $node.key = key$ then
        node ← node.parentNode;
        node.leftChild ← NULL;
        while node.leftChild = NULL do
            if node.middleChild ≠ NULL then
                node.leftChild ← node.middleChild;
                node.middleChild ← NULL;
                return;
            Agg23TreeMergeAggregates(node);
            if node.parentNode = NULL then
                node.rightChild.parentChild ← NULL;
                SetRootNode(node.rightChild);
                return;
            else
                Agg-2-3-Tree Node siblingNode ← NULL;
                if node.parentNode.middleNode ≠ NULL then
                    siblingNode ← node.parentNode.middleNode;
                else
                    siblingNode ← node.parentNode.rightNode;
                Agg23TreeMergeAggregates(siblingNode);
                if siblingNode.middleChild ≠ NULL then
                    Agg23TreeAddNode(node, siblingNode.leftChild);
                    siblingNode.leftChild ← siblingNode.middleChild;
                    siblingNode.middleChild ← NULL;
                    return;
                else
                    Agg23TreeAddNode(siblingNode, node.rightChild);
                    node.parentNode.leftChild ← NULL;
                    node ← node.parentNode;
```

For the delete operation of the Agg-2-3-tree we assume a sliding window behavior (that, of course, also includes the jumping windows). In particular, we expect partial aggregates to be removed only one another and strictly from the left to the right. Therefore, the delete operation shown in Algorithm 18 is only for removing the first partial aggregate (or all partial aggregates at once as a special case). If it is called with a key greater than the key of the root node, then the entire Agg-2-3-tree is deleted (line 1). Otherwise, it must be called eventually with the key of the first partial aggregate. During execution of the aggregation operator, the key to delete will always be the current point in time. Then, the first partial aggregate is removed by the algorithm if and only if it cannot be longer affected by any event in the future. Because of the assumption, only the leftmost leaf node is of interest. Therefore, the algorithm gets that node (line 4) and compares its key to the key given as argument. Only if the keys are equal, the leftmost leaf node is deleted. This is done by simply removing it from its parent node (lines 6–7). Obviously, this procedure leads to an invalid Agg-2-3-tree, because the parent node has no left child node anymore. The following loop (lines 8–32) fixes this problem recursively in a bottom-up fashion. In the best case, the parent node has a middle child node besides the right child node. Then, the middle child node can be taken over as new left child node (lines 9–12). After this little modification the entire Agg-2-3-tree is valid again and the algorithm terminates. Otherwise, the parent node is underflowing and a local reorganization is necessary. Because the underflowing node currently being processed will be modified in some way or other, its partial aggregate is pushed down to its child nodes (line 13). If this node is the root node, then the Agg-2-3 tree shrinks by one level. This is achieved by making its right child node (the only existing child node) to the new root node (lines 14–17). And if it is not the root node, then it has a sibling node on the right hand side. Those two nodes have at least three child nodes in sum and can exchange nodes or can be merged in order to resolve the underflow. To prepare the following procedure, also the partial aggregate of the sibling node is pushed down to its chid nodes. If the sibling node has three child nodes, then its left child node can simply be taken over (lines 25–28). In this case, the Agg-2-3-tree becomes completely valid again and the algorithm terminates. But if the sibling node has only two child nodes, a merge operation is performed (lines 29–32). Therefore, the remaining child node of the underflowing node currently being processed is added to its sibling node and the underflowing node is removed from its parent node afterwards. In this case, the underflow is resolved at the current level but may be delegated to the parent level. Because of this, the loop starts a new iteration for the parent node of the former underflowing node.

Algorithm 19: AGG23TREEAGGREGATIONWITHGROUPING(S_{in}, f_{init}, f_{merge}, f_{eval}, f_{group})

```
Input: Stream: $S_{in}$, Functions: $f_{init}$, $f_{merge}$, $f_{eval}$, $f_{group}$
Output: Stream: $S_{out}$
Data: Map: $groupToTree$ with entries ($gID$, $agg23tree$),
           $groupToOutput$ with entries ($gID$, $outputEvent$),
           $groupToClock$ with entries ($gID$, $t$),
      Heap: $timeToGroups$ with entries ($t$, $groups := \{gID_1, gID_2, \dots\}$) sorted by $t$
for $event := (p, [t_s : t_e)) \hookleftarrow S_{in}$ do
    Group Identifier $groupID \leftarrow f_{group}(event)$;
    if $groupToClock$.GET($groupID$) $< t_s$ then
        Timestamp $clock$; Set $groups$;
        repeat
            if $timeToGroups$.GETFIRST().$t > t_s$ then
                $clock \leftarrow t_s$;
                $groups \leftarrow \{groupID\}$
            else
                $clock \leftarrow timeToGroups$.GETFIRST().$t$;
                $groups \leftarrow timeToGroups$.GETFIRST().$groups$;
                $timeToGroups$.REMOVEFIRST();
            for $gID \in groups$ do
                Agg-2-3-Tree Node $root \leftarrow groupToTree$.GET($gID$);
                AGG23TREEGETNEXT($gID$, $root$, $clock$, FALSE);
            if $clock = t_s$ and $group2Output$.GET($groupID$) = NULL then
                Agg-2-3-Tree Node $root \leftarrow groupToTree$.GET($groupID$);
                AGG23TREEGETNEXT($groupID$, $root$, $clock$, TRUE);
            for ($gID$,$outputEvent$) $\in groupToOutput$ do
                ADDTOOUTPUTQUEUE($outputEvent$);
                Agg-2-3-Tree Node $root \leftarrow groupToTree$.GET($gID$);
                AGG23TREEDELETELEFTMOSTLEAFNODE($root$, $clock$);
            $groupToOutput$.CLEAR();
        until $clock = t_s$;
    Agg-2-3-Tree Node $root \leftarrow groupToTree$.GET($groupID$);
    if $root$ = NULL then
        $root \leftarrow$ NEWAGG23TREE($f_{init}$, $f_{merge}$, $f_{eval}$);
        $groupToTree$.PUT(($groupID$, $root$));
    AGG23TREEINSERT($root$, $event$);
    if $timeToGroups$.GET($t_e$) = NULL then
        $timeToGroups$.ADD(($t_e$, $\{groupID\}$));
    else $timeToGroups$.GET($t_e$).$groups$.ADD($groupID$) ;
```

12.3.3.3 Aggregation Operator

In the following, we present a more efficient implementation of the aggregation operator for event streams with time-interval semantics. Algorithm 12 shows our improved implementation that utilizes the Agg-2-3-tree and already includes the support of grouping. Exactly as the standard implementation, the algorithm requires the input event stream and the aggregate functions. In addition, a function f_{group} for partitioning the input event stream must be given. Then, each active group gets a separate Agg-2-3-tree for managing its partial aggregates. The map *groupToTree* stores for each group identifier the corresponding instance of the Agg-2-3-tree. Output events are buffered per group in the map *groupToOutput*. Since the input event stream is partitioned, each group has its own clock that is managed in the map *groupToClock*. Every new event is inserted into the Agg-2-3-tree of its associated group, while the Agg-2-3-trees of all other groups are not affected. However, a new event may let time progress. Assume that time progresses from some point in time now' to a later point in time now. Among all active groups, all partial aggregates that expired with respect to now must be reported in a temporal order. The naïve approach would be to let time progress from now' to now by iterating all time instants in-between in a temporal order. For each of those time instants, all groups are checked whether there is an expired partial aggregate to report. This naïve approach results in a semantically correct implementation, but is extraordinarily costly, because there are many false positive checks performed. For some points in time there are no expired partial aggregates at all. Checking all active groups for this point in time leads to a number of false positive checks that grows linearly with the number of active groups. But even if there are expired partial aggregates for some point in time, there still might be groups without expired partial aggregates. Therefore, we developed a better approach to grouping that is much more efficient. In particular, we use an index data structure to check only points in time for that at least one partial aggregate expired. Moreover, for each of those time instants we maintain a set that contains the identifiers of all groups which have an expired partial aggregate.

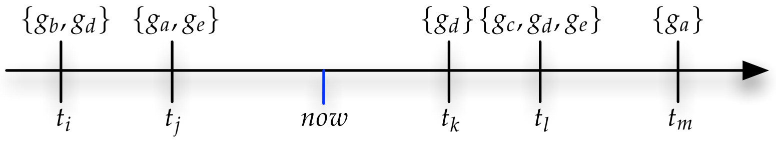


Figure 12.3: Index for efficient support of grouping

Figure 12.3 illustrates the index data structure we use to efficiently support grouping. It shows the timeline and all points in time at that at least one partial aggregate expires (t_i–t_m). For each shown point in time there is an associated set keeping the identifiers of all groups with an expiring partial aggregate. Assume that a new event lets time progress from any point in time before t_i to the point in time *now* that is marked in the figure. Then, the index can be queried to find out that at t_i the groups g_b and g_d as well as at t_j the groups g_a and g_e must be updated. Because a temporal order is important, the index *timeToGroups* is a heap that keeps pairs $(t, groups := \{gID_1, gID_2, \ldots\})$ ordered by t. Whenever an event lets time progress, Algorithm 19 utilizes this index to find, remove and report all expired partial aggregates in a temporal order using an output queue (lines 5–23). The last iteration is always done for *clock* set to t_s. The reason is that the associated group of the event may have a first partial aggregate that is not expired yet. Then, it exceeds the point in time t_s and must be split at t_s therefore. The first part with t_s as end timestamp gets removed and reported (obviously, we forcibly created an expired partial aggregate) while the right part becomes the new first partial aggregate. This also ensures that in the next part of the algorithm (lines 25–29) the event can be inserted into the corresponding Agg-2-3-tree, because now the start timestamps of the tree x_{min} and the event are identical to each other. At the end, the index must be updated because of the inserted event there is now a group that has a partial aggregate which will expire at t_e (lines 30–32).

Algorithm 20: AGG23TREEGETNEXT(*groupID*, *root*, *key*, *force*)

```
Input: Group Identifier: groupID, Agg-2-3-Tree Node: root, Key: key, Bool: force
Data: Functions: $f_{merge}$, $f_{eval}$,
      Map: groupToOutput with entries (gID, outputEvent),
           groupToClock with entries (gID, t)
Agg-2-3-Tree Node node ← AGG23TREEGETLEFTMOSTLEAFNODE(root);
Timestamp $t_s$ ← node.$t_s$;
if (node.key > key and force = TRUE) or (node.key = key and force = FALSE) then
    Payload p ← e;
    while node ≠ NULL do
        p ← $f_{merge}$(p, node.dataItem);
        node ← node.parentNode;
    Event outputEvent ← ($f_{eval}$(p), [$t_s$ : key]);
    groupToOutput.PUT(groupID, outputEvent);
    groupToClock.PUT(groupID, key);
```

The procedure shown in Algorithm 20 creates the next output event for a group and puts it into $groupToOutput$. Note that the algorithm has access to the aggregate functions f_{merge} and f_{eval} as well as to the data structures $groupToOutput$ and $groupToClock$ used by the aggregation. Algorithm 20 can be called in two different modes via the parameter $force$. In non-forced mode ($force$=`FALSE`), the first partial aggregate is checked whether it is expired. If so, it is transformed into a final aggregate by merging all partial aggregates on the path to it. Otherwise, there is no output created at all. And in forced mode ($force$=`TRUE`), the first partial aggregate is transformed into an output event independent of whether or not it is expired. Because *key* is used in every case as the end timestamp of the output event and as the new start timestamp of the tree, the first partial aggregate is split into two parts if it is not expired yet (line 10).

12.4 Evaluation

This section presents experiments we conducted on a commodity machine with an i7-2600 CPU, 8 GiB of main memory running 64-bit HotSpot Java VM (1.7.0_13). The standard implementation was evaluated using PIPES [KS04] and the implementation based on the Agg-2-3-tree was evaluated using the native EP provider that contains it as implementation of the aggregation EPA. Events were processed sequentially by using one thread only. Each experiment included a warm-up phase during which the operator state was filled up completely before measurements were taken.

12.4.1 Scalability

At first, we examined the performances of both implementations with respect to different sizes of the operator state. The experimental setup comprised the computation of an average aggregate over all events within a sliding time window of size x time instants. Each event had three 32-bit integer numbers as payload and events were pushed as fast as possible. At every point in time there was exactly one event so that x also gives the total number of events a sliding window permanently contained.

x	*Standard implementation*	*Agg-2-3-tree*
10k	1,237 events/s	209,590 events/s
20k	606 events/s	200,013 events/s
40k	304 events/s	188,513 events/s

Table 12.1: Scalability of different aggregation operator implementations

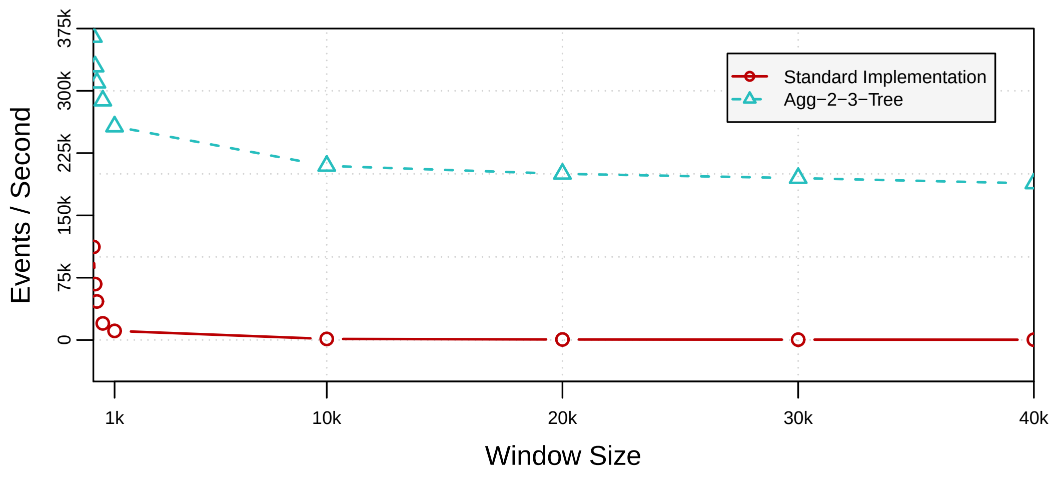


Figure 12.4: Effect of window size

Figure 12.4 shows the results for different values of x on the horizontal axis. In terms of absolute numbers, the implementation based on the Agg-2-3-tree clearly outperformed the standard implementation. Its corresponding graph also reveals the logarithmic time complexity of the implementation based on the Agg-2-3-tree. Because the corresponding graph of the standard implementation very quickly touches the zero line, Table 12.1 gives the exact measurements for window sizes of 10k, 20k and 30k time instants. The second column reveals the linear time complexity of the standard implementation. When the window size was doubled, the maximum event throughput halved. The third column shows that this was not true for the implementation based on the Agg-2-3-tree. Since the standard implementation is not able to cope with mid-sized and large operator states, we further increased the window size only for the implementation based on the Agg-2-3-tree. Figure 12.5 shows the results for window sizes ranging from 50k up to 2M time instants. Even in case of large windows the event throughput was adequate because of the logarithmic time complexity.

12.4.2 Simultaneous Events and Jumping Windows

If any window is applied to simultaneous events or if any jumping window is applied to any events (note that jumping windows are popular before aggregation), resulting events may have identical time intervals. The standard implementation benefits from this in multiple ways. First, an additional event with the same time interval as a previous event does not result in additional partial aggregates. Its time interval is the union of the time intervals of the first existing partial aggregates. Because its time interval does not overlap any existing partial aggregate, no additional partial aggregates must be added. As a consequence, the size of the operator state does not increase. Second, an additional event with the same time interval as a previous event does not let time progress so that no final aggregates must be computed.

In the experiments presented next, the previous experimental setup was changed as follows. At each point in time the input event stream had N events in total (N was varied from 1 to 15). The size of the time window was set to 10,000/N. This means that the aggregate was computed for 10,000 events in each run. But for the reasons discussed in the last paragraph, the state of the operator consisted of only 10,000/N

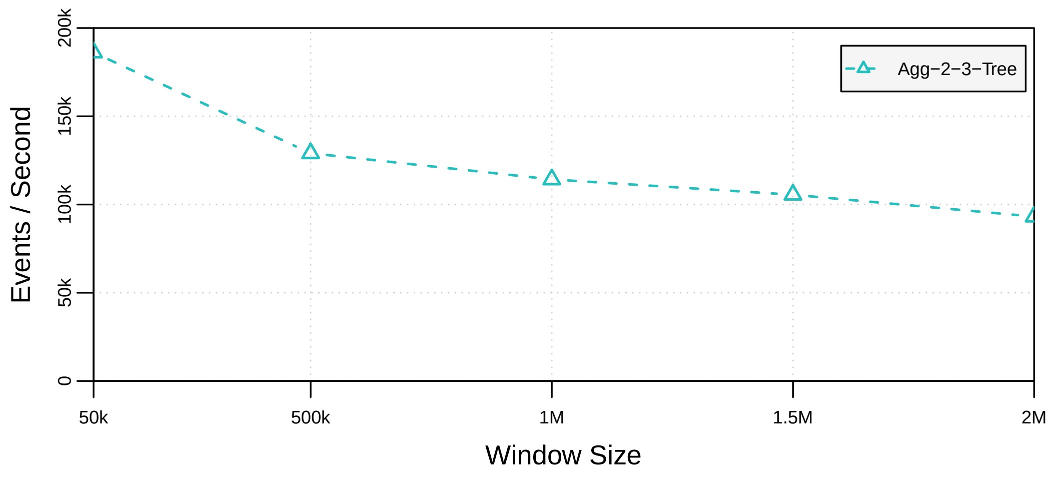


Figure 12.5: Effect of large window sizes

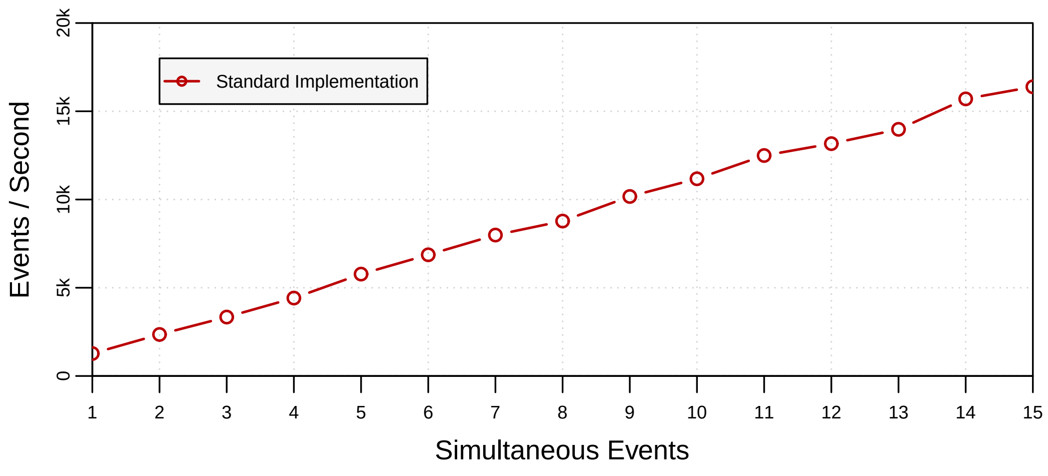


Figure 12.6: Effect of simultaneous events (standard implementation)

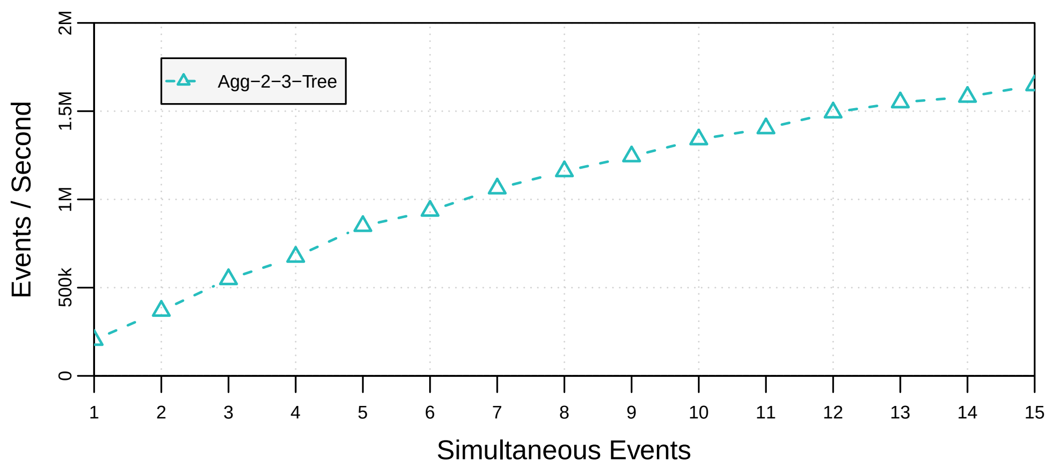


Figure 12.7: Effect of simultaneous events (Agg-2-3-tree)

partial aggregates. Figure 12.6 shows the performance of the standard implementation for different numbers of simultaneous events *N* on the x-axis. As discussed, the standard implementation benefited from simultaneous events in a linear fashion. The size of the state decreased as the number of simultaneous events increased. The shown graph again confirms the linear time complexity of the standard implementation.

Due to its design, simultaneous events do also not result in additional partial aggregates in the implementation based on the Agg-2-3-tree. And because simultaneous events do not let time progress, our implementation does not look for expired partial aggregates in this case. Algorithm 19 explicitly checks whether time has progressed before expired partial aggregates are searched (line 3). Figure 12.7 shows the results of the same experiment using the implementation based on the Agg-2-3-tree. Obviously, also the Agg-2-3-tree benefited from simultaneous events. However, the absolute numbers were significantly higher and just as the linear time complexity of the standard implementation showed up in this experiment, so also the logarithmic time complexity of the implementation based on the Agg-2-3-tree showed up.

12.4.3 Grouping

In our final experiments, we studied the effect of grouping. Both the standard implementation and the implementation based on the Agg-2-3-tree are designed to support grouping much more efficiently than the corresponding naïve approaches. To avoid biased results arising from the different time complexities of the implementations, the experiments were configured in a way such that the sliding time window used by the aggregation always contained exactly 10k events for every group. Again, the experimental setup comprised the computation of an average aggregate, but this time in combination with grouping. We partitioned the timeline into segments each covering one million time instants. For each segment we generated exactly one input event per group. The sliding time window used by the aggregation was configured to cover exactly 10k segments. In the first runs of the experiment, the timestamps of all events of a segment were set to the minimum time instant of their segment. Thus, there were globally as many simultaneous events per segment as groups. But note that within each single group there were no simultaneous events at all.

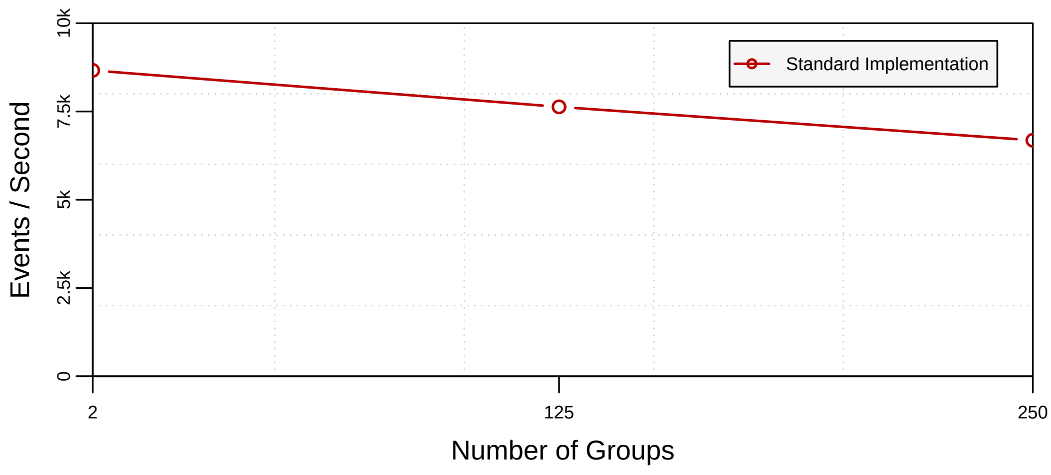


Figure 12.8: Effect of grouping (standard implementation)

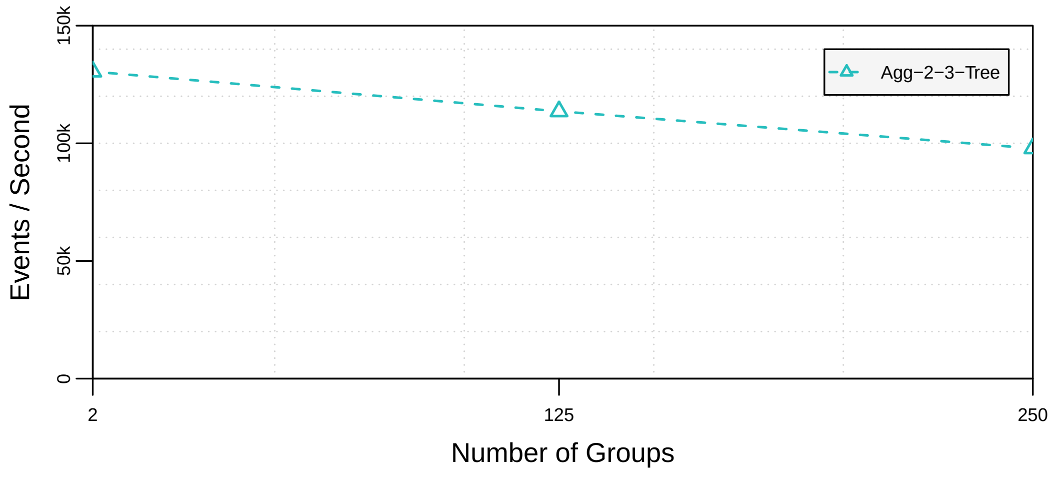


Figure 12.9: Effect of grouping (Agg-2-3-tree)

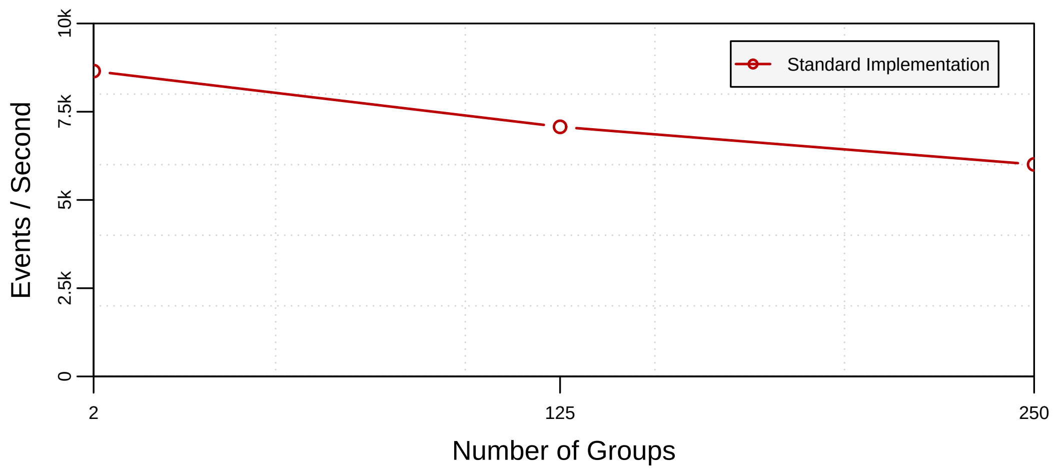


Figure 12.10: Effect of grouping (standard implementation)

Figure 12.8 shows the performance of the standard implementation for different numbers of groups. With an increasing number of groups, the throughput decreased slightly sublinearly. From 2 groups to 125 groups the throughput dropped by 1,033 events/s and from 125 groups to 250 groups it dropped by 947 events/s. The throughput of the implementation based on the Agg-2-3-tree also decreased slightly sublinearly (see Figure 12.9). It dropped by 16,873 events/s and by 15,719 events/s respectively. But again, the performance of the implementation based on the Agg-2-3-tree was significantly better than the performance of the standard implementation.

The last experiment we present had the same setup as the previous experiment. But this time, all events of a segment were uniformly distributed across their segment. As a consequence, there were almost no simultaneous events so that virtually every event let the time progress. Therefore, the effort to create finished aggregates must have been done per input event rather than per segment as in the previous experiment. Figure 12.10 shows that again the throughput decreased slightly sublinearly with an increasing number of groups using the standard implementation. However, the decrease was more sharply in comparison to the previous experiment. In particular, the throughput dropped by 1,579 events/s and then by another 1,065 events/s. This was simply due to the fact that there were no simultaneous events which could have been exploited. Figure 12.11 shows the results for the implementation based on the Agg-2-3-tree. Here, we observed exactly the same effect. The throughput also decreased slightly sublinearly and more sharply in the absence of simultaneous events (18,321 events/s followed by 15,861 events/s, to be precise).

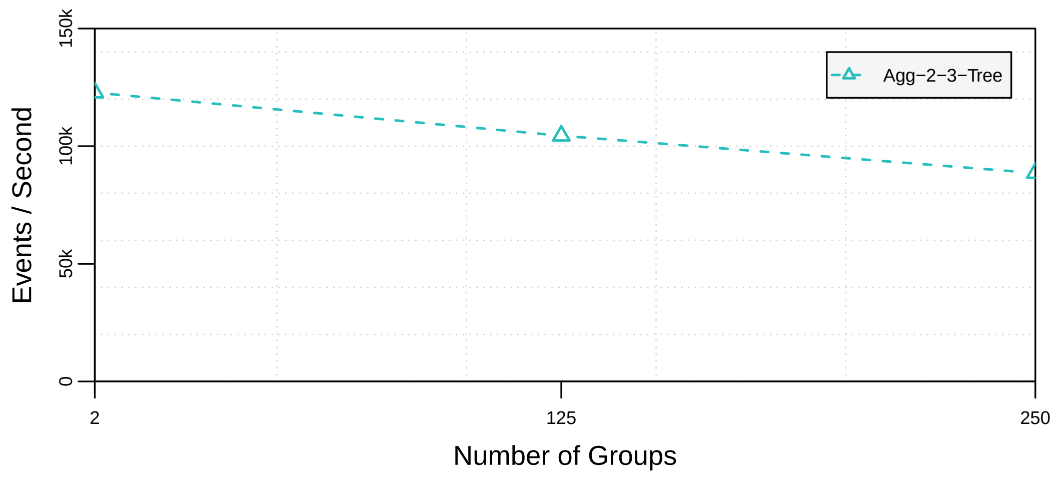


Figure 12.11: Effect of grouping (Agg-2-3-tree)

Altogether, our alternative implementation based on the Agg-2-3-tree not only achieves a much better performance than the standard implementation, but also supports grouping efficiently. As a pleasant side effect, this allows for efficient duplicate elimination via the aggregation operator [Lar97]. The impact of the used index can be determined when we disable it. Then, our alternative implementation must check for each single time instant that has past all active groups for expired partial aggregates. For instance, a run of the last experiment with only two active groups and the index disabled resulted in a maximum event throughput of only 14 events per second because of a very high number of unnecessary checks.

12.5 Related Work

The content of this chapter is related to work done in the areas of temporal database systems [JS96, Sno92] and data stream processing. Aggregation over tables in a temporal database has the same semantics as the aggregation operator of the snapshot-reducible data stream algebra. But all proposed approaches are tailor-made for data being stored on external memory. Moreover, some of the approaches are not incremental and most of them do not support a sliding window behavior. In the area of data stream processing, the proposed approaches to aggregation are incremental and support a sliding window behavior, but rely on a data model with time-instant semantics and not on the more challenging data model with time-interval semantics.

12.5.1 Temporal Aggregation in Databases

The aggregation tree and its extension the k-ordered aggregation tree [KS95] were the first disk-efficient (i.e., each tuple is read only once from disk) approaches to temporal aggregation. Both of them use a tree data structure based on the segment tree [Ber08] to manage partial aggregates that are updated by processing tuples one another. Tuples should be processed in a random order, because the tree data structure is not balanced. If tuples are processed in a temporal order, the data structure degenerates to a list. In the case of temporally ordered or only slightly disordered tables, the k-ordered aggregation tree is more memory-efficient. It requires every tuple to be at most k positions away from its position in a temporal order of the input table. Then, it is known when a partial aggregate cannot be affected by the remaining tuples anymore. This allows the use of a garbage collector in order to keep the data structure small. However, the garbage collector only reduces the needed main memory so that the time complexity is still $\mathcal{O}(N^2)$. The aggregation tree inspired subsequent work on temporal aggregation and was successfully parallelized [Gen99, YK97].

The balanced tree algorithm and the merge-sort aggregation [MLI00] are more efficient approaches. They are complementary to each other, because the key idea is to exploit the different characteristics of aggregate functions by using different techniques. The balanced tree algorithm is for computing `COUNT`, `SUM` and `AVG` aggregates. It requires the set of all timestamps (start timestamps as well as end timestamps) to be sorted first. Then, all timestamps are iterated in temporal order. For example, `COUNT` aggregates are computed by using a single counter variable (`SUM` and `AVG` are handled analogous). At every start timestamp the counter is incremented and at every end timestamp the counter is decremented. Since this approach is dominated by the sorting of all timestamps, it has a time complexity of $\mathcal{O}(N \log N)$. The authors propose to manage the timestamps in a balanced red-black tree as an optimization for repeated timestamps. Because the balanced tree algorithm is only capable of computing a limited set of standard aggregate functions efficiently, the paper proposes a merge-sort algorithm for computing `MIN` and `MAX` aggregates. However, neither the balanced tree algorithm nor the merge-sort aggregation perform an incremental computation. Therefore, they are not able to efficiently support a sliding window behavior. For large databases, the authors further propose the bucket algorithm that partitions the data along the timeline with respect to memory constraints. Because tuples with very large time intervals make partitioning difficult, the bucket algorithm uses a so-called *meta array* which keeps all those tuples. The bucket algorithm also allows for parallelization of the balanced tree algorithm and the merge-sort aggregation.

The SB-tree [YW03] is a B-tree [BM72] augmented with the segment tree [Ber08] for managing partial aggregates in a hierarchy. Thus, it is similar to the Agg-2-3-tree. The properties inherited from the B-tree make the SB-tree balanced and disk-efficient. All tree nodes have a minimum capacity and a maximum capacity as in the case of the B-tree. The SB-tree allows for incremental computations. It has been extended in several ways by its authors to support a sliding window behavior. The dual SB-tree and the joint SB-tree support sliding window queries which do not have a fixed window size. While the former maintains two SB-trees, the latter maintains the same information but in a single tree data structure only. Lastly, the MSB-tree is an extension that stores additional information in a regular SB-tree for efficient support of `MIN` and `MAX` aggregates in sliding window queries. Since the SB-tree is a disk-based data structure, it is not efficiently applicable in data stream processing.

12.5.2 Sliding Window Aggregation in Stream Processing

Much work done on processing (sliding window) aggregation queries over data streams is only approximate (e.g., [AM04, BS03]) in order to save time, space, or both. This work is not discussed, because an EP system must be able to report exact results for sliding window aggregation queries. Furthermore, proposed approaches which do not fully support a sliding window behavior are also not discussed (e.g., [Cra03]). The remaining approaches proposed by the stream processing community are discussed in the following. Except one of them that only supports a limited set of standard aggregates, all approaches focus on aggregation over event streams with time-instant semantics. Therefore, the standard implementation that has been already presented and discussed in detail in Section 12.2 is the only competing approach to sliding window aggregation over event streams with time-interval semantics.

B-Int [AW04] supports aggregation over event streams with time-instant semantics. This semantics allows for arbitrarily segmenting the input event stream according to base intervals of different granularities. Each base interval stores the precomputed aggregation result over all events that are covered by it. A lookup operation requires only to combine a (minimal) set of base intervals that together cover all events within the requested range. The work presented in [Tan15] describes a more efficient implementation that is faster by a constant factor in practice.

Paned windows [Li05] and paired windows [KWF06] also partition the input event stream in order to summarize events for partial aggregate pre-computation, but limit the total number of levels to two in order to speed up the insertion of new events. Both approaches perform well in the special case of jumping but overlapping windows with a large jump size. In this case, insert operations are much more frequent than final aggregate computations. Paned windows partition the input event stream into equally-sized parts. In contrast, paired windows are more sophisticated, because they create partitions with respect to the overlap of successive windows and have been proven to be superior to paned windows. However, for sliding windows both paned and paired windows degenerate and have a linear time complexity.

The work presented in [Gha07] supports aggregation over event streams on basis of the positive-negative approach (PNA). In the PNA, a new event is immediately inserted and valid until a corresponding negative event is inserted. Recall that the time-interval semantics can be mapped to the PNA and vice versa (see Section 7.2). The start timestamp of the associated time interval corresponds to the timestamp of the positive event and the end timestamp corresponds to the timestamp of the negative event. However, the proposed approach only supports aggregation functions that are invertible. Therefore, it is not possible to support all standard aggregates.

Summary

In this chapter, we introduce the Agg-2-3-tree, a novel data structure for efficiently managing partial aggregates. On basis of the Agg-2-3-tree, we propose a novel implementation of the aggregation operator for event streams with time-interval semantics. We present our evaluation in which we compared our proposed implementation based on the Agg-2-3-tree to the only competing implementation in a series of experiments. Those verified that our novel implementation has a better time complexity, is superior in terms of absolute performance, supports grouping with the same efficiency, and also takes advantage of events having identical time intervals.

13 Conclusions

During the last years, multiple systems for the real-time processing of event streams have been developed. Because of different roots and the lack of standardization, individual systems differ from each other so that the system landscape is highly heterogenous today. This part successfully demonstrated in the form of the design and the implementation of a prototype that despite the wide range of problems arising from the heterogeneity middlewares for EP in the sense of ODBC/JDBC for database systems are still technically feasible. But in contrast to ODBC/JDBC, middlewares for EP require a substantial richer specification and a higher implementation effort. It must be defined not only a common API, but also a common data model and a common query language. Our prototype, the Java Event Processing Connectivity, provides an expressive query language having a well-founded semantics and already allows the connectivity to different stream processing engines, to all standard database systems and to a native implementation of the JEPC specification, which provides among others a high-performance query index and a high-performance operator for aggregating events. Since JEPC overcomes the problem of vendor lock-in, different supported EP providers can not only be seamlessly interchanged, but also be mixed without any problems. For instance, this enables every application on top of JEPC to utilize the provided high-performance query index that ensures good scalability with the number of queries, while for all other tasks the desired EP provider is used.

Part III

Extensions to the Middleware

14 Introduction

Outline

14.1 Extending Event Processing Technology

This part presents several extensions to state-of-the-art event processing technology we developed on top of JEPC. The advantage of this approach is that an extension must be implemented only once and can then be used in combination with every raw EP provider supported by JEPC. Furthermore, also the semantics and behavior of an extension does not change when the raw EP provider is exchanged for another one. The extensions we introduce are specifically designed for the problems and upcoming challenges of event processing technology described in Chapter 2.

Sequential pattern matching which is the de facto standard approach to event pattern matching in EP does not fully exploit the temporal dimension of events. Therefore, we propose TPStream, a temporal pattern matcher with time-interval semantics, and show that temporal pattern matching is not a competing approach to sequential pattern matching, but a true extension to sequential pattern matching, because temporal pattern matching includes sequential pattern matching as a special case.

Typical EP systems can handle only primitive data types. However, events are often associated with not only a point in time, but also a location. The growing amount of mobile devices and real-world sensors will further increase the need to process spatial event attributes. We show how to extend JEPC in order to process events containing arbitrarily complex spatial data (i.e., spatial operators and spatial predicates can be used in queries). Our approach is applicable to every type of EPA and allows traditional EPAs as well as EPAs with spatial capabilities to seamlessly work together in EPNs. As a concrete example, we apply our approach to TPStream in order to detect spatiotemporal event patterns besides temporal event patterns.

There are many existing and emerging applications into which the integration of today's static EP technology causes problems. We propose dynamic event processing as a new paradigm therefore. Dynamic EP comprises two different aspects. First, we introduce a matchmaker that establishes fully automatically all connections between the elements of a stream processing application and immediately reacts to changes. This makes stream processing applications self-adaptive to varying sets of data sources, CQs and data sinks. Second, in existing EP systems a query is not allowed to change at runtime. However, context-sensitive EP applications require queries that can be quickly adapted when the context changes. We describe a general approach to quickly and safely updating CQs on-the-fly. Because this approach requires access to the recent history of event streams, we also present a high-performance event store for recording and reloading enormous event streams.

Lastly, we present extensions for improving the overall performance of EP applications built with JEPC. Just as in database systems, a query optimizer is essential for efficient query execution in event processing. We introduce a query optimizer for JEPC that takes over powerful rule-based and cost-based optimization techniques from database systems. Since pattern matching is not part of standard database systems, there exist no optimization techniques that could be adopted. Therefore, we propose novel optimization techniques for single pattern matching EPAs as well as for entire sets of pattern matching EPAs being applied to the same event stream. Because challenging workloads cannot be handled by a single JEPC instance, we present a framework that distributes queries across multiple and parallel running JEPC instances. Due to the unification achieved by JEPC, parallel running instances must not necessarily connect to the same raw EP provider. On the contrary, different types of EP providers are enabled to work together in federations. Because different raw EP providers have different strengths and weaknesses, synergies can be exploited in such federations to achieve a performance that is better than the performance of the parallelization of just a single type of EP provider. Our framework automatically distributes queries at the operator level with respect to not only load balancing, but also the individual strengths and weaknesses of the available types of EP providers.

14.2 Remainder of the Part

The outline of this part is as follows. Chapter 15 presents the temporal pattern matcher TPStream and shows how to extend JEPC by spatial features. Chapter 16 introduces the concept of automatic matchmaking in real-time stream processing. Chapter 17 presents the design and implementation of several event stores for JEPC including a high-performance implementation for archiving enormous event streams on external storage. Chapter 18 describes our general approach to updating continuous queries on-the-fly. Chapter 19 shows the design and implementation of the JEPC query optimizer. Chapter 20 presents the framework for building parallel and federated JEPC infrastructures. Chapter 21 concludes this part.

15

Temporal and Spatiotemporal Pattern Matching over Event Streams

Outline

15.1 Introduction

During the last decade, several techniques for the real-time detection of complex patterns in high-volume event streams have been developed [Bar07, CM10, Dem07, DIG07, MM09, WTA10, WDR06]. All those techniques utilize the temporal dimension of event streams in two different ways [MM09, WDR06]. First, a pattern to detect defines an exact order in which specific events must occur. Second, a time window within which a pattern must occur completely can be specified. While this kind of pattern matching is sufficiently expressive in many application domains, there are also application domains that require more expressiveness [Li11]. In particular, more options to specify the temporal relationships between events are needed. Therefore, we present TPStream for detecting complex temporal patterns with interval-based semantics. Because of the importance of the traditional sequential pattern matching, TPStream supports patterns with point-based semantics as well. Consequently, TPStream is not a competing approach but a true extension to state-of-the-art event pattern matching and provides best of both worlds in a seamless manner. Existing declarative query languages for expressing sequential pattern matching queries must only be slightly modified to support also temporal pattern matching queries. We show an implementation of TPStream that respects the important performance requirements of EP namely high-volume throughput and real-time detection of patterns.

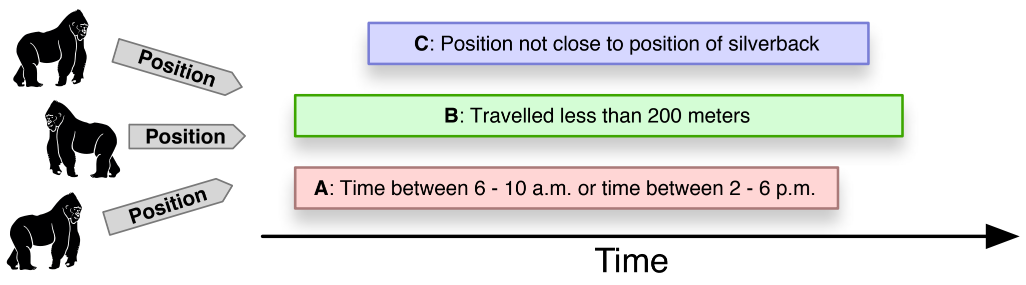

Position
Position
Position
C: Position not close to position of silverback
B: Travelled less than 200 meters
A: Time between 6 - 10 a.m. or time between 2 - 6 p.m.
Time

Figure 15.1: Temporal event pattern

Figure 15.1 illustrates a temporal event pattern a research group studying the dynamics in gorilla troops might be interested in. Gorillas form groups, called *troops*, around a single silverback that leads the troop [Gor]. But there are several situations in which gorillas leave their troop. Male gorillas are only accepted in their natal troop up to a certain age of about 11 years and then emigrate to start their own troop. Also, female gorillas emigrate from their natal troop to join another troop at around 8 years of age. Grown male gorillas are banished by the silverback when they try to take over leadership and ill gorillas are often left behind. A direct observation of a gorilla troop by researchers is time consuming, costly and would disturb the shy gorillas. Therefore, researchers could decide to attach a small GPS sensor to each gorilla of a single observed troop and to stay away from the whole home range of the observed troop. Every sensor sends periodically an event containing a unique identification, the current position as well as a timestamp. On basis of this very simple kind of information the researchers would want to be notified whenever a gorilla has left the troop. There are different temporal patterns for different kinds of leaving. The temporal pattern shown in Figure 15.1 specifically defines the situation in which a gorilla has been left behind after a rest period. Gorilla troops are resting together every midday and every night. When not resting, a gorilla troop searches for food within the home range (event `A` with time-interval semantics). Gorillas who traveled less than 200 meters in total during a food period likely collected food only for themselves or are ill (event `B` with time-interval semantics). An entire troop of gorillas has to collect much more food and travels a significant longer distance (at least 400 meters in general). To be truly sure a gorilla has left the troop or has been left behind, the distance to the silverback (as the center of the troop) must become greater than 100 meters at some point during a food period and is not allowed to become less than 100 meters again until the food period has ended (event `C` with time-interval semantics).

Intuitively, a temporal pattern defines temporal constraints (e.g., "overlaps" or "during") over a set of symbol events with time-interval semantics. Temporal patterns are more complex than sequential patterns that define order constraints over a set of symbol events with time-instant semantics (only "after", "before" and "at the same time" are expressible in this case). To support a temporal pattern query such as the one in the last paragraph, two features must be supported. First, arbitrary temporal constraints over events with time-interval semantics must be expressible. Considering the temporal pattern in Figure 15.1, it must be possible to define that event `A` and event `B` start at the same point in time, but event `B` ends after event `A`. Additionally, event `C` must have its first endpoint somewhere in event `B` and its second endpoint after the end of event `A`. Second, symbol events with time-interval semantics must be derived from events with time-instant semantics. In Figure 15.1 for instance, sensors are providing position events attached with a single timestamp only. However, a sequence of position events implicitly contains the desired symbol events `A`, `B` and `C`. Users must be able to specify how to derive symbol events from incoming events. Moreover, symbol events with time-interval semantics should be constructed within the responsible operator in an online manner. Assume for example a gorilla who left its troop changes its home range and meets the silverback never again or not until a couple of days has elapsed. Then, the time interval of event `C` becomes unbounded or very large and the second endpoint is never known or only after a couple of days. This would violate the requirement of EP that patterns are detected in the very moment they occur. However, to detect the pattern in Figure 15.1 in a timely manner, it is not necessary to know the second endpoint of event `C` exactly. As soon as it is known that the second endpoint of event `C` is after the second endpoint of event `A`, the pattern matches and must be detected. There is currently no method that supports all these features.

In this chapter, we present the design and implementation of the temporal pattern matching operator TPStream. It has been carefully designed to be not more complicated to use than sequential pattern matching operators, to derive user-defined symbol events with time-interval semantics from incoming events with time-instant semantics in an online fashion, to be able to process high volumes of events, to detect patterns in real-time, and to be a generalization of sequential pattern matching operators. Because of the latter point the detection of sequential patterns is also supported, which makes TPStream a true extension to sequential pattern matching.

Throughout this chapter we use the term *point event* to refer to an event being valid at only a single point in time t and the term *interval event* to refer to an event being valid at multiple successive points in time $t_s, \dots, t_e$ with $t_e > t_s$. For an event stream E, the expression $E(i)$ denotes the i-th event of the stream. As in the case of JEPC and almost all other pattern matching methods, pattern matching is performed on streams containing point events only. Also, we assume that event streams are ordered by time. However, there are approaches to handle out of order streams (e.g. [Li07, Li08, SW04]) which are fully compatible with the operators presented in this chapter so that our assumption simplifies presentation without restricting applicability in practice. Lastly, note that interval events are modeled by closed time intervals $[t_s : t_e]$ and that an event having a time interval with $t_s = t_e$ is called a point event although it is in interval event representation. As a consequence, a stream of events in interval representation can consist of both point and interval events. However, if this sharp distinction is not wanted in a query, it can be disabled (see Section 15.3.3).

15.2 ASEQ Operator

In this section, we discuss sequential pattern matching over point event streams and introduce the abstract operator ASEQ. The operator ASEQ is only virtual and covers popular operators in terms of expressiveness and semantics. In comparison to a single existing operator, ASEQ is designed to be equally or more expressive.

Today's pattern matching operators detect sequential patterns in the form of a set of specific point events that must occur in a specific order and within a certain time window [MM09, WDR06]. Besides this basic functionality, several operators support extensions such as alternative (event `A` or event `B` must occur), conjunction (event `A` and event `B` must occur simultaneously), negation (an event may not occur) or Kleene closure (an event may occur several times in a row) [Dem07, DIG07, MM09, WTA10].

15.2.1 Symbols and Predicates

A single element of a sequential pattern definition represents a point event having a certain property and is termed a symbol. The definition of ASEQ symbols is oriented towards the syntax and meaning of the pattern matching EPA of JEPC.

Definition 12 (ASEQ Symbol)**.** *An* ASEQ symbol *is emitted by point events on basis of a user-defined condition. Whenever an incoming point event fulfills the condition of an ASEQ symbol, it emits the associated ASEQ symbol and globally available variables can be set simultaneously. The syntax used to define an ASEQ symbol is as follows:*

$$\texttt{S AS } \varphi(E) \texttt{ [DO } var_1 = E.x\texttt{, } var_2 = E.y\texttt{, } \ldots \texttt{]}$$

The symbol `S` is emitted if and only if the user-defined Boolean-valued function φ, referred to as "ASEQ predicate", evaluates to `TRUE` (`AS` clause). This function is evaluated for each incoming event of an arbitrary but fixed point event stream E. In the optionally `DO` clause, values of attributes $(x, y, \ldots)$ of an event that emits `S` can be assigned to user-defined global variables $(var_1, var_2, \ldots)$. In the case of multiple symbols being defined on the same event stream, a single input event may emit multiple different symbols simultaneously.

Variables are used to correlate events. Event correlations are supported by all modern pattern matching operators and make pattern matching more powerful than regular expression matching. While some operators support event correlations indirectly via global variables as ASEQ, others (e.g., SASE [WDR06]) allow to access the payload of an event that emits a symbol directly via the symbol character. Since both approaches differ only in syntax, they are totally equivalent. An important parameter of symbol definitions are predicates which check events for user-defined properties.

Definition 13 (ASEQ Predicate)**.** *An* ASEQ predicate $\varphi : (\mathbb{D} \times \mathbb{T})^* \times \mathbb{D}^{\mathbb{V}} \times (\mathbb{D} \times \mathbb{T})^{\mathbb{N}} \to (\mathbb{B} \times \mathbb{T})^*$ *is a Boolean-valued higher-order function that maps every point event of some stream $E \in (\mathbb{D} \times \mathbb{T})^*$ with payload domain $\mathbb{D}$ and time domain $\mathbb{T}$ to a value in $\mathbb{B} := \{$`TRUE`, `FALSE`$\}$. Additionally, an ASEQ predicate can access the values of all global variables denoted by $\mathbb{V}$ and all previous events via a function `PREV`: $\mathbb{N} \to \mathbb{D} \times \mathbb{T}$.*

ASEQ predicates have access to not only the event $e \in E$ currently being processed and the values of all global variables being summarized in the set $\mathbb{V}$, but also previous events belonging to the stream E. While many existing operators do not support this functionality, ASEQ predicates have access to any previous event (as it is provided by Esper [Esp] that implements match recognize queries (MRQ) [Zem07] for example). In the query language of ASEQ, the function `PREV` is used as follows:

$$\texttt{PREV}(i).attr$$

This notation means that the i-th predecessor of the latest event is accessed and its value of the attribute $attr$ is obtained.

15.2.2 Sequential Patterns

Well-defined symbols (e.g., `A`, `B` and `C`) can be combined into a sequential search pattern (e.g., `ABC`). For instance, a typical sequential pattern definition specifies an increase or decrease in a series of numerical values (i.e., a time series). Formally, a sequential pattern is a regular expression over ASEQ symbols that specifies a (potentially infinite) set of totally ordered symbol sequences. The following grammar describes all legal ASEQ patterns P_{ASEQ} over ASEQ symbols:

$$P_{ASEQ} \leftarrow \emptyset \mid \texttt{S} \mid (\texttt{S}_1\texttt{;}\,\texttt{S}_2\texttt{;}\,\texttt{S}_3\texttt{;}\,\ldots) \mid (\texttt{S}_1\texttt{,}\,\texttt{S}_2\texttt{,}\,\texttt{S}_3\texttt{,}\,\ldots)$$
$$\mid \texttt{?S} \mid \texttt{S}^* \mid \texttt{S}^+ \mid \texttt{S}\{n \in \mathbb{N}\} \mid \texttt{!S} \mid P^1_{ASEQ}P^2_{ASEQ}$$

The empty pattern $\emptyset$ as well as a single symbol `S` are legal patterns (rules 1 and 2). Multiple symbols can be made alternatives (rule 3). Then, at least one of the symbols must be emitted at this position in the pattern. Multiple symbols can also be connected via a conjunction resulting in a pattern that requires all those symbols to be emitted at this position (rule 4). The question mark operator can be applied to a symbol for making it optional (rule 5). A symbol can be allowed to be emitted any number of times in a row by applying the Kleene star operator to it (rule 6). The Kleene plus operator is similar but requires the associated symbol to be emitted at least once (rule 7). A symbol can be required to be emitted exactly n times in a row by using the exact count operator (rule 8). The negation operator requires a symbol to be not emitted at this position (rule 9). Lastly, two ASEQ patterns can be combined into a sequence requiring the first pattern P^1_{ASEQ} to be followed by the second pattern P^2_{ASEQ} (rule 10). Of course, the presented grammar is powerful and not fully supported by a single existing pattern matching operator in general. For example, SASE+ supports the Kleene plus operator but not the Kleene star operator [DIG07].

15.2.3 Syntax

Listing 15.1 presents the query language of ASEQ that is similar to the query languages of existing operators for sequential pattern matching. In the FROM clause, one or more input event streams must be defined. An optional partitioning of the input event streams can be defined in the PARTITION BY clause. In the DEFINE clause, all symbols must be defined in the form of a list of symbol definitions. Each symbol definition must consist of a symbol condition in the form of an ASEQ predicate and may include the binding of any number of global variables according to Definition 12. In the PATTERN clause, a sequential search pattern over the defined symbols must be specified. The size of a time window within the pattern must occur completely can be defined in the optional WITHIN clause. If no size is defined, ASEQ uses a time window of infinite size by default. Finally, one or more global variables for composing the payload of the output events must be given in the RETURN clause.

```
 FROM          E_1, E_2, E_3, ...
[PARTITION BY  E_i.x, E_j.y, ...]
 DEFINE        S_A, S_B, S_C, ...
 PATTERN       P_ASEQ
[WITHIN        w]
 RETURN        payload of output event
```

Listing 15.1: ASEQ query language

15.2.4 Semantics

For every match contained in N input event streams $E_1, E_2, \dots, E_N$ the operator ASEQ contains exactly one event (m, t) in its output event stream:

$$ASEQ(E_1, E_2, \dots, E_N) := \sigma(\{(m_1, t_1), (m_2, t_2), \dots\})$$

Each output event (m, t) is produced by exactly one successful match, that is:

$$\begin{aligned}(m,t) := \forall i \in [1:N] : \exists E_i' \subseteq E_i : isPartition(E_i', E_i) \\ \wedge \exists F : F \succ getHistory\left(\bigcup_{i\in[1:N]} E_i'\right) \\ \wedge t = \max_{f\in F}(f.t) \wedge S = map(emit(), F) \\ \wedge m = match(S) \wedge m \neq \emptyset \wedge t - F(1).t \leq w\end{aligned}$$

The first requirement for every match is the existence of a valid partition of each input event stream. Whether a set of events E' is a valid partition of an input event stream E is determined by the Boolean-valued function *isPartition*. This function remains abstract for the following reasons. First, partitioning is done differently among different existing operators. A default definition of the function is given for TPStream in Section 15.3.5. Second, some existing operators let users define the function by themselves. For example, SASE+ provides different event selection strategies from which users can chose [DIG07]. Different selection strategies would not be possible if a fixed partitioning schema was defined in the semantics. The second requirement for every match is the existence of a substream F of the history of the union of all partitions. A history of a set of events is an event stream that contains all events in temporal order (payloads are allowed to be from different domains). A substream F of an event stream E (denoted by $\succ$) means that F begins and ends with arbitrary events of E and includes every event in-between. That is, F is a connected part of E. Then, every event of the substream F is replaced by the set of symbols it emits. The set of all emitted symbols for a given event is provided by the function *emit* which is applied via a map-operation to all events. If an event emits no events according to the symbol definitions, then *emit* returns the empty symbol $\square$. It indicates that no symbols are emitted but an event is present. For a stream of sets of symbols S the function *match* returns the payload of the output event m in case of a successful match:

$$match(S) := \begin{cases} payload\ of\ output\ event & \text{if } S \text{ matches } P \\ \emptyset & \text{otherwise} \end{cases}$$

Languages for pattern matching over event streams are more powerful than languages for regular expression matching [Agr08] and have a wide variety among different designs and implementations. To cope with that variety, the function *match* remains abstract. On the one hand, some existing operators detect a sequential pattern if the respective next symbol of a pattern occurs *eventually*. That is, events that potentially emit other symbols are allowed to be mixed in (eventually-followed-by semantics). On the other hand, some existing operators detect a pattern only if the symbols of a pattern occur *directly* one another. Thus, no other events are allowed to be in-between the events that emit two adjacent symbols of a pattern (directly-followed-by semantics). However, if such an operator supports Kleene star, then the eventually-followed-by semantics can also be achieved. Between these two different semantics, there exist operators such as TESLA [CM10] that allow to define a detection window for adjacent symbols. That is, a symbol must occur eventually after another symbol but within a user-defined time window (followed-by-within semantics). Because of these different existing semantics, ASEQ has the function *match* as a user-defined parameter in order to be able to implement every of the different proposed semantics. While the payload of an output event m is given by *match*, the timestamp is taken over from the last event of F. Finally, F must fit into a time window with user-defined size w.

ASEQ is defined to potentially report all matches contained in the input. However, some of its output events could be unwanted. While for some applications such as stock monitoring it is important to report every match, it is sufficient for a fire detection system to create only one output event for each individual fire [CM10]. Therefore, some operators provide different strategies users can chose from (e.g., via a `SKIP` clause such as in Esper [Esp], MRQ [Zem07], SASE+ [DIG07]). Those strategies can for example restrict an event to take part in at most one match. This means, the first correct match in that an event takes part is reported while all following matches in that the event takes part are not reported. Therefore, ASEQ allows to apply a user-defined filter σ to prevent correct but unwanted matches from entering the output stream.

The design of ASEQ simply integrates all extensions and semantics that can be found among popular pattern matching operators in order to cover them all. Note that a formal proof cannot be given, because many pattern matching operators were introduced only informally [CM10]. However, for verifying the expressiveness of TPStream towards sequential patterns it is only important to have a sequential pattern matching operator that is as powerful as possible for comparison reasons.

15.3 TPStream Operator

In this section, we introduce our novel temporal pattern matching operator TPStream which is a modified version of ASEQ. TPStream differs from ASEQ in the representations of symbols and patterns, but is designed in such a way that existing query languages can be easily upgraded to also support temporal pattern matching queries. A TPStream pattern defines temporal constraints over symbols in time-interval representation. By default, TPStream strictly distinguishes between symbols that are point events and symbols that are interval events.

15.3.1 Temporal predicates

While in ASEQ all emitted symbols are point events, emitted symbols can be point or interval events in TPStream. The latter can be achieved by replacing the ASEQ predicate by a so-called *temporal predicate* in the definition of a symbol.

Definition 14 (Temporal Predicate). *A* temporal predicate $\psi : (\mathbb{D} \times \mathbb{T})^* \times \Omega_\varphi \rightarrow (\mathbb{B} \times \mathbb{T} \times \mathbb{T})^*$ *is a Boolean-valued higher-order function that takes a point event stream $E \in (\mathbb{D} \times \mathbb{T})^*$ and an arbitrary ASEQ predicate $\varphi \in \Omega_\varphi$ as input and evaluates to an event stream where every event is a triple specifying a closed time interval within the output value $b \in \mathbb{B}$ of the ASEQ predicate does not change. The exact definition is as follows:*

$$\psi(E, \varphi) := \{\, (b, t_s, t_e) \mid \exists (p, t_s) \in E \ \wedge \ \exists (p, t_e) \in E$$

$$\wedge\, \forall (p, t) \in E : (t \in [t_s : t_e] \Rightarrow \varphi(p) = b)$$

$$\wedge\, \mathbb{T}_E = \{\, t \mid (p, t) \in E \,\} \ \wedge \ \forall (t'_s, t'_e) \in \mathbb{T}_E \times \mathbb{T}_E :$$

$$[t_s : t_e] \subset [t'_s : t'_e] \Rightarrow \exists (p, t) \in E : (t \in [t'_s : t'_e] \wedge \varphi(p) \neq b)$$

According to Definition 14, a time interval $[t_s : t_e]$ is generated by coalescing the time instants of consecutive point events that evaluate to the same output value b for a user-defined ASEQ predicate φ. At first, the formula requires the existence of a substream of E whose point events all evaluate to the same output value for the given ASEQ predicate. The rest of the formula requires that there is no larger substream of E having this property. Therefore, all generated time intervals are of maximum size. Algorithm 21 on page 216 shows the incremental evaluation of temporal predicates. An advantage of this definition is that user-defined predicates are carried over from ASEQ. In the syntax of the TPStream query language, only the function `TEMP` must be applied to a user-defined ASEQ predicate to make it a temporal predicate.

Consider an event stream E with integer numbers v as payload and an ASEQ predicate φ that checks whether v is even ($\varphi(E) :=$ `v % 2 = 0`). Listing 15.2 shows the `FROM` clause and the `DEFINE` clause that defines two symbols `X` and `Y`. While `X` is defined directly by the user-defined ASEQ predicate φ, `Y` is defined by the temporal version of φ. Table 15.1 shows all emitted `X` and `Y` symbols in the second and third columns for an instance of E given in the first column. Symbol `X` is emitted every time when the predicate evaluates to `TRUE`. In contrast, symbol `Y` is emitted for each range of time in which the predicate evaluates always to `TRUE`.

```
FROM    E
DEFINE X AS φ(E)
       Y AS TEMP(E,φ)
```

Listing 15.2: Definition of ASEQ predicates and temporal predicates

$E = (v, t)$	*Emitted* X *symbols*	*Emitted* Y *symbols*
(20,235)	(X,235,235)	(Y,235,240)
(18,240)	(X,240,240)	
(19,245)		
(21,250)		
(22,255)	(X,255,255)	(Y,255,265)
(24,260)	(X,260,260)	
(22,265)	(X,265,265)	
(19,270)		

Table 15.1: Comparison of ASEQ predicates and temporal predicates

Recall that the use of an ASEQ predicate always results in point events. The use of the temporal version of an ASEQ predicate leads to interval events in general, but it can also result in point events. This happens when the ASEQ predicate evaluates to TRUE for some event and to FALSE for its adjacent events. In TPStream, every symbol can be individually defined by either an ASEQ predicate or a temporal predicate.

The use of a temporal predicate affects the assignment of global variables. In the case of ASEQ predicates, values of an emitting event can be assigned to variables. But in the case of temporal predicates, multiple events may take part in the creation of a single emitted symbol. Only values of the first and last event that created a symbol can be used to set variables, because only these two events are present in every case. If a temporal predicate is used to define a symbol, users must specify not only the attribute to bind, but also an associated event in the DO clause. Therefore, the functions START and END are provided for selecting the first event and the last event. The following example presents the binding of variables in the context of temporal predicates.

```
FROM   Stream s
DEFINE A AS TEMP(s,φ1) DO var1 = START(s).x,
                          var2 = START(s).y
       B AS TEMP(s,φ2) DO finished_at = END(s).t
```

Listing 15.3: Temporal predicates and binding of variables

In Listing 15.3, the values of the event attributes x and y are assigned to the global variables var1 and var2 whenever symbol A is emitted. For both variables the first event that starts a symbol A is used. And whenever a symbol B is emitted, the timestamp of the last event is assigned to the variable finished_at. In the special case where a point symbol event is emitted, START and END refer to the same event.

15.3.2 Temporal Constraints

Since the temporal relationship between two interval events can be more complex than the relationship between two point events, order constraints according to the $<$ relation (or to the $\leq$ relation respectively) as in the case of ASEQ are not sufficient anymore. Instead, TPStream allows to use temporal constraints to specify the relationships between symbols. The upper part of Table 15.2 shows all 13 possible relationships between two interval events (known as Allen's operators [All83]). These temporal constraints clearly specify for two interval symbol events the exact order of the timestamps of their corresponding closed time intervals. But instead of defining temporal constraints via keywords (just as in Allen's schema [All83]) or in the form of Boolean expressions over endpoints aka endpoint encoding (just as in ISEQ [Li11]), we decided to adopt the encoding schema presented in [SG11]. This encoding schema does not require to remember a large set of keywords or to write long and complex Boolean expressions. On the contrary, it is a simple, compact and clear visual representation of temporal constraints. In this encoding schema, a temporal constraint is represented by a word of length four over two different characters. The two characters refer to defined symbols and are used to represent their timestamps. Therefore, each individual character must occur exactly twice per word. Reading from left to right, the first occurrence of a character in a word defines the position of the start timestamp of a symbol event and its second occurrence represents the position of the end timestamp of the same symbol event. For example, the temporal constraint `ABAB` means that the start timestamp of a symbol event `A` must be followed by the start timestamp of a symbol event `B`, the start timestamp of symbol event `B` must be followed by the end timestamp of symbol event `A`, and the end timestamp of symbol event `A` must be followed by the end timestamp of symbol event `B`. In terms of Allen's operators this is equivalent to the "`A` overlap `B`" relationship and in endpoint encoding this is equivalent to the Boolean expression $(\texttt{A}.t_s < \texttt{B}.t_s) \wedge (\texttt{B}.t_s < \texttt{A}.t_e) \wedge (\texttt{A}.t_e < \texttt{B}.t_e)$. In addition to the two characters, dots can be used in words to express equality. For instance, the word `AA.BB` means that the end timestamp of `A` must be equal to the start timestamp of `B` (i.e., $\texttt{A}.t_e = \texttt{B}.t_s$). In contrast to Allen's operators, we further distinguish between interval events and point events. In the upper part of Table 15.2, `A` and `B` are interval events. That is, it is always $\texttt{A}.ts < \texttt{A}.te$ and $\texttt{B}.ts < \texttt{B}.te$. The additional 10 temporal constraints in the center part of Table 15.2 can be used to express that `A` or `B` must be a point symbol event, and the additional three temporal constraints in the lower part of Table 15.2 can be used to express that `A` and `B` must be point symbol events.

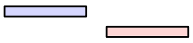
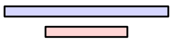
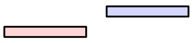
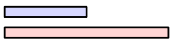
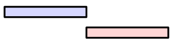
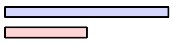
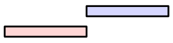
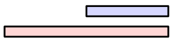
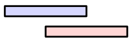
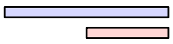
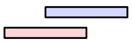
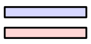
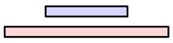
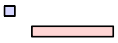
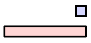
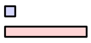
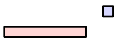
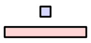
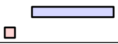
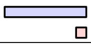
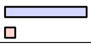
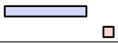
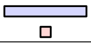
Encoding	*Pictorial*	*Encoding*	*Pictorial*	*Encoding*	*Pictorial*
`A` and `B` are interval events (Allen's operators)					
`AABB`		`ABBA`		`BBAA`	
`A.BAB`		`AA.BB`		`A.BBA`	
`BB.AA`		`BAA.B`		`ABAB`	
`ABA.B`		`BABA`		`A.BA.B`	
`BAAB`					
`A` is point event, `B` is interval event					
`A.ABB`		`BB.A.A`		`A.A.BB`	
`BBA.A`		`BA.AB`			
`A` is interval event, `B` is point event					
`B.BAA`		`AA.B.B`		`B.B.AA`	
`AAB.B`		`AB.BA`			
`A` and `B` are point events					
`A.AB.B`		`B.BA.A`		`A.A.B.B`	

Table 15.2: Encoding of temporal constraints in TPStream

15.3.3 Temporal Patterns

A temporal constraint (TC) is one of the 26 basic relationships shown in Table 15.2:

$$TC \leftarrow \texttt{AABB} \mid \texttt{ABBA} \mid \ \texttt{...} \ \mid \texttt{A.A.B.B}$$

Each TC clearly specifies the two symbols it refers to (determined by the two characters which are used in the word), the required types (point or interval event) as well as the exact order of timestamps. However, a single TC for a pair of symbols is insufficient in many situations. To express also indefinite relationships [NB95] between two interval events, TPStream allows to combine TCs into a temporal constraint list (TCL):

$$TCL \leftarrow \emptyset \mid TC \mid TC; TCL$$

Every non-empty TCL is either a single TC or a list of multiple TCs being connected by logical or-operators (;). All TCs in a TCL must refer to the same two symbols. Thus, a TCL defines a disjunction of TCs for two symbols and allows for expressing all relationships between two symbols that can also be expressed in endpoint encoding [Tom96]. A TCL evaluates to `TRUE` if and only if at least one of its TCs evaluates to TRUE. For example, it might be important that some symbol begins before another symbol while their end timestamps are permitted to be in any order (i.e., the relationship between their end timestamps is indefinite). In the use case of the introductory section for instance, it is only important that symbol C starts in symbol B. Whether symbol C ends before, at the same time or after symbol B does not matter. In this case, all possible TCs in which symbol B starts in symbol C can be combined into the TCL BCBC;BCCB;BCB.C. Another important situation in which multiple different TCs must be combined is if it does not matter whether a symbol is an interval event or a point event. For example, AABB;A.ABB specifies that symbol A is completely before symbol B independent of its type. In most cases, specifying the relationship between only two symbols is insufficient. Therefore, TPStream allows to specify a separate TCL for each pair of defined symbols and to combine them into a temporal pattern (TP):

$$TP \leftarrow \emptyset \mid (TCL) \mid (TCL), TP \mid !\mathtt{S}, TP$$

A non-empty TP is either a single TCL or a list of multiple TCLs being connected by logical and-operators (,). For each pair of symbols at most one corresponding TCL can be contained in a TP. If there is no corresponding TCL for a specific pair of symbols, then every relationship between the symbols is accepted. A TP evaluates to TRUE if and only if all of its TCLs evaluate to TRUE. Because TCLs are connected via logical and-operators in a TP, every relationship between two or more symbol events that can be expressed in endpoint encoding (which is studied in [NB95] and used in ISEQ [Li11]) can also be expressed in TPStream. Additionally, TPStream allows to negate single symbols via the !-operator in a temporal pattern (then, the associated symbol may not occur). Sometimes, a situation of interest can only be described by multiple alternative TPs. Therefore, TPs can be combined into a temporal pattern list (TPL):

$$TPL \leftarrow \emptyset \mid (TP) \mid (TP); TPL$$

Each non-empty TPL is either a single TP or a list of multiple TPs being connected by logical or-operators (;). A TPL evaluates to TRUE if and only if at least one of its TPs evaluates to TRUE. Note that it is possible to define invalid TPs and TPLs (e.g., A.t_s before B.t_s and B.t_s before C.t_s and C.t_s before A.t_s). For the rest of the chapter we assume that TPs and TPLs are always well-defined.[1]

[1]In practice, usually the query compiler is in charge of checking the validity of pattern definitions. In the special case of TPStream, the component testing for constraint satisfaction (see Section 15.6) can easily determine whether a temporal pattern is valid.

15.3.4 Syntax

In comparison to ASEQ, TPStream only provides extended functionality for defining symbols and a new grammar for expressing patterns. Therefore, the overall structure of the query language can remain the same as shown by Listing 15.4.

```
 FROM           E_1, E_2, E_3, ...
[PARTITION BY   E_i.x, E_j.y, ...]
 DEFINE         S_A, S_B, S_C, ...
 PATTERN        TPL
[WITHIN         w]
 RETURN         payload of output event
```

Listing 15.4: TPStream query language

The parameters of the FROM, PARTITION BY, WITHIN and RETURN clauses are not changed at all in the query language of TPSteam. But the query language of TPStream allows to define a symbol via either a pure ASEQ predicate or a temporal predicate in the DEFINE clause. In the PATTERN clause of the query language of TPStream, a TPL must be given instead of an ASEQ pattern. For example, a query for detecting the temporal pattern shown in Figure 15.1 can be defined as shown in Listing 15.5.

```
FROM         Sensors s
PARTITION BY s.id
DEFINE       A AS TEMP(s.t IN [6 a.m. : 10 a.m.] OR s.t IN [2 p.m. :  6 p.m.])
               DO gorilla     = START(s).id,
                  gorilla_pos = START(s).pos
             B AS TEMP(abs(s.pos-gorilla_pos) < 200)
             C AS TEMP(abs(s.pos-posSilverback())) > 100)
               DO time_of_leaving = START(s).t
PATTERN      A.BAB,(BCBC;BCCB;BCB.C),ACAC
WITHIN       6 HOURS
RETURN       gorilla, time_of_leaving
```

Listing 15.5: Finding gorillas that left their troop

Assume that the schema of the event stream Sensors used in Listing 15.5 has the attributes pos keeping the position of a sensor and id keeping its identification. This stream is partitioned by id, because the pattern must be searched for each monitored gorilla individually. Symbol A is generated exactly two times per day, because there are one period ranging from 6 a.m. to 10 a.m. and one period ranging from 2 p.m. to 6 p.m. every day. The generation is simply triggered by the clock contained in

`Sensors`. In the concrete example, symbol `B` is correlated with symbol `A` via the variable `gorilla_pos`. As long as the current position has a distance less than 200 to the position when `A` started, `B` is expanded. In case of `C`, the current position of the silverback is obtained via a function `posSilverback` and compared to the current position of the sensor. We determine the point in time when the current position has a distance greater than 100 to the position of the silverback for the first time. This is the point in time of leaving. Therefore, the timestamp of the first event of `C` is assigned to the variable `time_of_leaving`. The pattern requires symbol `B` to start with symbol `A` but to end after symbol `A`. Symbol `C` must start in symbol `B`, but it does not matter whether `C` ends before, at the same time, or after symbol `B`. The beginning of `C` must be after the beginning of `A` and the end of `C` must be after the end of `A`. Finally, the pattern must occur within 6 hours and every detected match leads to a new output event having the identification of the sensor and the point in time of leaving as payload.

15.3.5 Semantics

The output of TPStream is a stream containing one event for every detected match:

$$TPStream(E_1, E_2, \ldots, E_N) := \sigma(\{(m_1, t_1), (m_2, t_2), \ldots\})$$

Each output event (m, t) is produced by exactly one individual match, that is:

$$\begin{aligned}(m,t) := \forall i \in [1:N] : \exists E'_i \subseteq E_i : isPartition(E'_i, E_i) \\ \wedge \exists F : F \succ getHistory\left(\bigcup_{i\in[1:N]} E'_i\right) \wedge \exists \overline{S} : \forall s \in \overline{S} : \exists F' \succ F : s \in emit(F') \\ \wedge\, m = match(\overline{S}) \wedge m \neq \emptyset \wedge\, t = \max_{f\in F}(f.t) \;\wedge\; t - F(1).t \leq within\end{aligned}$$

To achieve the same flexibility as ASEQ, the Boolean-valued function *isPartition* can be user-defined again. However, in our implementation of TPStream for JEPC we use the commonly used partitioning schema by default:

$$\begin{aligned}isPartition(E', E) := \forall i : \forall j : (e'_i \in E' \wedge e'_j \in E') \Rightarrow \\ (\forall k \in [1:n] : e'_i.x_k = e'_j.x_k \;\wedge\; \forall e \in E \setminus E' : \exists l \in [1:n] : e.x_l \neq e'_i.x_l)\end{aligned}$$

Let the attributes $x_1, x_2, \ldots, x_n$ of E be referenced in the `PARTITION BY` clause. Then, two conditions must hold. First, every event of a valid partition E' must have the same value for each referenced attribute. Second, every event of E that is not contained in E' must have a different value for at least one referenced attribute.

Again, there must be a fixed substream F of the history. But in contrast to ASEQ, the function *emit* returns all emitted point and interval symbol events for a substream F' of F with respect to Definition 14. In particular, *emit* returns only intervals that are of maximum size and completely contained in F'. While the function *match* remains abstract in the semantics of ASEQ, it is clearly defined in the semantics of TPStream:

$$match(\overline{S}) := \begin{cases} payload\ of\ output\ event & \text{if } TPL(\overline{S}) = true \\ \emptyset & \text{otherwise} \end{cases}$$

The input of *match* is a set of point and interval symbol events emitted by events of F. Every TPL can be translated into a Boolean expression over timestamps of symbols as follows. Each TC defines the order of the timestamps of two symbols and can be translated into a Boolean expression (see Section 15.3.2). Every TCL connects all of its TCs by logical disjunctions ($\vee$). Each TP connects all of its TCLs by logical conjunctions ($\wedge$). Finally, every TPL connects all of its TPs by logical disjunctions ($\vee$). At the end, a valid Boolean expression over timestamps is created. This expression can be evaluated for the timestamps of the symbols in the input. If a certain configuration evaluates to `TRUE`, the payload of the output event is generated and returned.

The semantics of the within condition and the computation of the timestamp t are the same as in ASEQ. Also, a user-defined filter σ can be applied to the output which contains every possible match. This gives the same degree of flexibility as ASEQ.

15.4 Expressing Sequential Patterns

In this section, we outline how to express sequential patterns via temporal patterns. For each rule in the ASEQ pattern grammar we present an equivalent TPStream pattern definition. We also show how to express the different semantics that exist among different operators for sequential pattern matching (namely eventually-followed-by, directly-followed-by, and followed-by-within semantics). Without loss of generality, we assume that incoming events have unique timestamps for ease of presentation.

Sequence. The sequence `AB` of two symbols `A` and `B` can be expressed as follows in TPStream. First, the symbols `A` and `B` are defined by exactly the same predicates φ_A and φ_B that would be used in ASEQ. Thus, emitted symbols are always point events. Second, the order constraint `AB` is expressed by the temporal constraint `A.AB.B`. Listing 15.6 shows the final translation.

```
DEFINE  A AS φ_A, B AS φ_B
PATTERN A.AB.B
```

Listing 15.6: A followed by B

Conjunction. To define a pattern in which multiple symbols $\{\mathtt{A}, \mathtt{B}, \mathtt{C}, \ldots\}$ must occur simultaneously, every symbol is defined via the predicate that would be used in ASEQ. Then, the temporal constraint `X.X.Y.Y` is used for each pair of symbols to express the conjunction. However, because of transitivity of the conjunction a temporal constraint must not be added for every pair of symbols. Instead, a transitive reduction can be performed resulting in a minimal set of necessary temporal constraints to express the conjunction of more than two symbols. Listing 15.7 shows the final translation.

```
DEFINE  A AS φ_A, B AS φ_B, C AS φ_C, ...
PATTERN (A.A.B.B), (B.B.C.C), ...
```

Listing 15.7: Conjunction

Alternative. An alternative is used to define that at least one symbol of $\{\mathtt{A}, \mathtt{B}, \mathtt{C}, \ldots\}$ must occur. In TPStream, multiple TPs each requiring another symbol at the position of the alternative are defined. Finally, all TPs are combined into a TPL. Listing 15.8 presents the translation of the ASEQ pattern $\mathtt{X}(\mathtt{A};\mathtt{B};\ldots)\mathtt{Z}$.

```
DEFINE  X AS φ_X, Z AS φ_Z, A AS φ_A, B AS φ_B, ...
PATTERN ((X.XA.A),(A.AZ.Z)); ((X.XB.B),(B.BZ.Z)); ...
```

Listing 15.8: Alternative

Kleene Plus. In TPStream, the Kleene plus operator can be expressed by exchanging the ASEQ predicate in the definition of the associated symbol for its temporal version by simply applying the function `TEMP` to it. Because an emitted symbol can now be a point or an interval event, the TP must cover both cases. The translation of the ASEQ pattern $\mathtt{AB}^{+}\mathtt{C}$ is shown in Listing 15.9.

```
DEFINE  A AS φ_A, B AS TEMP(φ_B), C AS φ_C
PATTERN (A.ABB;A.AB.B), (BBC.C;B.BC.C)
```

Listing 15.9: Kleene plus

Negation. A negated symbol must be defined as a regular symbol in order to specify its properties and exact position within a pattern. Then, in every TP that contains the negated symbol the negation operator must be applied in addition. For example, the pattern `A!BC` is expressed in TPStream as shown in Listing 15.10.

```
DEFINE  A AS φ_A, B AS φ_B, C AS φ_C
PATTERN (A.AC.C), (A.AB.B), (B.BC.C), !B
```

Listing 15.10: Negation

Exact Count. The exact count operator is just syntactic sugar in ASEQ. For example, `A{5}` is a shortcut for the pattern `ABCDE` in which all symbols are defined by exactly the same predicate φ. A translation of sequences is already presented above.

Question Mark. The question mark operator requires a symbol to occur not or exactly once. Both alternatives can be expressed in the form of alternative TPs being combined into a TPL. Listing 15.11 shows the translation of the ASEQ pattern `A?BC`.

```
DEFINE  A AS φ_A, B AS φ_B, C AS φ_C
PATTERN ((A.AB.B),(B.BC.C)); (A.AC.C)
```

Listing 15.11: Question mark

Kleene Star. Similar to the question mark operator, the application of the Kleene star operator results in two different patterns. In one pattern, the symbol a Kleene star is applied to may not occur. In the other pattern, the symbol a Kleene star is applied to may occur multiple times in a row but must occur at least once (equivalent to the Kleene plus operator). For example, the pattern $\mathtt{AB}^*\mathtt{C}$ can be substituted by the alternative patterns `AC` and $\mathtt{AB}^+\mathtt{C}$. For both patterns there already are translations into the query language of TPStream. The combination of the translated patterns into a TPL is equivalent to the Kleene star operator.

With the translations presented so far, every ASEQ pattern can be expressed in the form of a temporal pattern. However, the resulting temporal patterns inherently implement the eventually-followed-by semantics. To express that two symbols must be emitted by two adjacent events (directly-followed-by semantics), a temporal pattern can be extended as shown in Listing 15.12.

```
DEFINE  A AS φ_A DO begin = t
        B AS φ_B ∧ PREV(1).t = begin
PATTERN A.AB.B
```

Listing 15.12: A directly followed by B

In comparison to Listing 15.6, Listing 15.12 extends the condition of symbol B. In addition to the user-defined condition φ_B, it is required that the directly preceding event emitted symbol A. This can be achieved by remembering the timestamps of the events that emitted symbol A. Then, it can be checked via PREV in the definition of symbol B whether the directly preceding event emitted symbol A. Lastly, the followed-by-within semantics can also be expressed. The query in Listing 15.13 defines a sequence in which A is followed by B within a time window of size w. Again, the condition of symbol B is extended. But this time, the timestamp of the current event is compared to the timestamp of emitted A symbols that can be accessed via a global variable.

```
DEFINE  A AS φ_A DO begin = t
        B AS φ_B ∧ t - begin <= w
PATTERN A.AB.B
```

Listing 15.13: A followed by B within w

The presented translations for every rule in the grammar of ASEQ as well as for the different semantics serve as building blocks for arbitrary sequential patterns. To express a complex ASEQ pattern, every atomic part can be translated individually first. Then, the translated parts can be combined step by step into the final TPL by specifying the exact order and semantics (eventually-followed-by, directly-followed-by, followed-by-within) for every pair of adjacent symbols. Note that TPStream is more powerful than ASEQ since it is possible to have both sequential and temporal parts in a single TPStream pattern definition. Also note that different sequential semantics can be mixed within a single TPStream pattern definition.

15.5 Expressing Spatiotemporal Patterns

The spatial capabilities of existing stream processing systems are quite limited, because spatial functionality is not natively supported. Of course, the position of a fixed or moving object can be encoded as a series of numeric attributes (e.g., latitude, longitude and height) defining a point in some coordinate system, and simple operations such as computing the distance between two points can be reduced to an arithmetic expression. In simple cases, this method may serve as workaround. For instance, the motivating example in Section 15.1 can be implemented this way. However, the workaround is not sufficient anymore when it comes to more complex geometries (e.g., lines, areas) [Güt00] and more complex spatial operations (e.g., checking whether a point is in or close to a certain area, computing the intersection of two areas).

The integration of a stream processing system into applications that deal with complex spatial data and complex spatial operations requires native support of spatial data types and spatial operators, which process the spatial datatypes properly, in order to support complex spatial queries. Simple Feature Access (SFA) [Her11] defines all necessary data types and operations for representing and processing spatial data, is standardized by both OGC and ISO, and is platform-independent. Among others, SFA specifies well-known text (WKT) which is a markup language for describing vector geometries. The advantage of WKT is that a complex geometry can be encoded as a single ordinary string. This allows spatial operators to seamlessly work together with non-spatial operators. Spatial operators can compile WKT strings into some object representation internally, process the geometries, and then encode all resulting geometries again into WKT strings for the output. Non-spatial operators cannot interpret or process WKT strings, but because a string is a primitive data type they can simply pass through WKT strings and do not become blocking therefore.

JEPC can be extended by new spatial EPAs as follows [HKS15]. The core already provides the data type `DataType.GEOMETRY` for keeping WKT strings (see Table 7.1 on page 52). Thus, spatial EPAs can easily identify spatial attributes and all others handle attributes of type `DataType.GEOMETRY` as attributes of type `DataType.STRING` by default. Recall that JEPC allows to bring in user-defined EPAs (see page 59). For example, we implemented TPStream in the form of a user-defined EPA (UEPA) for JEPC. Because UEPAs are defined by arbitrary Java code, functionality for compiling WKT strings and processing the resulting geometry objects can be implemented. However, we prefer to use a third party library such as JTS Topology Suite [JTS] or GeoTools [Geo] rather than implementing the SFA specification by ourselves.

The approach described in the last paragraph can be used to add every kind of spatial operator as UEPA to JEPC [Ali10]. As an example and proof of concept, we extended TPStream, which already existed as UEPA, in order to detect complex spatiotemporal patterns [Erw04, SG11] besides temporal patterns. In particular, our extended version of TPStream allows to apply all spatial operators of SFA to spatial attributes in the definition of symbols. This enables the manipulation of geometries and, more importably, the use of spatial predicates. For instance, complex spatiotemporal patterns in the trajectories of moving objects can be detected by TPStream now.

To give an impression of the interesting and powerful class of spatiotemporal pattern queries, we take over a use case being presented in [Sak10] for TPStream and streaming data. In this use case, the trajectories of snow storms are monitored and some particular snow storms are selected by a spatiotemporal pattern query being described in the following. The used spatiotemporal pattern query detects all strong snow storms that first hit Berlin and then hit Dresden in Germany within 4 hours. From a spatial point of view, this means that a moving area representing a snow storm must have intersected two fixed areas representing the cities Berlin and Dresden. In addition, it is required that the snow storm was strong when it hit both cities. The exact meaning of *strong* is specified on basis of the speed of a snow storm that must be equal to or greater than 80.0 km/h. From a temporal point of view, only snow storms that first hit Berlin and then hit Dresden are of interest. Moreover, when the snow storm hit the cities and as long as the snow storm was raging in the cities, it must have been strong. Lastly, the just defined spatiotemporal pattern must occur completely within a time window of size 4 hours.

```
(SELECT *
 FROM   SnowStorms s
 MATCH_RECOGNIZE_TEMPORAL (
   MEASURES     START(A).id AS storm_id, END(B).polygon AS storm_polygon
   WITHIN       4 HOURS
   PARTITION BY s.id
   PATTERN      AABB,(CAAC;CAA.C),(DBBD;DBB.D)
   DEFINE       A AS TEMP(intersects(s.polygon, Berlin)),
                B AS TEMP(intersects(s.polygon, Dresden)),
                C AS TEMP(s.speed >= 80.0),
                D AS TEMP(s.speed >= 80.0)
 )
) AS SnowStormExample
```

Listing 15.14: Spatiotemporal pattern query in JEPC-QL

Listing 15.14 shows the query being ported to TPStream and expressed in JEPC-QL. The query consumes an event stream `SnowStorms` that consists of position updates of snow storms. Each position update contains the identification of the corresponding snow storm (`id`), its current speed (`speed`) and its current area represented by a polygon encoded as a WKT string (`polygon`). `Berlin` and `Dresden` are constant WKT strings that define the areas of the cities in the form of polygons. For more examples which are also supported by TPStream in case of streaming data see [SG11].

Because we already had extended Boolean expressions to support spatial operations, we carried over our implementation for TPStream to our native implementations of the filter and the correlation EPAs. For the aggregation EPA, we extended our native implementation to support the spatial aggregates `UNION` (gives the union of a set of polygons), `DISTANCE` (gives the length of the corresponding linestring of a sequence of points) and `TRAJECTORY` (gives the trajectory as linestring of a sequence of points). Thus, JEPC supports spatial functionality in all basic EPAs via the native EP provider and additionally offers spatiotemporal pattern matching by now.

15.6 Implementation

In Figure 15.2, we give a high-level view on the internals of our TPStream implementation in the form of an UEPA for JEPC. It can be used to detect a single TP over a set of input event streams. If the temporal pattern is a TPL consisting of multiple alternative TPs, one physical instance of TPstream is currently created for each single TP. Processing a complex TPL by a single physical TPStream operator most likely would give better performance and is on our agenda for future work.

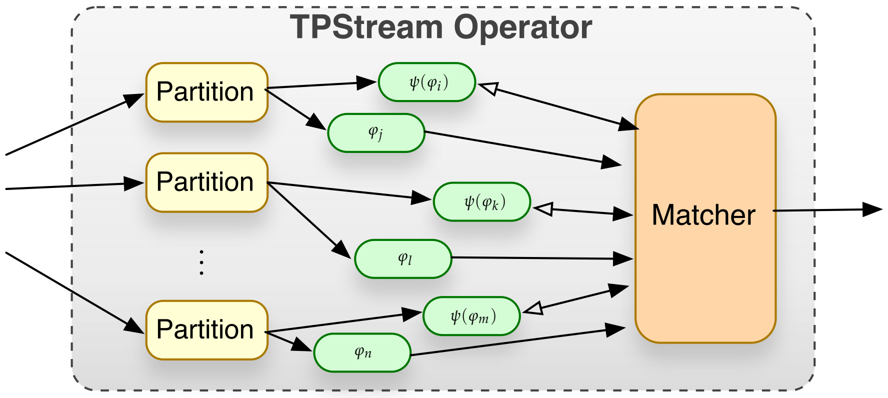


Figure 15.2: Architecture of TPStream

The leftmost arrows in the figure represent multiple input event streams. Every incoming event is copied into its associated partition first. If no partitioning is specified, then there is exactly one partition per event stream as illustrated in the figure. Each partition has an instance of all compatible predicates (a predicate is defined for a specific event stream, i.e., event type) for deriving symbols in an online manner. The figure shows one temporal predicate ψ and one ASEQ predicate φ per input event stream. Every event entering a partition is forwarded to all instantiated ASEQ predicates and temporal predicates.[2] In the case of ASEQ predicates, symbols are emitted directly as point events and forwarded to the matcher component. And in the case of temporal predicates, the creation of symbols is performed via Algorithm 21.

Algorithm 21: TEMPORALPREDICATE((p,t))

```
Input: Event: $(p,t)$
Data: ASEQ Predicate: $\varphi$, Event: $buffer$, $lastEvent$
1  if $buffer.p = \varphi(p)$ then
2      $buffer.t_e \leftarrow t$; // extend time interval of symbol event
3      $lastEvent \leftarrow (p,t)$;
4  else
5      if $buffer.p =$ TRUE then // finish symbol event
6          SETVARIABLESEND($lastEvent$);
7          FORWARD($buffer$);
       // start new symbol event
8      $buffer \leftarrow (\varphi(p),t,t)$;
9      if $buffer.p =$ TRUE then
10         SETVARIABLESSTART($(p,t)$);
```

Algorithm 21 determines all time intervals of maximum size during which φ is always fulfilled (see Definition 14). The endpoints of those time intervals are used as timestamps for symbol events being emitted. For generating time intervals in an online manner a buffer is used. This buffer keeps the time interval currently being in creation and the corresponding output value of φ as payload. If a new input event that does not change the output value of φ streams in, the end timestamp of the buffered symbol event is replaced by the timestamp of the input event (lines 1–3). Thus, time intervals are extended incrementally as long as the output value of φ does not change. Every

[2]Though parallel processing of partitions would be an opportunity to improve performance, our current implementation still runs sequentially. Optimizing the performance is part of our future work.

input event for which the predicate changes its output value results in the creation of a new symbol event (lines 4–10). If the symbol event created so far was produced by input events for which the predicate evaluates to `TRUE`, then this symbol event must be finished. The last event that participated in its creation is used for setting global variables defined via `END` (line 6) and the symbol event keeping the finished time interval is forwarded to the matcher component (line 7). Then, a new symbol event is started on basis of the input event (line 8). If this new symbol event will be forwarded to the matcher component eventually (i.e., the ASEQ predicate is satisfied), then the first input event is used for setting global variables defined via `START` (line 10).

As indicated by the bi-directed arrows in Figure 15.2, the matcher component not only receives finished symbol events from the instantiated predicates, but also has access to the buffer of every instantiated temporal predicate. This is of utmost importance for supporting real-time detection. For example, consider the query in Listing 15.5. A successful match requires the end timestamp of a symbol event `C` to be greater than the end timestamp of a symbol event `A`. However, it is not necessary to know the exact value (thus, it must not be completed). As soon as the end timestamp of a symbol event `C` in the buffer of the associated temporal predicate becomes greater than the end timestamp of a symbol event `A` (and all other TCLs of the pattern hold), our matcher already produces the output event. Without this live generation of time intervals there arise two critical problems. A matcher that requires completed time intervals will block until the interval of a symbol event `C` is completed. Consider a gorilla, which has left the troop, meets the silverback after a couple of days again. Then, not until this point in time symbol event `C` is finished. In a worst case scenario, the gorilla never meets the silverback again. As a consequence, there would be no completion of symbol event `C` and the match would never be detected. These issues violate the real-time requirements of event processing and are avoided in our implementation by giving the matcher access to the buffers.

Almost all sequential pattern matching operators use automata in the implementation of the matcher component (e.g., SASE [WDR06], Cajuga [Dem07]), because searching for an ASEQ pattern on a stream of symbols is similar to regular expression matching. In contrast, temporal pattern matching is related to constraint satisfaction problems [Kum92] (ZStream [MM09] shows that sequential pattern matching can be interpreted in the same way). Because we currently use a single TPStream operator for every TP in a TPL and because every TC is binary, the TP to detect can be represented as a constraint graph. For example, Figure 15.3 shows the constraint graph for the temporal pattern `(ABBA,(ACCA;A.CCA; A.CA.C), ADDA,CCEE,(DDEE;DD.EE))`.

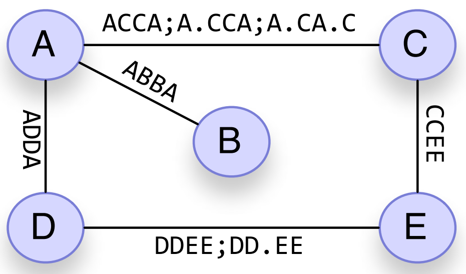


Figure 15.3: Constraint graph of temporal pattern

The user-defined symbols are the vertices of a constraint graph. If there is a user-defined TCL for a pair of symbols, those symbols are connected by an undirected edge being labeled with all TCs of the TCL. A constraint graph evaluates to `TRUE` if and only if at least one TC of every edge is satisfied. In contrast to a real constraint satisfaction problem where configurations are searched, we must only check whether a given candidate configuration satisfies the constraint graph. In general, there are multiple candidate configurations that must be checked at a time. The set of all candidate configurations is determined as follows. Within the time window there can be any number of emitted symbol events (thus, a set) for each defined symbol. With respect to the above example, there is a set X_A keeping all symbol events associated with symbol `A`, a set X_B keeping all symbol events associated with symbol `B`, and so on. The Cartesian product of all those sets gives the set of all candidate configurations that must be checked. Because the number of candidate configurations can be large and because of the fact that a candidate configuration disqualifies as soon as one TCL (i.e., an edge in the constraint graph) is not satisfied, our implemented matcher generates the set of all configurations lazy and tries to detect non-matching configurations as early as possible during generation. From an algorithmic point of view, we start with generating the Cartesian product of only two sets of symbol events and check the corresponding TCL (if it exists). In the best case, no or only few pairs of symbol events will remain. Then, the Cartesian product of all remaining pairs and another set of symbol events is created. Again, corresponding TCLs are checked already by now to reduce the next set of intermediate configurations. This iterative procedure is continued until the last set of symbol events has been processed. Changing the order of the sets of symbol events, the sizes of the sets of intermediate configurations may vary. Obviously, there is an optimal order that minimizes the sizes of all sets of intermediate configurations so that a minimal number of checks must be performed.

Note that this kind of problem is similar to the problem of join ordering [Moe09]. The sets of symbol events are the input relations and the TCLs are join conditions referring to a pair of relations. Our implemented matcher component orders the sets of symbol events according to a simple heuristic that is based on the following assumption. The more TCLs which can be checked there are, the less is the size of the next set of intermediate configurations. Therefore, our strategy is as follows and requires the constraint graph. The first set of symbol events is the corresponding set of the symbol being involved in more TCLs than all other symbols (i.e., the symbol having the most edges in the constraint graph). The second set is one of the neighbors of the first set. This guarantees that there will be at least one TCL for pruning the next set of intermediate configurations. In general, a symbol has more than one neighbor in the constraint graph. Then, we select the neighbor which has the most edges among all neighbors. The next sets are selected in the same way as the second one. That is, the third symbol is a neighbor of the second symbol having the most edges among all neighbors of the second symbol, and so on and so forth.

To illustrate the general principle and to clarify all ambiguities, consider the constraint graph given in Figure 15.3. According to our strategy, the matcher component would derive an order of sets of symbol events as follows. The set of symbol events X_A associated with symbol `A` is the first, because symbol `A` is the only symbol being involved in three TCLs. Next, we must select one of its neighbors. Because the symbols `C` and `D` have two neighbors each and symbol `B` only one, the symbols `C` and `D` are preferred over symbol `B`. After all, we select the set X_D associated with symbol `D`, because the TCL of symbol `A` and symbol `D` consists of only one TC while the TCL of symbol `A` and symbol `C` consists of three TCs and is considered less restrictive therefore. However, if the numbers of TCs were equal, we would select one of the two sets by random. Then, we select the set X_E, because symbol `E` is the only remaining neighbor of symbol `C`. Lastly, we have to turn back to the remaining neighbors of symbol `A` that are the only remaining symbols. According to our heuristic, we prefer symbol `C` over symbol `B` so that the final optimized order is $X_A \times X_D \times X_E \times X_C \times X_B$.

15.7 Evaluation

Based on our implementation of TPStream, we conducted the following experiments on a workstation with an Intel i7-2600 3.4 GHz processor (only a single core was used) and 8 GiB main memory running Oracle JRE 1.7.0_13.

15.7.1 Optimizing the Evaluation Order

In our first experiment, we examined the effect of our strategy to optimize the evaluation order of TCLs by using the matcher component in isolation. The setup of the experiment was as follows. We used a fixed set of 8 user-defined symbols and selected 16 TCs for these symbols. The two symbols of a TC as well as the TC itself were randomly chosen. For each single run of the experiment we generated 10,000 sets each containing exactly x symbol events for each user-defined symbol. We tested different values of x in this experiment. The time intervals of the symbol events were randomly chosen from $[10 : 100]$. We put all 10,000 sets one another into two instantiated matcher components. One matcher component evaluated TCLs in a random order and the other matcher component evaluated TCLs in an optimized order using our ordering strategy. We measured the time each matcher component needed for checking all configurations contained in each of the 10,000 test sets.

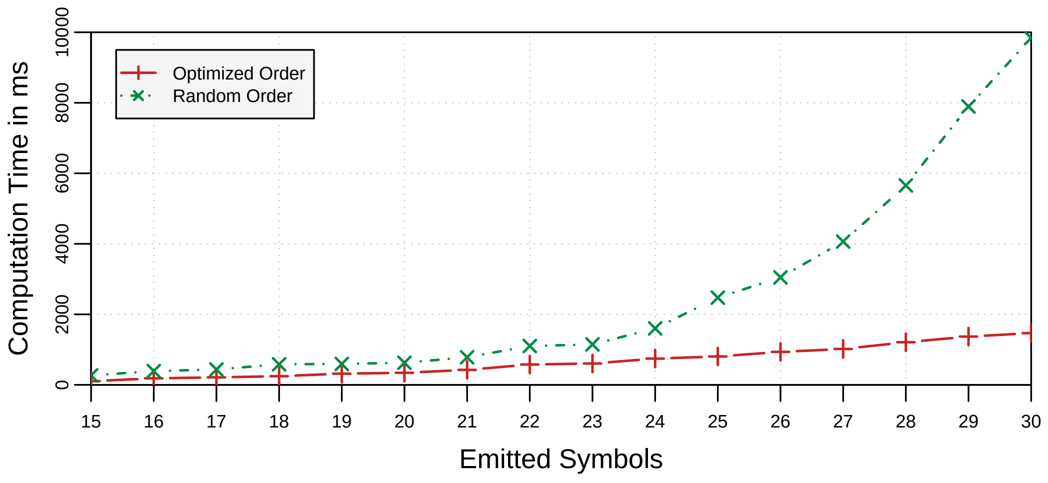


Figure 15.4: Effect of evaluation order

Figure 15.4 shows the averaged total runtime as a function of x. The matcher component that optimized the evaluation order of TCLs (Optimized Order) was clearly superior to the matcher component that evaluated TCLs in a random order (Random Order). For example, the processing of 10,000 sets each containing 25 symbol events per user-defined symbol needed 805 ms using an optimized evaluation order and 2,473 ms using a random evaluation order. In the case of input sets containing 30 symbol events per user-defined symbol, an optimized evaluation order already was more than six times faster than a random evaluation order. Therefore, the matcher component optimized the evaluation order of TCLs in all following experiments.

15.7.2 Maximum Throughput under Varying Parameters

In the following experiments, we pushed events as fast as possible into TPStream to determine its maximum throughput in different situations. We used synthetically generated event streams and queries in order to have full control of all parameters.

```
FROM         EventStream e
PARTITION BY e.id
DEFINE       S_1 AS temp(φ(e.x_1)) DO out = end(e).id
             S_2 AS temp(φ(e.x_2))
             ...
             S_n AS temp(φ(e.x_n))
PATTERN      TP_rand
WITHIN       w TIME UNITS
RETURN       out
```

Listing 15.15: Parametrized test query

Listing 15.15 shows our parametrized test query for an event stream `EventStream` whose schema consisted of a numeric attribute named `id` and n Boolean-valued attributes named $\mathtt{x}_1$, $\mathtt{x}_2$, ..., $\mathtt{x}_n$. The parameter n specified not only the total number of Boolean-valued attributes, but also the total number of user-defined symbols. In other words, n Boolean-valued event attributes led to test queries in which n different symbols $\mathtt{S}_1$, $\mathtt{S}_2$, ..., $\mathtt{S}_n$ were defined. Of course, also every generated input event had n Boolean values besides a value for the mandatory attribute `id` as payload. Every symbol $\mathtt{S}_i$ was defined via a temporal predicate (thus, every emitted symbol event could be either a point event or an interval event) which was dependent only on the event attribute $\mathtt{x}_i$. The ASEQ predicate φ used by every temporal predicate was simply the identity function $\varphi(b \in \mathbb{B}) := b$. Lastly, the temporal pattern TP_{rand} was a generated but valid TPStream pattern over all user-defined symbols and consisted of m randomly chosen TCs in total (as in the previous experiment).

We used an event generator that allowed us to control and vary all important characteristics. In particular, the generator created events evenly distributed across k partitions. There was exactly one event every 1,000 time units per partition. With probability of 50 %, an input event took part in the creation of a symbol event. The sizes of the time intervals of emitted symbol events were randomly chosen (uniform distribution) between zero time units (thus, a point event) and l time units so that the average size of time intervals was $l/2$. Because the total number of partitions k had no noticeable impact on the performance (using the default schema of partitioning), we present our results for $k = 1$ and different values of n, m, l and w in the following.

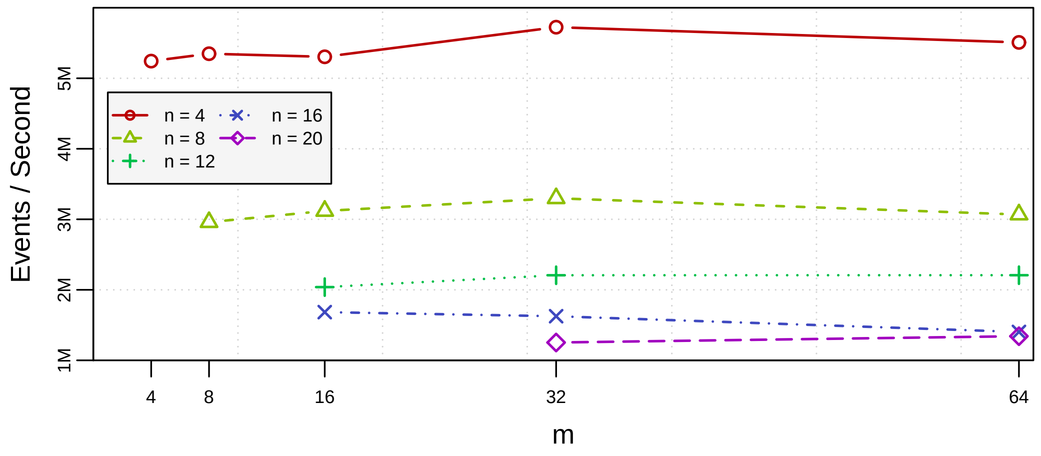


Figure 15.5: Effects of parameters n and m

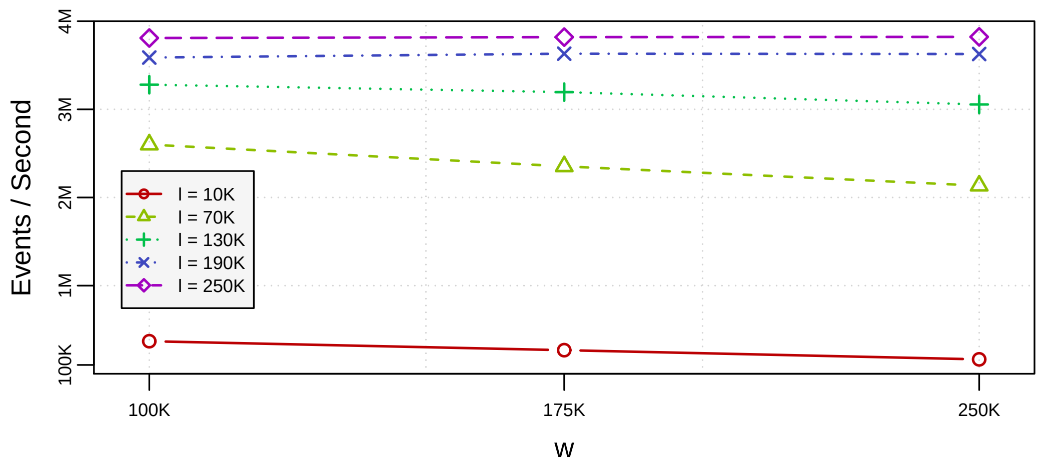


Figure 15.6: Effects of parameters w and l

At first, we tested different numbers of user-defined symbols n and of TCs m of the temporal pattern TP_{rand}. The size of the time window was fixed to $w = 175,000$ time units and the maximum length of time intervals of symbol events was fixed to $l = 130,000$ time units. Figure 15.5 shows the maximum throughput as a function of m for different settings of n. While the number of symbols n had a strong impact on the throughput, it remained relatively stable when the parameter m was changed. An increasing n led to an increasing number of predicates, an increasing number of sets of symbol events and an increasing number of TCLs. In contrast, an increasing m did not increase the numbers of predicates, symbol sets and TCLs. Additionally, more TCs had almost no effect because TCLs are translated into compact Boolean expressions.

In our last experiment, we tested different sizes of the time window w and of the time intervals of emitted symbol events l. The number of user-defined symbols was fixed to $n = 8$ and the number of TCs of the pattern TP_{rand} was fixed to $m = 16$. Figure 15.6 shows the results as a function of w for different settings of l. Our measurements clearly show that the performance dropped when l was decreased as well as when w was increased. The reason for this lies in the problem which TPStream must solve. Recall that the matching problem is similar to stream joins. For stream joins it is known that the performance decreases as the size of the state increases in general. In this experiment, the number of emitted symbol events within a fixed-size time window increased with a decreasing maximum size of time intervals of emitted symbol events l. Analogues, the larger the time intervals of emitted symbol events were the less emitted symbol events were within a fixed-size time window. The window size w affected the size of the state and, thus, the throughput in the same way. When w was increased the time window contained more emitted symbol events and when w was decreased the time window contained less emitted symbol events.

15.8 Related Work

Sequential pattern matching over event streams is discussed extensively in Section 15.2. None of the proposed operators is able to derive symbol events with time-interval semantics and to detect temporal patterns. Besides the general idea of pattern matching over event streams, TPStream is related to ISEQ as well as ZStream and mainly inspired by spatiotemporal pattern queries.

Spatiotemporal Pattern Queries. The spatial databases community identified a new and powerful class of so-called *spatiotemporal pattern queries* (STPQ) for detecting complex patterns in spatiotemporal data [Erw04]. Recently, a holistic approach to expressing and evaluating STPQ in trajectory databases was presented [SG11]. The temporal design of this approach served as a foundation for TPStream. In particular, TPStream adopts the concept of temporal predicates and the encoding schema for temporal constraints. In order to improve expressiveness towards sequential patterns, TPStream adds multiple important extensions (e.g., negation, alternatives, etc.). While TPStream is designed to detect patterns in streaming data in real-time, the approach presented in [SG11] is specifically designed for trajectory databases. That is, all data is available and allows for efficient indexing and query execution. Therefore, the approach presented in [SG11] is not applicable to streaming data.

ISEQ. To the best of our knowledge, ISEQ [Li11] is the only existing pattern matching operator that is based on time-interval semantics so far. ISEQ motivates the need for pattern matching on basis of time-interval semantics, adopts the flexible and expressive two-endpoint representation for specifying temporal patterns and has a strong emphasis on efficient query execution. However, ISEQ requires incoming events to have already time intervals. In addition, input events must be ordered by end timestamps (a quite unusual assumption especially in real-world applications) and ISEQ does not provide a high-level query language. The most significant difference to TPStream is that ISEQ does not allow users to define symbols having an interval-based semantics. Instead, ISEQ already expects those events with time intervals in its input instead of raw point events. The input events are created by some external middleware which remains completely unspecified. While TPStream supports the specification of symbols having an interval-based semantics (including correlations between different symbols) and temporal patterns in a single query definition, such queries cannot be expressed in ISEQ. In the case of ISEQ, the entire proposed operator is only for expressing and detecting temporal patterns. Since ISEQ is an efficient matcher component rather than a holistic operator such as TPStream, one could consider to use it as matcher component for TPStream. However, this is not easily possible due to its odd assumption that events must be ordered by end timestamps. Lastly, ISEQ depends on an additional punctuation mechanism [Tuc03] to avoid blocking. Such a mechanism is not provided by many EP systems (including JEPC) and not required in case of TPStream because of its non-blocking nature.

ZStream. Almost all operators for sequential pattern matching utilize an automaton for matching. ZStream [MM09] is different and proposes to parse a sequential pattern definition into an operator tree having the defined symbols as leaf nodes. On top of the leaf nodes, different operators are then applied. For instance, ZStream provides operators for expressing a sequence of symbols, a negation of a symbol and Kleene closures. During execution, input events stream from the corresponding leaf nodes upwards to the root. Symbol definitions are applied by labeling edges with the corresponding conditions (correlations are supported). The resulting trees are similar to traditional join trees and, thus, to the constraint graph evaluation of TPStream. Consequently, ZStream orders the operator nodes optimally according to a cost model. The proposed cost model is more complex, because it has to deal with user-defined conditions whose selectivities must be estimated. In the case of TPStream, every atomic condition is one of the 26 TCs and allows for a simpler cost estimation therefore.

Summary

This chapter presents TPStream, a pattern matching operator that is able to detect complex temporal patterns in event streams. On basis of proven concepts of state-of-the-art sequential pattern matching, a powerful query language is proposed for expressing complex temporal patterns over symbol events with time-interval semantics. While the input of TPStream are still instantaneous events that are produced by almost all event sources in practice, TPStream supports deriving higher knowledge in the form of symbol events with time-interval semantics. The events being derived are completely user-defined. The careful design of the query language makes it a real extension to existing query languages. It allows to express sequential as well as temporal restrictions within a single query. In this chapter, a first implementation of TPStream that is non-blocking and detects occurrences of patterns in real-time is also presented.

16

Automatic Matchmaking in Real-Time Data Stream Processing

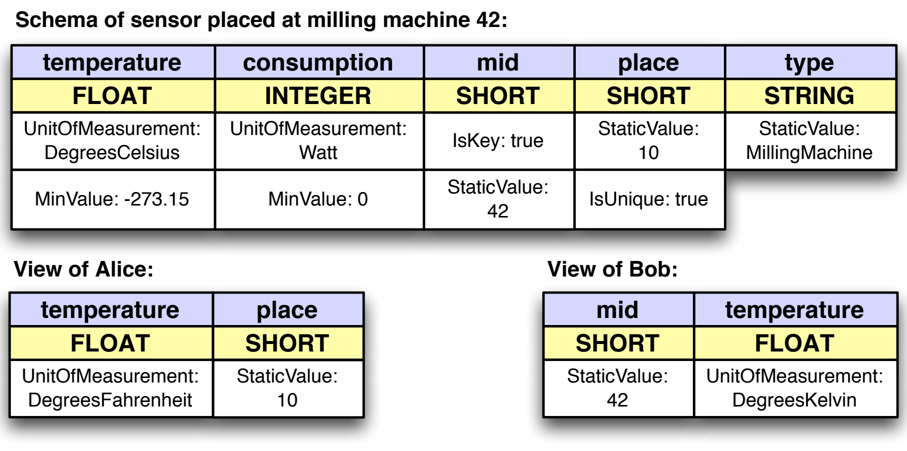
Outline

16.1 Introduction

One of the most important characteristics of a modern SPE is its ability to execute continuous queries over high volumes of streaming data with low latency [SÇZ05]. To achieve this goal, continuous queries and data sources are directly linked via fixed connections. Also continuous queries and data sinks as well as continuous queries with each other are hard-wired. This kind of coupling results in a very efficient flow of data, because no routing decisions must be made. Unfortunately, all connections must be specified purely manually. As a consequence, the creation of a stream processing application requires a lot of effort and the adaption of a stream processing application to a changed context is costly (e.g., if new data sinks are available or existing data sources have been updated). In short, the kind of connection management of today's SPEs is highly inflexible, but causes no performance overhead at runtime.

In contrast to hard-wired elements, there is the famous publish/subscribe pattern [Eug03] in which a central broker mediates between publishers (e.g., data sources) and subscribers (e.g., continuous queries). The result is a full decoupling of all elements in dimensions of time, space and synchronization. Publish/subscribe provides a high degree of flexibility, but leads to overhead at runtime because routing decisions must be made for each arriving data item. The imposed runtime overhead is significant and contradicts the requirements of real-time data stream processing therefore [Kal05]. Moreover, flexibility is quite limited in the sense that there must be an exact agreement on the semantics of the vocabulary (e.g., users must agree that a specific attribute *temperature* contains sensor readings measured in degrees Celsius).

What is needed is a hybrid approach that combines the good performance of fixed connections with the flexibility of the publish/subscribe pattern. In this chapter, we introduce the concept of automatic matchmaking that has been identified as a key enabler for advanced stream processing platforms [HS13]. Because of the real-time requirements, this concept still uses fixed connections. But in contrast to state-of-the-art stream processing technology, all connections are established automatically and adapted dynamically whenever a data producer/consumer is deployed or updated. This not only reduces the effort of building and maintaining complex applications, but also enables the use of stream processing technology in highly dynamic application domains such as the Internet of Things. In particular, the sets of data producers and consumers are allowed to change arbitrarily without affecting the runtime performance. Furthermore, our approach is able to automatically handle semantic differences among the connected data producers and consumers in a declarative manner.



Schema of sensor placed at milling machine 42:

temperature	consumption	mid	place	type
FLOAT	INTEGER	SHORT	SHORT	STRING
UnitOfMeasurement: DegreesCelsius	UnitOfMeasurement: Watt	IsKey: true	StaticValue: 10	StaticValue: MillingMachine
MinValue: -273.15	MinValue: 0	StaticValue: 42	IsUnique: true	

View of Alice:

temperature	place
FLOAT	SHORT
UnitOfMeasurement: DegreesFahrenheit	StaticValue: 10

View of Bob:

mid	temperature
SHORT	FLOAT
StaticValue: 42	UnitOfMeasurement: DegreesKelvin

Figure 16.1: Matchmaking example

The following example gives an impression of the concept of automatic matchmaking. Consider a production plant consisting of milling machines. Each machine has a sensor which provides a stream of measurements. The sensor placed at milling machine 42 (M42) periodically sends a data item following the schema shown at the top of Figure 16.1 (the first row gives the names of the attributes, the second their data types and all others contain metadata of the corresponding attribute). For example, *temperature* is a floating point number, measured in degrees Celsius and always greater than or equal to -273.15. Let Alice and Bob be two engineers being responsible for all machines of the plant. Both are constantly interested in the average temperature value over the last 50 measurements belonging to M42. They want to utilize a SPE for executing CQs that derive this information. But the desired shapes of data items differ from the sensor schema and from each other. Alice is most familiar with temperature values in degrees Fahrenheit and associates a specific machine with its place within the facility. In contrast, Bob prefers to use artificial identifications (*mid*), which are assigned to each machine, to select a specific one. Furthermore, he uses a software for post-processing that can analyze only temperature values in degrees Kelvin. In state-of-the-art SPEs, Alice and Bob are forced to connect their CQs with the data stream containing the sensor readings from all machines within the facility and to select as well as to manipulate data items by themselves (e.g., converting temperature values). In contrast, automatic matchmaking enables Alice and Bob to create CQs that consume data items following their individual views which are shown in the lower part of Figure 16.1. Then, the SPE establishes the connections automatically and takes care of selecting and modifying data items accordingly before delivery to the CQs.

Automatic matchmaking becomes more important or even an enabler in highly dynamic application domains such as the Internet of Things or security monitoring. In the latter case, a sophisticated system for monitoring IT infrastructures could deploy software-based sensors and CQs with respect to the degree of danger [HS13]. This means, in normal state only sensors and CQs of vital importance are active to save resources. But as soon as there is an indication of danger, the monitoring system deploys more and more sensors and CQs for getting deeper insights (e.g., observing all system calls of a suspect process). Because there are several situations of danger, the monitoring system decides on its own which sensors and CQs to deploy. Obviously, such a self-adaptive behavior of a monitoring system requires not only the automatic deployment of sensors and CQs, but also the automatic interconnection of them.

In automatic matchmaking, a data producer is automatically connected with all matching data consumers while slight differences in the semantics are fixed. We formally define the problem of matchmaking in this chapter. Because the naïve algorithm for solving the matchmaking problem is extremely costly, we propose various optimizations that are proven to perform well in practical settings.

16.2 Background and Motivation

In this section, we discuss the state-of-the-art connection management in stream processing applications and outline the benefits which stream processing technology achieves through its extension by automatic matchmaking.

16.2.1 State-of-the-Art Connection Management

Figure 1.1 on page 7 illustrates the overall structure of today's stream processing applications. In particular, it shows the three main components of every stream processing application (i.e., data sources, CQs, data sinks) as well as the fixed and user-defined connections between them. While data sources send their data items to CQs, CQs send their data items to other CQs and data sinks. Existing SPEs require users to define all connections by themselves. This not only results in a lot of effort during application development, but also prevents rapid application adaptation in highly dynamic applications where the context changes with high frequency. For human users it is not possible to rearrange the connections once every few seconds. Moreover, not all existing SPEs support changes at runtime. Instead, an application must be stopped, modified and started again. Ultimately, even little changes of a single element of an application can cause changes of many other elements (domino effect).

16.2.2 Benefits of Automatic Matchmaking

The matchmaker, which is a novel component in the architecture of stream processing that manages connections automatically, leads to the following important advantages.

16.2.2.1 Fine-Grained Connections

Typically, the term *data stream* refers to all active instances of the same type of data producer. For instance, all milling machines of the motivating example have the same type of sensor. It is best practice and forced by SPEs to merge all data items coming from data producers of the same type into a single data stream. This is totally analogous to the world of database systems where a relation *Students* contains all students instead of using a separate table for each individual student. Continuous queries that consume such data streams must use additional filter and partition operations in order to select and group the data items of a data stream. Furthermore, each input of a CQ can consume exactly one data stream and, thus, exactly one type of data.

The proposed matchmaker establishes connections at a lower abstraction level. In particular, connections are established between individual data producers and consumers. Therefore, it is possible for a CQ to obtain data items from an arbitrary subset of all data producers without the need to filter incoming data items. Moreover, if there are different types of data producers, CQs can receive data items from those on the same input. This enables CQs to take into account as much information as possible, comparable to multisensor data fusion [HL97]. Because the matchmaker establishes all connections automatically, the management of a large number of fine-grained connections instead of only few data streams becomes possible in the first place.

16.2.2.2 Self-Adaptivity

This chapter shows that the matchmaker can be implemented on top of (and of course, inside) every modern SPE in order to manage fine-grained connections fully automatically and dynamically. The latter means that it is possible to add, remove and update single data producers and consumers at any time during execution even if the underlying SPE does not support changes at runtime. Therefore, stream processing applications become self-adaptive so that SPEs can be used in dynamic application domains. For example, when a new sensor becomes available, the matchmaker connects this sensor to all relevant continuous queries of a running SPE immediately and automatically on-the-fly. Self-adaptivity allows for continuously varying sets of data producers and consumers such as they occur in the Internet of Things for instance.

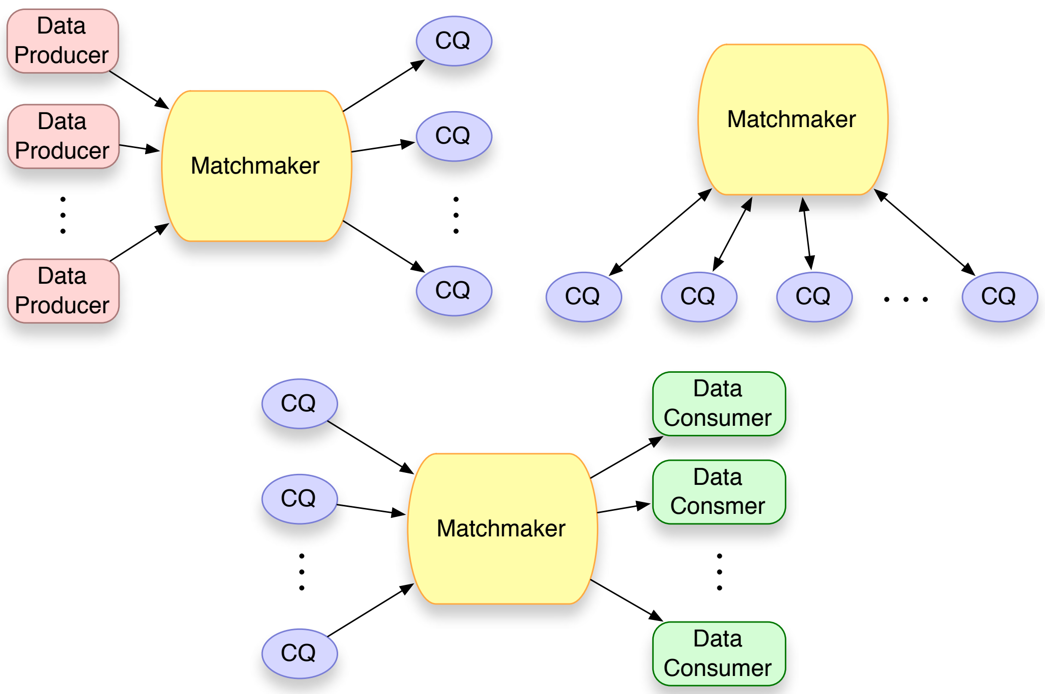


Figure 16.2: Producer-CQ, inter-CQ, and CQ-consumer independence

16.2.2.3 Independence

Physically connected to only the matchmaker, every single element of a stream processing application becomes independent of all others (i.e., a loose coupling is achieved). Figure 16.2 illustrates the different kinds of independence. The first kind arises from the decoupling of data producers and CQs (producer-CQ independence). This allows data producers and CQs to become available and to change arbitrarily. Additionally, this allows to execute a CQ even if there are no suitable data producers yet. When a suitable data producer becomes available, it is immediately connected to the CQ by the matchmaker. The same is true for two CQs, one acting as a data producer and the other acting as data consumer (inter-CQ independence). In fact, SPEs that do not support subqueries get this feature by using the matchmaker. Lastly, the CQ-consumer independence arises from the decoupling of CQs and data consumers.

16.3 Matchmaking Problem

This section introduces and formally specifies the problem of automatic matchmaking in real-time data stream processing. But first, the underlying data model is defined.

16.3.1 Data Model

Before we can formally specify the matchmaking problem, we must define the underlying data model. Its first component are properties that can be used to describe attributes more precisely such as in Figure 16.1. The main intention is the same as in, for example, the area of online dating where people describe themselves by properties (such as *(hobby, soccer)*, *(job, teacher)*) in order to create matches.

Definition 15 (Property)**.** *A* property *P is a pair (key, value) consisting of a key and a value. The set of all properties is denoted by $\mathbb{P}$. Two properties $P_1 \in \mathbb{P}$ and $P_2 \in \mathbb{P}$ are equal if and only if their components are equal: $(key_1, value_1) = (key_2, value_2) :\Leftrightarrow key_1 = key_2 \wedge value_1 = value_2$.*

Attributes are defined and used in the same way as in database management systems. In addition, an attribute may have any number of properties. Instead of allowing only predefined properties (such as in the case of constraints in database management systems), an attribute may have arbitrary properties being defined by users.

Definition 16 (Attribute)**.** *An* attribute *A is a triple $(name, type, \{P | P \in \mathbb{P}\})$ consisting of a name, a type and a set of properties. The set of all attributes is denoted by $\mathbb{A}$. Two attributes $A_1 \in \mathbb{A}$ and $A_2 \in \mathbb{A}$ are equal if and only if their names, types and sets of properties are equal: $(name_1, type_1, X) = (name_2, type_2, Y) :\Leftrightarrow name_1 = name_2 \wedge type_1 = type_2 \wedge X = Y$. An attribute $A_1 \in \mathbb{A}$* matches *an attribute $A_2 \in \mathbb{A}$ (denoted by $A_1 \models A_2$) if and only if their names and types are equal and each property of A_2 is also a property of A_1: $(name_1, type_1, X) \models (name_2, type_2, Y) :\Leftrightarrow name_1 = name_2 \wedge type_1 = type_2 \wedge Y \subseteq X$.*

In this chapter, we focus on relational schemas because of their simplicity and wide application in stream processing. Therefore, a schema is always a list of attributes. However, note that the presented concepts and algorithms can be easily carried over to any other kind of structured data model such as, for example, XML.

Definition 17 (Schema)**.** *A* schema *S is a tuple $(A_1, A_2, \ldots, A_n) \in \mathbb{A}^n$. Every attribute name must be unique within a schema: $\forall A_i, A_j \in S : i \neq j \Rightarrow A_i.name \neq A_j.name$. Two schemas $(A^1)^n \in \mathbb{A}^n$ and $(A^2)^m \in \mathbb{A}^m$ are equal if and only if $n = m \wedge \forall i \in [1:n] : A^1_i = A^2_i$. A schema $(A^1)^n \in \mathbb{A}^n$* matches *a schema $(A^2)^m \in \mathbb{A}^m$ (denoted by $(A^1)^n \models (A^2)^m$) if and only if $n = m \wedge \forall i \in [1:n] : A^1_i \models A^2_i$.*

Since schemas are lists of attributes as in database systems, data items are simply tuples that conform to their corresponding schemas.

Definition 18 (Data Item)**.** *A* data item *DI^S is a tuple conforming to a schema S.*

Now, all components of the data model are clearly defined and we can introduce transformations that allow to modify those components.

Definition 19 (Transformation)**.** *A* transformation τ *is a pair* (τ_S, τ_D) *consisting of two functions. The function* τ_S *maps a schema* $S \in \mathbb{A}^n$ *to a schema* $S' \in \mathbb{A}^m$ *and the function* τ_D *performs a corresponding mapping of data items. That is,* τ_D *maps every data item* DI^S *to a data item* $DI^{S'}$*. The set of all transformations is denoted by* $\mathbb{T}$*. Users can define an arbitrary set* $\mathbb{T}_U \subseteq \mathbb{T}$ *of active transformations.*

Because schemas are the domain and the codomain, transformations can be composed to complex mappings. Besides transformations, a mapping always includes a projection and a permutation as final steps. The projection is used to select a subset of attributes and the permutation is used to order the selected attributes.

Definition 20 (Mapping)**.** *A* mapping μ *is a composition* $\tau^\sigma \circ \tau^\pi \circ \tau^n \circ \ldots \circ \tau^2 \circ \tau^1$ *consisting of any number of transformations* $\forall i \in [1:n] : \tau^i \in \mathbb{T}$*, a projection denoted by* τ^π *and a permutation denoted by* τ^σ*.*

16.3.2 Problem Statement

The overall structure of stream processing applications shown in Figure 1.1 on page 7 can be simplified as follows. Obviously, the instances of every data source are data producers and those of every data sink are data consumers. Continuous queries cannot be classified so clearly, because they are both at the same time. A CQ acts as a data consumer at its input side and as a data producer at its output side. Because a CQ may have more than one input, each input of a CQ must be handled as an individual data consumer. Algorithm 27 on page 250 shows how CQs can be completely disassembled into a set of data consumers and exactly one data producer.

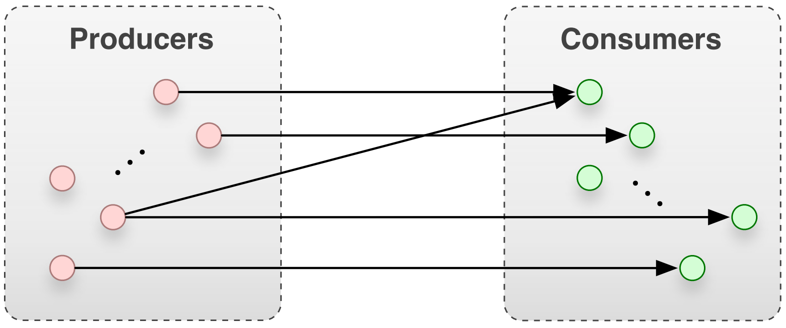


Figure 16.3: Basic problem

Figure 16.3 illustrates the basic problem. It shows one set of data producers and one set of data consumers. Any connections can be possible between both sets. In particular, every single data producer may (indirectly) match zero, one or multiple data consumers. Analogous, every single data consumer may (indirectly) match zero, one or multiple data producers. The task of the matchmaker is to compute and establish all possible connections between the data producers and the data consumers.

Definition 21 (Connection)**.** *A* connection *is a triple* $((X, S_X), (Y, S_Y), \mu)$ *consisting of a data producer* X *having the schema* S_X*, a data consumer* Y *having the schema* S_Y*, and a mapping* μ *that transforms* S_X *and all of its corresponding data items such that they match* S_Y *and all of its corresponding data items.*

On basis of the formally specified data model and the description of the basic problem, we can exactly define the general matchmaking problem.

Definition 22 (Matchmaking Problem)**.** *Let* X *be a data producer having the schema* S_X *and* Y *be a data consumer having the schema* S_Y*. A connection between* X *and* Y *can be established if and only if the following condition holds:*

$$\exists \tau^1, \ldots, \tau^k \in \mathbb{T}_U \cup \{\iota\} : (\ S'_X = (\tau^k \circ \ldots \circ \tau^1)(S_X)$$

$$\wedge\ \exists V \subseteq \{A' \mid A' \in S'_X\} : ($$

$$\exists S''_X = (A''_1, A''_2, \ldots, A''_m) : (\ (\forall i \in [1:m] : A''_i \in V)$$

$$\wedge\ m = |V|\ \wedge\ S''_X \models S_Y\ )\ )\ )$$

Altogether, the condition requires the existence of a suitable mapping for being able to connect X and Y. In particular, there must exist user-defined transformations which map the schema of the producer to a schema S'_X that includes the schema of the consumer. This means, a subset of all attributes of the schema S'_X can be ordered such that the resulting schema S''_X matches the schema of the consumer. Note that also the identity ι that does not modify a schema is always allowed as a transformation.

16.4 Solutions

We present different solutions of the matchmaking problem in the following. All solutions automatically compute connections in an offline manner. However, it is still important to focus on computing connections efficiently, because until a connection has been established all corresponding data items must be either buffered or discarded.

16.4.1 Basic Transformations

Instead of allowing users to specify arbitrarily complex transformations, we decided to support only three basic types. There are mainly two reasons that led to that decision. On the one hand, it is easier to define primitive transformations which perform simple mappings. On the other hand, users are forced to define multiple basic transformations to create a single complex one so that reusability is increased. However, users should still be able to express any kind of mapping. Therefore, the basic transformations are designed to not reduce the expressiveness. We describe only their schema transformations τ_S. Corresponding data items are treated exactly in the same way.

16.4.1.1 Conversion

The conversion τ^{conv} is a basic transformation that replaces a single attribute A of an input schema by another attribute A':

$$\tau_S^{conv}((\dots, A, \dots)) = (\dots, A', \dots)$$

The attribute being replaced A is called *required attribute* of τ_S^{conv} and the new attribute A' is called *image* of τ_S^{conv}.

16.4.1.2 Merge

The merge τ^{merge} is a basic transformation that creates exactly one additional attribute A' for a given schema on basis of any number of its attributes $\{A_i, A_j, \dots\}$ and adds the new attribute to the given schema:

$$\tau_S^{merge}((\dots, A_i, \dots, A_j, \dots)) = (\dots, A_i, \dots, A_j, \dots, A')$$

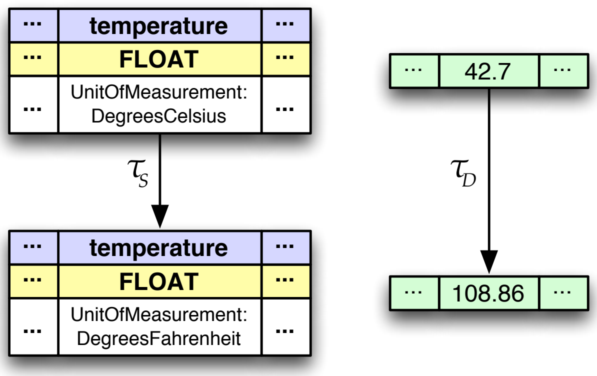

temperature
FLOAT
UnitOfMeasurement: DegreesCelsius
τ_S
temperature
FLOAT
UnitOfMeasurement: DegreesFahrenheit
42.7
τ_D
108.86
...

Figure 16.4: Conversion example

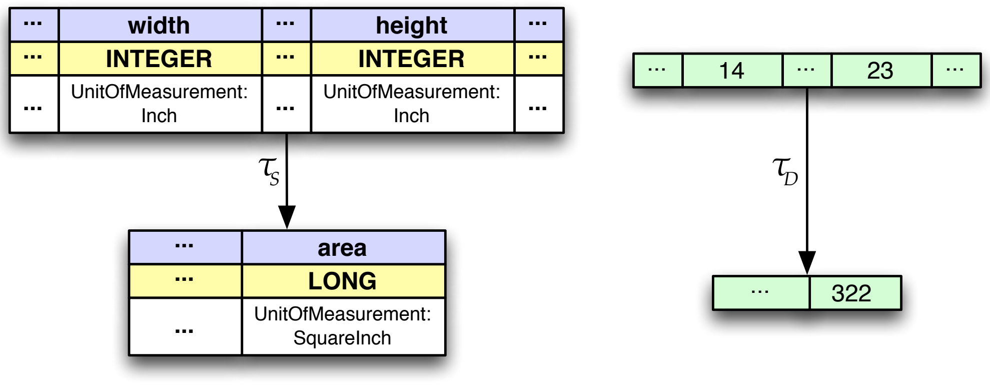


Figure 16.5: Merge example

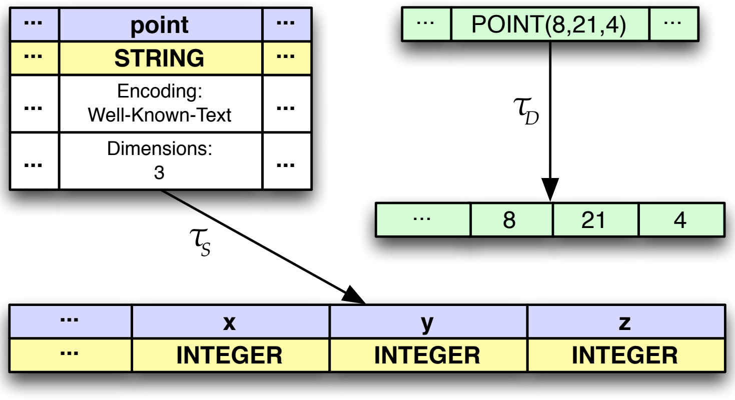


Figure 16.6: Split example

The attributes in the set $\{A_i, A_j, \ldots\}$ are called *required attributes* of τ_S^{merge} and the new attribute being added A' is called *image* of τ_S^{merge}.

16.4.1.3 Split

The split τ^{split} is a basic transformation that creates any number of additional attributes $\{A_n, A_{n+1}, \ldots, A_{n+m}\}$ for a given schema on basis of exactly one of its attributes A and adds the new attributes to the given schema:

$$\tau_S^{split}((\ldots, A, \ldots)) = (\ldots, A, \ldots, A_n, A_{n+1}, \ldots, A_{n+m})$$

The attribute A is called *required attribute* of τ_S^{split} and the set keeping the new attributes $\{A_n, A_{n+1}, \ldots, A_{n+m}\}$ is called *image* of τ_S^{split}.

If for at least one required attribute A of a basic transformation τ there is no attribute A' with $A' \models A$ in a schema S or if a schema S already contains at least one attribute of the image of τ, then the transformation τ behaves like the identity ι that leaves the schema unchanged (i.e., $\tau_S(S) = S$). We denote the set of all basic transformations by $\mathbb{T}_B$ and define the notion of applicability as follows:

Definition 23 (Applicability)**.** *A basic transformation $\tau \in \mathbb{T}_B$ is called* applicable to a schema $S \in \mathbb{S}$ *if and only if $\tau_S(S) \neq S$. Otherwise τ is* not applicable to S.

Figures 16.4, 16.5, and 16.6 give concrete examples of all types of basic transformations. The conversion shown in Figure 16.4 requires an attribute *temperature* of type floating point number with the property *(UnitOfMeasurement, DegreesCelsius)* and replaces it by an attribute with the same name and type but with the property *(UnitOfMeasurement, DegreesFahrenheit)*. For every corresponding data item, this transformation converts the attribute value from degrees Celsius to degrees Fahrenheit. The merge shown in Figure 16.5 requires two attributes *width* and *height* each of type integer number and with the property *(UnitOfMeasurement, Inch)*. Then, it adds a new attribute *area* keeping the size of the area. For a given data item, the additional attribute has the product of *width* and *height* as value. The split shown in Figure 16.6 requires an attribute *point* keeping point definitions in three-dimensional space as WKT strings. Then, it extends the schema by three new attributes *x*, *y* and *z* each containing the position in the associated dimension as an integer number.

Because $\mathbb{T}_B$ is a proper subset of $\mathbb{T}$, it must be clarified whether the expressiveness is reduced by allowing users to define only basic transformations.

Theorem 5. *For every transformation $\tau \in \mathbb{T}$ with $\tau_S(S) = S'$ there exists a composition of basic transformations $\tau^k \circ \ldots \circ \tau^1$ with $\forall i \in [1:k] : \tau^i \in \mathbb{T}_B$ and $(\tau^k \circ \ldots \circ \tau^1)(S) = S^+$ such that $\forall A' \in S' : \exists A \in S^+ : A' = A$.*

Proof. Let $\tau \in \mathbb{T}$ be an arbitrary transformation that maps a schema $S = (A_1, \ldots, A_n) \in \mathbb{A}^n$ to a schema $S' = (A'_1, \ldots, A'_m) \in \mathbb{A}^m$ and handles data items accordingly. Because every transformation consists of two functions, each attribute $A' \in S'$ is deterministically computable only on basis of attributes contained in S (the same is true also for data items). We must focus only on all attributes $A' \in S'$ that are new. An attribute $A' \in S'$ is called new if and only if $A' \notin S$. Because attribute names must be unique within a schema, we distinguish two cases:

i) $\nexists A \in S : A.name = A'.name$

Because A' has a name that is not used in S, A' is computable by a single merge that attaches A' to S.

ii) $\exists A \in S : A.name = A'.name$

In this case we have to take care of the requirement that every attribute has a unique name within a schema. If A' is computable only on basis of A, a single conversion is sufficient. Otherwise, we apply analogous to i) a merge in order to attach an attribute A'' with the same type and properties as A' but with a name that is not used in S and S' to S. At the end, a conversion gives A a name that is used neither in S nor in S' and another conversion renames A'' in A'.

The methods i) and ii) can be used to successively map the schema S to a schema S^+ that includes S'. The conversions used in ii) must be delayed until all new attributes that depend on an attribute which must be renamed are created. □

As shown by the proof, a schema S^+ that includes every attribute of S' can always be created via merges and conversions (splits are only supported for reasons of comfort). Unfortunately, the order of desired attributes might differ in S^+ and there might be undesired attributes of S. But in the context of matchmaking, every mapping includes a projection and a permutation after the composition of transformations.

Corollary 2. *Within a mapping μ, every transformation $\tau \in \mathbb{T}$ can be substituted by a composition of basic transformations.*

Proof. Let $\mu = \tau^\sigma \circ \tau^\pi \circ \ldots \circ \tau \circ \ldots$ be an arbitrary mapping including a transformation $\tau \in \mathbb{T}$. According to Theorem 5, the transformation τ can be replaced by a composition of basic transformations without affecting the subsequent transformations. By adjusting the permutation τ^σ and the projection τ^π of μ, all undesired attributes can be removed and the remaining attributes can be ordered correctly. □

16.4.2 Naïve Matchmaking

The computation of connections is triggered whenever a data producer or consumer has been added. When a data producer or consumer has been removed or updated, all associated connections are simply deleted. In case a data producer or consumer has been updated, all its connections are then recomputed entirely from scratch. Algorithm 22 shows the procedure that performs matchmaking for a new or updated data producer or consumer Z having the schema S_Z. If Z is a data producer, it must be checked for each data consumer whether a connection between it and Z can be established. Thus, the candidates for potential connections are all data consumers. And if Z is a data consumer, it must be checked for each data producer whether a connection between it and Z can be established. Therefore, all data producers are the candidates for potential connections. The checks are performed by a procedure f that solves the matchmaking problem for a given pair consisting of a data producer and a data consumer. Every solution being presented in the following assumes that users have already defined all basic transformations completely. For the rest of the chapter, let $\mathbb{T}_{BU} \subseteq \mathbb{T}_B$ be the set of all user-defined basic transformations.

While the procedure shown in Algorithm 22 is shared by all solutions being presented in the following sections, the implementation of the procedure f that solves the matchmaking problem differs from solution to solution. The first solution that we present follows a simple brute-force approach. In fact, it is the direct and naïve translation of the matchmaking problem (see Definition 22). Algorithm 23 describes that naïve solution of the matchmaking problem.

Algorithm 22: MATCHMAKING((Z, S_Z))

```
Input: New/updated producer/consumer: $(Z, S_Z)$
Data:  Set of all producers: $Producers$,
       Set of all consumers: $Consumers$
Set $Candidates \leftarrow$ NULL;
if $(Z, S_Z) \in Producers$ then
    $Candidates \leftarrow Consumers$;
else
    $Candidates \leftarrow Producers$;
foreach $(C, S_C) \in Candidates$ do
    if $(Z, S_Z) \in Producers$ then
        $f((Z, S_Z), (C, S_C))$;
    else
        $f((C, S_C), (Z, S_Z))$;
```

Algorithm 23: NAÏVEMATCHMAKING((X, S_X), (Y, S_Y))

```
Input: Producer: $(X, S_X)$, Consumer: $(Y, S_Y)$
foreach Schema $S \in$ TRANSFORM$(S_X, \mathbb{T}_{BU})$ do
    foreach Set $S' \in \mathcal{P}(S)$ do
        foreach Schema $S_\sigma \in$ PERMUTE$(S')$ do
            if $S_\sigma \models S_Y$ then
                New Connection: $((X, S_X), (Y, S_Y), \mu)$;
```

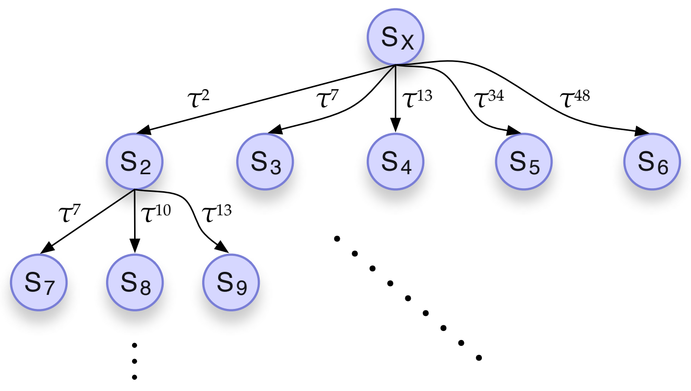


Figure 16.7: Transformation tree

The naïve implementation of f has two parameters. It must be called with a producer X having the schema S_X as first argument and a consumer Y having the schema S_Y as second argument. At first, a loop traverses all nodes in the tree that is created by $\text{TRANSFORM}(S_X, \mathbb{T}_{BU})$. This procedure creates a tree that has schemas as nodes and the schema S_X as root. Its edges are transformations taken from $\mathbb{T}_{BU}$. Figure 16.7 depicts such a *transformation tree*. The second level consists of all schemas that are created by applying every applicable transformation (see Definition 23) $\tau \in \mathbb{T}_{BU}$ to the root node. For each node at the second level, every applicable transformation $\tau \in \mathbb{T}_{BU}$ is applied to it. The other levels are created in the same way so that a transformation tree contains all schemas that can be created via a combination of basic transformations being applied to S_X. A transformation $\tau \in \mathbb{T}_{BU}$ is called *potentially applicable* to a producer schema if and only if the corresponding transformation tree has at least one edge labeled with τ. In practice, only a small fraction of $\mathbb{T}_{BU}$ is potentially applicable. For example, a conversion is potentially applicable only if the producer schema contains its required attribute or its required attribute is contained in the image of another potentially applicable transformation (PAT). Additionally, every transformation is allowed to occur at most once on every path. It is possible that users have specified a set of conversions that can destroy the image of a PAT completely (the simplest case are two conversions being inverse to each other). Then, this PAT would be applicable multiple times on a single path, but not result in new nodes. Instead, a cycle that leads to infinite paths would have been created. Such undesired cycles are prevented by forbidding repeated applications of a transformation on a path.

A second loop iterates over all subsets for each node of the transformation tree. The algorithm $\mathcal{P}(S)$ creates the powerset of S (i.e., S is interpreted as a set of attributes). Then, the algorithm $\text{PERMUTE}(S)$ creates every permutation of the attributes in a set

S and a third loop iterates over all those permutations. Finally, every schema created by the three loops is checked whether it matches the consumer schema. If so, a connection can be created but not established yet. Note that the mapping μ of this connection has been already computed by the three loops. The path from the root of the transformation tree to the node currently being visited gives the composition of basic transformations. Furthermore, the projection has been computed by the loop in the middle and the innermost loop defines the order of attributes. At the end, we select the connection with the shortest mapping if multiple connections have been created. In case of the naïve and optimized solutions, there may be multiple connections with this property. However, we do not study this problem in detail, because the final solution (fast matchmaking) always gives a clear connection at this point.

Before we can analyze the time complexity of Algorithm 23, we must name the most important parameters first. Let C be the total number of candidates, N_P be the average number of attributes of the producer schemas and N_C be the average number of attributes of the consumer schemas. Furthermore, let M be the average size of the transformation trees being measured in total number of nodes and let the parameter N_T represent the average number of attributes of a node of a transformation tree. Note that it is always the case that $N_T \geq N_P$, because there is no type of basic transformation which reduces a schema but two types (merge and split) which extend a schema. On basis of these parameters, we can determine the time complexity of naïve matchmaking (Algorithm 22 in conjunction with Algorithm 23) as follows:

$$\mathcal{O}(C * M * 2^{N_T} * N_T! * N_C)$$

For each candidate, Algorithm 23 must be executed exactly once. Its execution comprises traversing the entire corresponding transformation tree, iterating over the powerset for each node of the transformation tree and iterating over all permutations for each element of the powerset. Finally, the check whether a created schema matches the candidate schema requires at least traversing all attributes in the consumer schema. The size M of the transformation tree is influenced by the total number of PATs, denoted by x. In the worst case, every PAT is applicable to the root node and never becomes non-applicable because of the application of other PATs. Then, the corresponding transformation tree has as many levels as PATs and a size M according to:

$$M(x) = \overbrace{\sum_{0 \leq i \leq x}}^{\text{Level}} \overbrace{\frac{x!}{(x-i)!}}^{\text{\#Nodes/Level}}$$

16.4.3 Optimized Matchmaking

The analysis of the time complexity of naïve matchmaking clearly shows that the brute-force computation of connections is way too expensive for practical uses. In particular, the iterations over the powerset and over all permutations of each element of the powerset are already for mid-sized schemas extremely costly (see also our experiments in Section 16.6.2). Recall that these iterations are used to check whether a certain node in the transformation tree matches the schema of the consumer and, if so, to provide the projection and the permutation of the corresponding mapping. This problem is very similar to the well-known problem of schema matching. There is a more efficient solution to the schema matching problem in the form of nested loops that compare each attribute of one schema to every attribute of the other [RB01].

Algorithm 24: OPTIMIZEDMATCHMAKING((X, S_X), (Y, S_Y))

```
Input: Producer: $(X, S_X)$, Consumer: $(Y, S_Y)$
1 foreach Schema $S \in$ TRANSFORM($S_X, \mathbb{T}_{BU}$) do
2     foreach Attribute $A \in S$ do
3         foreach Attribute $A' \in S_Y$ do
4             if $A \models A'$ then
5                 remember positions of $A'$ and $A$;
6     if $\forall A' \in S_Y : \exists A \in S : A \models A'$ then
7         New Connection: $((X, S_X), (Y, S_Y), \mu)$;
```

Algorithm 24 shows an improved implementation of f. For each schema in the transformation tree, the second loop iterates over all its attributes and the third loop iterates over all attributes of the consumer schema. As a result, every pair of attributes is checked exactly once. If such a pair of attributes matches, the algorithm remembers their positions within their corresponding schemas. This way the two innermost loops determine the projection and the permutation of a mapping in quadratic time. After the two innermost loops have finished, it must be checked whether there has been found a matching attribute in the schema currently being iterated in the transformation tree for every attribute of the consumer schema. If so, a schema match has been found and a new connection is created. The modification of the two innermost loops results in an update of the third and fourth term of the time complexity:

$$\mathcal{O}(C * M * N_T * N_C * N_C)$$

16.4.4 Fast Matchmaking

Our final solution to the matchmaking problem differs in two aspects from the previous ones. First, the nested loops that check for matches between every node of a transformation tree and a consumer schema are replaced by a more efficient approach. Second, subtrees of a transformation tree are pruned if they do not contain matching schemas or if they contain only schemas that have already been checked.

16.4.4.1 Zig-Zag Matching

The nested loops approach used in optimized matchmaking is taken over from the solution to the schema matching problem in which nested loops compare each attribute of one schema to every attribute of another schema. However, schema matching is approximate using some similarity measure for scoring every pair of attributes. At the end, those pairs that have the highest score above some threshold are selected. In contrast, the match relation of matchmaking (i.e., $\models$) requires equality of attribute names and types (nevertheless, features such as matching synonyms/hypernyms and type conversions are still possible through the concept of transformations). Therefore, schema matching in the context of matchmaking is a join problem for that better solutions than nested loops are available. The strategy of fast matchmaking is to sort two schemas being checked and to perform a modified zig-zag join [GUW08].

Figure 16.8 shows this kind of schema matching called *zig-zag matching*. Let the upper sorted schema be a (transformed) producer and let the lower sorted schema be a consumer. For the sake of simplicity, all attributes have the same type and no properties. Zig-zag matching starts with a pointer to the first attribute of the consumer and another pointer to the first attribute of the producer, which moves forward until an attribute that matches the current attribute of the other pointer is found. Then, the

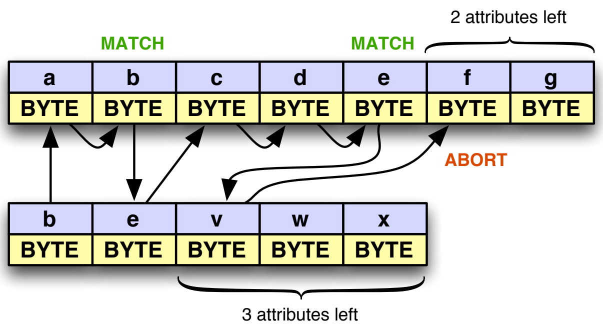


Figure 16.8: Zig-zag matching

Type	*Schema transformation*	*Superfluous pruning*
Split	$\tau^1 : x \to \{a\}$	
Conversion	$\tau^2 : y \to w$	
Merge	$\tau^3 : \{z\} \to b$	
Split	$\tau^4 : y \to \{c\}$	✗
Conversion	$\tau^5 : z \to e$	✗
Merge	$\tau^6 : \{x, w\} \to d$	✗

Table 16.1: PATs for a schema having the attributes x, y, and z

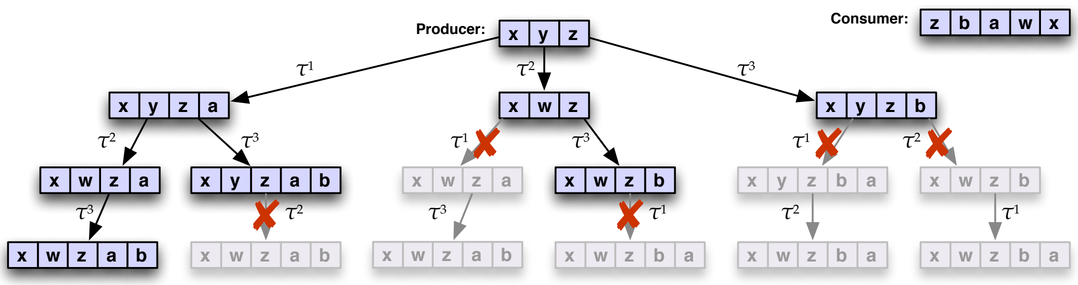


Figure 16.9: Pruning of redundant nodes in a transformation tree

pointer to the consumer is moved to the next attribute and the pointer to the producer starts from its current position in order to find the next match. Zig-zag matching aborts early in the following situations. If for an attribute of the consumer no match can be found, the pointer to the producer reaches the end and zig-zag matching stops, because the pointers always move forward. Moreover, if the number of attributes ahead the pointer to the consumer is greater than the number of attributes ahead the pointer to the producer, then there is definitively no schema match possible and zig-zag matching aborts. In Figure 16.8, the matching is aborted because the consumer has still three attributes left but the producer has only two attributes left.

16.4.4.2 Pruning of Superfluous Transformations

Up to now, all PATs are used to create a transformation tree. The PATs are obtained from the set of all active transformations $\mathbb{T}_{BU}$ by filtering it on basis of constraints coming from the producer schema. In real-world applications, only few transformations will remain as a PAT in general. Figure 16.9 shows a producer schema having the attributes x, y and z. Assume that the six transformations shown in Table 16.1 are the only PATs in $\mathbb{T}_{BU}$. All listed transformations except τ^6 are PATs, because they are directly applicable to the producer schema. The merge τ^6 is a PAT, because the producer schema contains its required attribute x and its other required attribute w is contained in the image of τ^2, which has already been identified as a PAT. All previous solutions use these six transformations to create the transformation tree. But if we consider the consumer schema (shown at the upper right corner of Figure 16.9), the set of PATs can be further filtered on basis of constraints coming from the consumer schema.

Definition 24 (Superfluity). *A PAT $\tau \in \mathbb{T}_{BU}$ is* superfluous *if and only if its image contains no attributes of the consumer schema and there is no other PAT which is not superfluous and requires an attribute of the image of τ.*

Our first pruning strategy, called *superfluous pruning*, comprises removing all superfluous transformations from the set of all PATs before creating a transformation tree. In the example shown in Table 16.1 and Figure 16.9, half of the PATs (namely τ^4, τ^5 and τ^6) are superfluous and not used to create the transformation tree therefore.

16.4.4.3 Pruning of Redundant Nodes

Figure 16.9 shows a transformation tree created on basis of all PATs that are not superfluous. The tree has many nodes that are equal or differ only in the order of attributes from each other. This kind of redundancy can be specified more precisely:

Definition 25 (Redundancy). *Two nodes S_1 and S_2 in a transformation tree are* redundant *to each other (denoted by $S_1 \simeq S_2$) if and only if all attributes of S_1 are contained in S_2 and vice versa: $S_1 \simeq S_2 :\Leftrightarrow A \in S_1 \Rightarrow A \in S_2 \wedge A \in S_2 \Rightarrow A \in S_1$*

Obviously, redundancy is not affected by the order of attributes (set semantics). Because all basic transformations specify neither the positions of required attributes nor their order and because of the subsequent permutation included in every mapping, the relation $\simeq$ behaves as if it is the equality relation. This means that a redundant node is a duplicate and can be safely removed therefore. Before we develop a strategy for the pruning of redundant nodes, we introduce the notion of independence.

Definition 26 (Independence). *Two transformations τ^1 and τ^2 are* independent *of each other if and only if $\forall S \in \mathbb{S} : (\tau^2 \circ \tau^1)(S) \simeq (\tau^1 \circ \tau^2)(S)$.*

According to Definition 26, two adjacent transformations being independent of each other can be safely swapped within a composition of transformations.

Theorem 6. *A subtree of a transformation tree U which is started by some transformation $\tau \in \mathbb{T}_{BU}$ contains only redundant nodes if on the path leading to U there exists a node S that already has a subtree which is started by τ and all transformations on the path between U and S are independent of τ.*

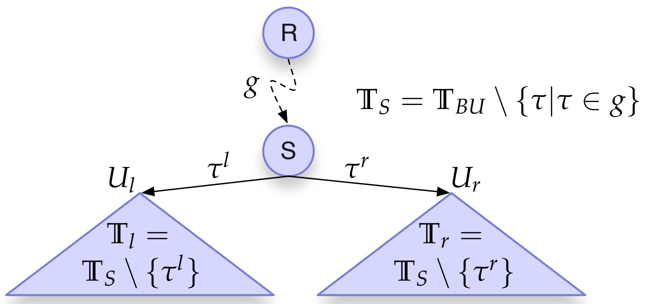

R
g
S
$\mathbb{T}_S = \mathbb{T}_{BU} \setminus \{\tau | \tau \in g\}$
U_l
τ^l
τ^r
U_r
$\mathbb{T}_l = \mathbb{T}_S \setminus \{\tau^l\}$
$\mathbb{T}_r = \mathbb{T}_S \setminus \{\tau^r\}$

Figure 16.10: Verification of redundant pruning

Proof. Consider the situation illustrated in Figure 16.10. Let $\mathbb{T}_{BU}$ be an arbitrary set of user-defined basic transformations and R be a producer schema (thus, the root of a transformation tree). Furthermore, let S be an arbitrary node in the corresponding transformation tree and g be a composition of transformations $\tau \in \mathbb{T}_{BU}$ with $g(R) = S$. The composition g is allowed to be empty such that $S = R$. Because a transformation can occur on every path at most once (see Section 16.4.2), every transformation in g is excluded from paths beginning at S. The set $\mathbb{T}_S$ contains all transformations that are candidates for S and its subtrees. Let two different transformations $\tau^l \in \mathbb{T}_S$ and $\tau^r \in \mathbb{T}_S$ be applicable to S. Let U_l be the left subtree of S with root $\tau^l(S)$ and candidate transformations $\mathbb{T}_l$. Let U_r be the right subtree of S with root $\tau^r(S)$ and candidate transformations $\mathbb{T}_r$. Let h be a path $(\tau^l \circ \ldots \circ \tau^r)(S)$ starting at S and ending somewhere in U_r. It must be shown that every node following $h(S)$ is redundant to a node in U_l if all transformations in h are independent of τ^l.

Let S' be a node that arises at the end of a path $(h' \circ h)(S)$ with h as prefix (i.e., τ^l has been applied before) and an arbitrary composition h'. We focus only on transformations in h that we assumed to be independent of τ^l. Via repeated swaps, τ^l can be moved from the end of h to its beginning while the order of all other transformations in h remains unchanged. The resulting path from S to S' must be already contained in U_l so that S' is redundant to a node in U_l. □

Theorem 6 allows the detection and the pruning, called *redundant pruning*, of subtrees that only contain redundant nodes. We integrated redundant pruning into a post-order based creation of transformation trees. Before creating a subtree via some transformation τ^l, it is checked whether there is a node S on the path created so far that has a subtree starting with τ^l. If so, all transformations on the path between the current node and S are checked whether they are independent of τ^l. In the case that all those transformations are independent of τ^l, we stop and do not create the subtree of τ^l. The information whether two transformations are independent of each other is obtained by querying a two-dimensional triangular matrix which stores for each pair of transformations this information. This matrix is updated in linear time whenever a user adds or removes a basic transformation. We identify two basic transformations τ^x and τ^y as being independent of each other, if no attribute of the image of τ^x matches a required attribute of τ^y and no attribute of the image of τ^y matches a required attribute of τ^x. According to this definition, the transformations τ^1, τ^2 and τ^3 in Table 16.1 are identified as being independent of each other. All subtrees that can be safely pruned are cut off by a cross and grayed out in Figure 16.9.

Algorithm 25: FASTMATCHMAKING((X, S_X), (Y, S_Y))

```
  Input: Producer: $(X, S_X)$, Consumer: $(Y, S_Y)$
1 Schema $S'_Y \leftarrow$ SORT($S_Y$);
2 foreach Schema $S \in$ TRANSFORMFAST($S_X, \mathbb{T}_{BU}$) do
3     if ZIGZAGMATCHING(SORT($S$), $S'_Y$) $=$ TRUE then
4         New Connection: $((X, S_X), (Y, S_Y), \mu)$;
```

Fast matchmaking shown in Algorithm 25 combines all optimization techniques. The nested loops of optimized matchmaking have been replaced by zig-zag matching and the procedure that creates and traverses transformation trees has been replaced by TRANSFORMFAST that performs superfluous pruning and redundant pruning in addition. The time complexity is updated due to zig-zag matching as follows:

$$\mathcal{O}(C * [N_C * \log N_C + M * ((N_T * \log N_T) + (N_C + N_T))])$$

16.5 Implementation

We implemented the matchmaker including all three solutions being presented in this chapter as a JEPC extension that can be used purely on top. Therefore, our matchmaker extension can be combined with all EP providers that are supported by JEPC. Since our implementation uses only the core API of JEPC which is an abstraction of the APIs of SPEs, it is sufficiently flexible to be easily ported to other SPEs.

Table 16.2 shows the core API of our implementation. It provides basic functionality for stream processing, but the notion of a data stream has been removed completely. For example, instead of registering a data stream *MillingMachines* into which

Method	*Description*
AddTransformation(τ)	Adds a user-defined basic transformation τ
RemoveTransformation(τ)	Removes the user-defined basic transformation τ
RegisterProducer(S)	Registers a new producer having schema S
UnregisterProducer(ID)	Unregisters the producer with identification ID
UpdateProducer(ID, S)	Replaces the schema of the producer ID by S
RegisterConsumer(S)	Registers a new consumer having schema S
UnregisterConsumer(ID)	Unregisters the consumer with identification ID
UpdateConsumer(ID, S)	Replaces the schema of the consumer ID by S
CreateQuery(CQ)	Creates and executes a continuous query CQ
DestroyQuery(ID)	Destroys the query with identification ID
UpdateQuery(ID, CQ)	Replaces the query with identification ID by CQ
PushDataItem(ID, DI)	Pushes the data item DI for the producer ID

Table 16.2: API of matchmaker

Algorithm 26: PushDataItem(ID, DI)

```
Input:  Producer: $ID$, Data Item: $DI$
1 Data Item $DI' \leftarrow$ Add static values to $DI$;
2 foreach Connection $(ID, consumer, \mu) \in$ LookUp($ID$) do
3     Data Item $DI'' \leftarrow \mu$(Clone($DI'$));
4     Forward $DI''$ to $consumer$;
```

all milling machine sensors push their data items, every milling machine sensor registers individually by sending its schema to the matchmaker. Then, a unique identification is returned and matchmaking (see Algorithm 22) is performed for the new data producer. Exactly the same is done for every data consumer. Registered data producers and consumers use their identifications to update the schema and to unregister. In case of data producers there are two more aspects. First, a data producer also uses its identification to push new data items. Second, the matchmaker remembers all attributes being declared as static and the corresponding static values. These values that will not change for a period of time must not be send anymore. Consequently, this reduces the amount of data to send and improves the overall performance. For example, the sensor of milling machine 42 in Figure 16.1 declared the attributes *mid* and *place* as static (note that *StaticValue* is a special property in our implementation). Thus, the sensor announces that these values will not change for all future data items until it updates its schema and that it will send data items containing only values for

Algorithm 27: CREATEQUERY(CQ)

```
Input: Continuous Query: $CQ$
1 foreach Operator MATCH($S$)$\in$ $CQ$ do
2     REGISTERCONSUMER($S$);
3 REGISTERPRODUCER(GETOUTPUTSCHEMA($CQ$));
4 EXECUTE($CQ$);
```

temperature and *consumption*. All other attributes are declared static and attached automatically within the matchmaker before forwarding to data consumers. Algorithm 26 shows the corresponding steps in detail. The matchmaker attaches all static attributes and performs a lookup for obtaining all established connections. For each established connection, the matchmaker clones the enriched data item, applies the corresponding mapping and forwards the result to the specified data consumer.

The loss of the notion of a data stream has an important implication for the specification of the inputs of continuous queries and data sinks. Instead of selecting one or more data streams, one or more input schemas must be specified. For this purpose, we adopted the MATCH operator known from schema matching [RB01] (note that this operator has a completely different meaning in schema matching). Listing 16.3 gives examples of this operator in JEPC-QL. Its only parameter is a schema definition. In JEPC-QL, a schema is defined as a list of attribute definitions separated by commas. An attribute definition consist of three parts. At first, the attribute name must be defined. Then and separated by a colon, the attribute type must be defined. Lastly, a set of key-value pairs (i.e., the attribute properties) can be defined. Algorithm 27 shows how the matchmaker handles continuous queries that include the MATCH operator. We focus only on continuous queries that produce exactly one type of output. But it is straightforward to handle also continuous queries with multiple outputs. Appendix B presents a complete example of our implemented matchmaker and is intended to give a better impression of this novel kind of stream processing.

16.5.1 Parallelization

Matchmaking can be parallelized. The obvious and best opportunity is to partition the set of candidates in Algorithm 22 and to process the partitions of candidates in parallel. Since there are no dependencies between two candidates, a maximum degree of parallelization can be achieved. However, our current implementation runs sequentially and parallelization is on our agenda for future work.

16.5.2 Query Updates

Note that the matchmaker API shown in Table 16.2 provides a method for updating a running continuous query. This method has no counterpart in the traditional API of stream processing and is not supported by today's SPEs. Therefore, we had to implement our approach to updating CQs on-the-fly (see Chapter 18) also for the matchmaker. As a pleasant side effect, the matchmaker has an interesting positive impact on our proposed approach. In the traditional stream processing paradigm that deals with fixed and user-defined data streams, there is exactly one but naturally given restriction in our proposed approach. This restriction is that the output schema of an updated query definition must be equal to the output schema of the original query definition. However, this restriction ceases to exist when the matchmaker is used. In particular, this means that the definition of a running continuous query can then be exchanged for any other definition. The connections to the inputs as well as from the output are recomputed by the matchmaker. Of course, if the output schema changes it may be that some connections disappear and/or completely new connections are established.

16.5.3 Properties

Throughout this chapter, we always assume that users explicitly specify the properties of attributes. The semantic enrichment of attributes is crucial for the quality of the automatically established connections and should be done carefully therefore. But there is also the possibility to derive or generate properties (semi-)automatically. For example, assume sensors that fly in balloons over a whole country (e.g., for measuring meteorological data such as temperature, humidity and air pressure). Every sensor provides its geographic coordinates (e.g., via GPS) as a property. Additional higher-level properties can be derived from the coordinates. For instance, a property can be added that gives the name of the city where the sensor is closest to. Then, a query can specify a city of interest in its input schemas. The matchmaker takes care that only sensors close to that city are connected with the query. Note that properties such as the position of a (slowly) moving object are allowed to change with low frequency. In fact, the method to update producers is mainly provided for updating the properties from time to time (schema evolutions are also supported but expected to be quite rare). Even if experts say that state-of-the-art metadata generation techniques cannot completely replace humans [GSC06], we think that it is interesting future work to adopt current and future techniques for (semi-)automatic metadata generation from active research done in the area of the Semantic Web [Dil03, HSC02, Kir04].

16.6 Evaluation

To evaluate the presented solutions to the matchmaking problem and our implementation, we conducted several experiments on a machine with an Intel i7-2600 3.4 GHz processor and 8 GiB main memory running Oracle JRE 1.7.0_13 and Esper 4.9.0.

16.6.1 Impact on Performance

Concerning the design and implementation of the matchmaker, we primarily cared about not introducing overhead that decreases the performance at runtime. Therefore, we decided to compute fine-grained connections in an offline manner. At runtime, all established connections are simply looked up for a new data item. Although a positive impact on the runtime performance was not a goal of our work, we observed an interesting effect that is caused by the fine-grained connections the matchmaker establishes. For a producer that does not match the desired properties of a consumer there is no connection established even if their schemas without properties match. This implements an efficient filter mechanism in front of all running queries.

In the first experiment, we executed three different query workloads each consisting of exactly two continuous queries that implemented the motivating example described in Section 16.1. Every set of queries computed the average temperature over the last 50 measurements of milling machine 42 and converted the temperature values for Alice and Bob differently. Two different workloads consisted of queries with different definitions so that the resulting processing of data items differed. The first two workloads contained query definitions conforming to the traditional stream processing paradigm and the third workload made use of the matchmaker.

```
@Name("Alice")
select (35+(avg(temperature) * 9f/5f)) as temperature, place
from MillingMachines.std:groupwin(place).win:length(50)
group by place
having place = 10;

@Name("Bob")
select mid,(avg(temperature) + 273.15f) as temperature
from MillingMachines.std:groupwin(mid).win:length(50)
group by mid
having mid = 42;
```

Listing 16.1: Test queries of “Esper 1” in Esper EPL

```
create window AliceSub.win:length(50) select temperature, place
from MillingMachines;
insert into AliceSub
select cast(35f + (temperature * 9f / 5f), float) as temperature, place
from MillingMachines(place = 10);
@Name("Alice")
select avg(temperature) as temperature, place
from AliceSub;

create window BobSub.win:length(50) select mid, temperature
from MillingMachines;
insert into BobSub
select mid, cast(temperature + 273.15f, float) as temperature
from MillingMachines(mid = 0);
@Name("Bob")
select mid, avg(temperature) as temperature
from BobSub;
```

Listing 16.2: Test queries of “Esper 2” in Esper EPL

```
(SELECT   AVG(temperature) AS temperature, place
 FROM     MATCH(temperature:FLOAT {(UnitOfMeasurement,DegreesFahrenheit)},
                place:SHORT {(StaticValue,10)}
          ) WINDOW(COUNT 50 EVENTS)
 GROUP BY place
) AS Alice;

(SELECT   mid, AVG(temperature) AS temperature
 FROM     MATCH(mid:SHORT {(StaticValue,42)},
                temperature:FLOAT {(UnitOfMeasurement,DegreesKelvin)}
          ) WINDOW(COUNT 50 EVENTS)
 GROUP BY mid
) AS Bob;
```

Listing 16.3: Test queries of “Esper M” in JEPC-QL

The queries of the set *Esper 1* (see Listing 16.1) are expressed in Esper EPL and define the selection of M42 and the conversion of temperature values after the computation of the average temperature value. In contrast, selection and conversion are defined before the aggregation by the queries of the set *Esper 2* (see Listing 16.2). The queries of the set *Esper M* (see Listing 16.3) are expressed in JEPC-QL and define only the aggregation, because they use the `MATCH` operator. Selection and conversion are part of the individual views of Alice and Bob (see Figure 16.1) which are the arguments of the `MATCH` operator and specify how incoming data items must look like.

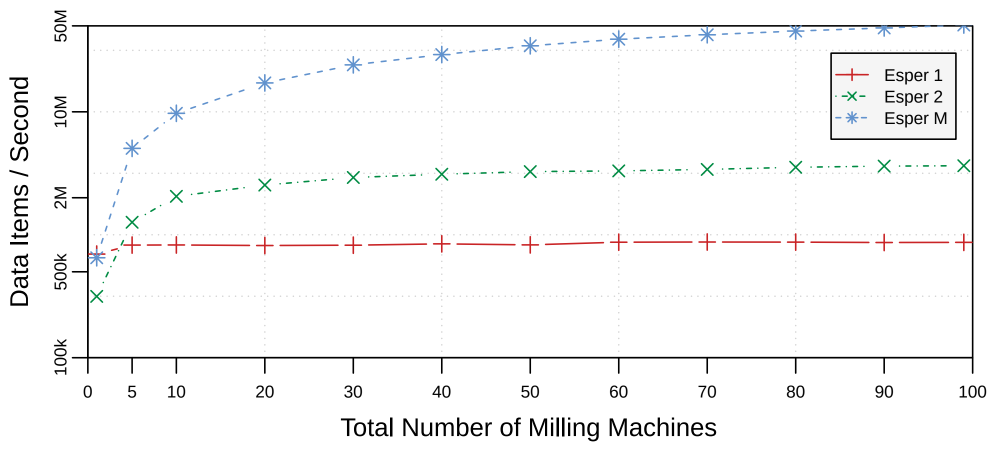


Figure 16.11: Impact on runtime performance

We executed each set of queries separately and measured the maximum possible event throughput. *Esper 1* as well as *Esper 2* were executed directly in the SPE Esper and *Esper M* was executed in Esper using the matchmaker on top. The data items came evenly distributed from a size-varying set of different milling machine sensors that always included the sensor of M42. Figure 16.11 shows our results. When the number of individual milling machine sensors was increased, also the number of irrelevant data items increased. Consequently, the filter condition contained in each single query became more selective. The queries of *Esper 1* could not take advantage of a lower selectivity, because the filtering was performed only after the average temperature value had been computed for every active milling machine sensor including all irrelevant ones. But the queries of *Esper 2* and the queries of *Esper M* benefitted from a lower selectivity. In case of both workloads, all sensor readings not belonging to M42 were discarded before aggregation. Thus, the average temperature value was computed only for M42 and not for irrelevant milling machines. Because the processing steps of *Esper 2* and *Esper M* were identical to each other, we can state that preventing irrelevant data items from entering continuous queries or a SPE (when there is no fine-grained connection, the matchmaker does not forward data items) is more efficient than directly filtering data items after they entered the continuous queries or the SPE. From the important perspective of runtime performance, it is beneficial to compute fine-grained connections via the matchmaker. Additional costs only occur for matchmaking in an offline manner. The superiority of *Esper M* to *Esper 2* clearly shows that the filter mechanism being inherent in the matchmaker is very efficient and performs better than the filter mechanism of continuous queries executed in a SPE.

16.6.2 Evaluation of Matching Techniques

Among all three solutions to the matchmaking problem, there are three different techniques for checking whether two schemas match and, if so, computing the projection and the permutation of the corresponding mapping (brute force matching, nested loops matching and zig-zag matching). To determine the costs only of matching, the setup of the next experiments was as follows. There were absolutely no transformations defined so that every transformation tree consisted only of the root node keeping the original producer schema. One run of the experiment comprised exactly one invocation of matchmaking. Therefore, we registered exactly one data producer and exactly one data consumer at the matchmaker. This led to exactly one call of f, because there was only one producer-consumer pair that must have been checked. Lastly, the schemas of the data producer and consumer had the same number x of attributes in total (thus, the same size) and were defined to not match.

For different total numbers of attributes x, Figure 16.12 shows the results using the brute force matching of naïve matchmaking. The results of this experiment clearly confirm what had already been indicated by the study of the time complexity of naïve matchmaking: brute force matching becomes costly very quickly. For example, a producer schema consisting of 12 attributes in total (which is not a large number in many real-world applications) already required 3.4 minutes for iterating over the powerset of the schema and over all permutations for each element of the powerset. Every further increase of x led to totally unacceptable runtimes for matching. Recall that matching is only one part of matchmaking. In general, the transformation tree consists of many nodes and matching must be performed for each single node.

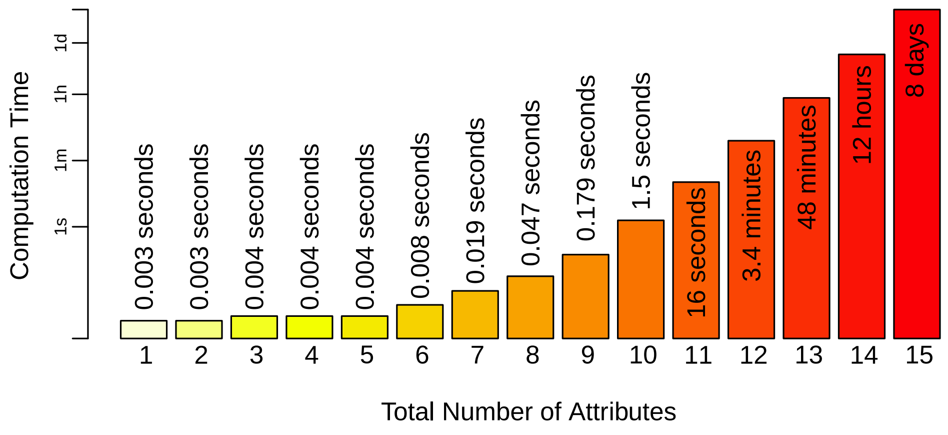


Figure 16.12: Brute force matching

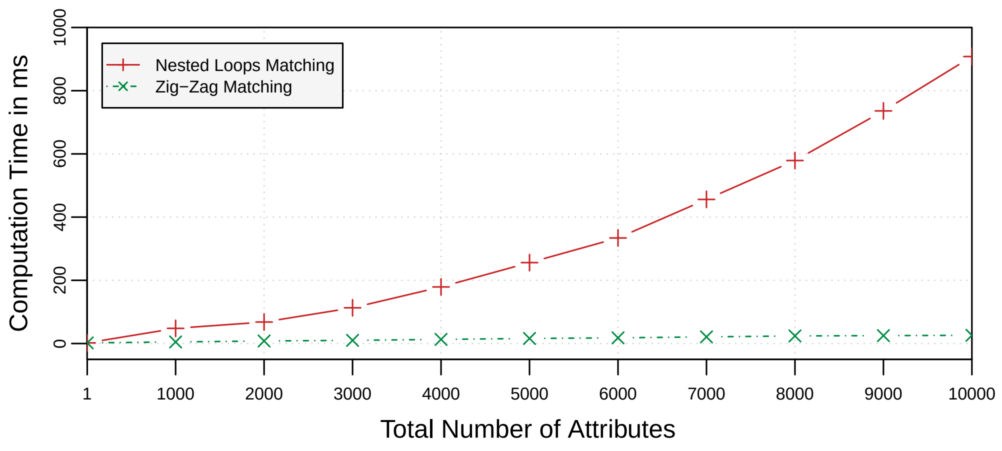


Figure 16.13: Nested loops matching and zig-zag matching

Figure 16.13 shows the results for nested loops matching which is used in optimized matchmaking and for zig-zag matching which is used in fast matchmaking. Both matching techniques completed the computation within milliseconds, even for extremely large schemas. In case of small schemas, the results show that both of the techniques were sufficiently fast. However, zig-zag matching clearly outperformed nested loops matching and scaled significantly better with the total number of attributes.

16.6.3 Evaluation of Pruning Techniques

The following experiments were performed to investigate the pruning techniques and the influence of external parameters. Our experimental setup consisted of a single consumer and a single producer, again. The schemas of both had ten attributes and zig-zag matching was always used for matching. We generated multiple sets of active transformations. Each generated set had a different size, consisted of one percent potentially applicable transformations with respect to the producer schema, and contained no superfluous transformations with respect to the consumer schema.

Figure 16.14 shows the average matchmaking time for different total numbers of active transformations. In particular, we compared the procedure TRANSFORM which creates and traverses entire transformation trees (Without Redundant Pruning) to the improved procedure TRANSFORMFAST which prunes redundant nodes in transformation trees (With Redundant Pruning). As the figure shows, the effect of redundant pruning was significant. For instance, matchmaking with redundant pruning required less than 50 milliseconds for 1,000 active transformations, whereas matchmaking without redundant pruning required more than 15 seconds.

In the next experiments, we kept the basic setup but performed always fast matchmaking. With respect to the consumer schema, many transformations are superfluous in practice. Therefore, we relaxed the artificial condition that there are no superfluous transformations and made a certain fraction of active transformations superfluous. Figure 16.15 presents the results for different numbers of active transformations and fractions of 10 %, 33 % and 67 % superfluous transformations. The figure shows that already 33 % superfluous transformations led to a clearly better performance. A set of transformations containing 67 % superfluous transformations resulted in significantly lower computation time. In this configuration, fast matchmaking with 2,500 active transformations finished within 13 milliseconds on average.

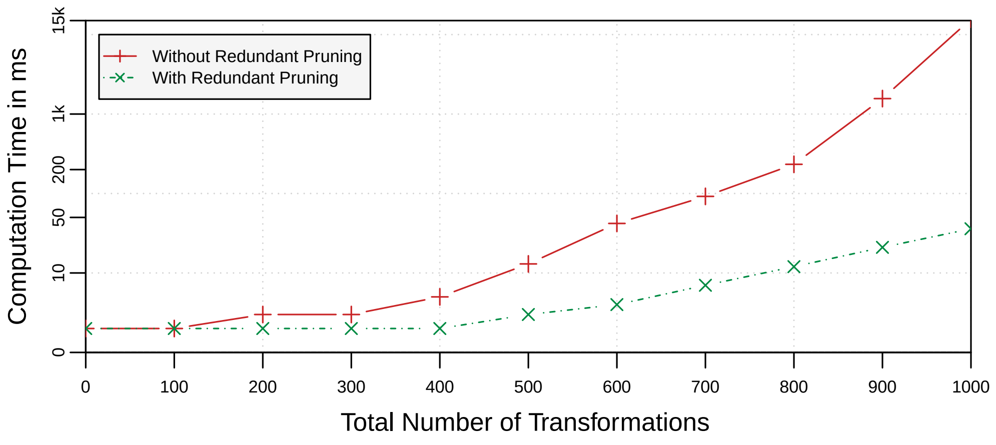


Figure 16.14: Effect of redundant pruning

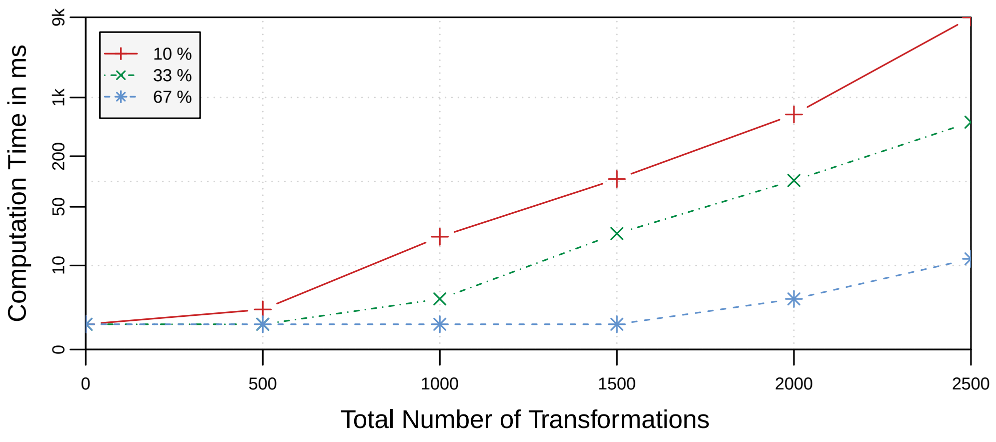


Figure 16.15: Effect of superfluous pruning

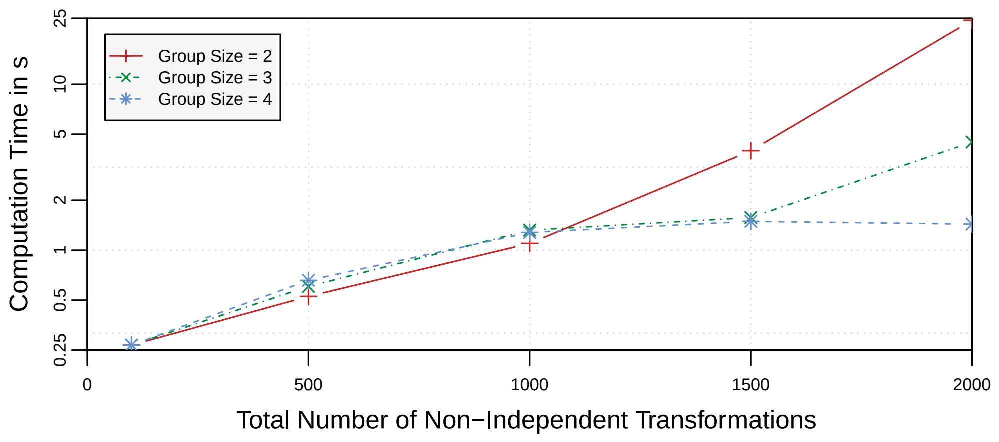


Figure 16.16: Effect of non-independence

Non-independence of transformations is another influencing parameter we had to examine, because non-independence potentially prevents the application of redundant pruning (see Theorem 6). We used the same basic setup as in the last experiments. But the set of active transformations was fixed this time and always consisted of 2,500 active transformations. In addition, a fixed fraction of 33 % of all active transformations was superfluous with respect to the consumer schema. Among all 2,500 active transformations, we varied the amount of non-independent transformations. Figure 16.16 shows the results for different configurations of this experiment. On the x-axis, the total number of non-independent transformations is given. For example, a value of 1,000 means that 1,000 of the 2,500 active transformations were non-independent. The non-independent active transformations were generated as follows. We partitioned all active transformations that should become non-independent in groups of 2, 3 or 4 active transformations each. Within each group, the active transformations were made non-independent in the form of a chain. For instance, in a group $\{\tau^x, \tau^y, \tau^z\}$ of size 3, the transformation τ^y was dependent on τ^x and τ^z was dependent on τ^y such that τ^y could be only applied after τ^x and τ^z only after τ^y. Note that in real-world applications non-independent transformations tend to be part of longer chains, because users are forced to express one complex transformation in the form of multiple basic transformations (recall that only conversion, split and merge are supported). Figure 16.16 shows that an increasing amount of non-independent active transformations reduced the effect of redundant pruning. However, the degree of reduction became less influencing the longer the chains of non-independent transformations were.

16.7 Related Work

The proposed component for automatic matchmaking touches on multiple different research areas. In this section, we give an overview of all relevant related work and discuss the similarities as well as the differences to automatic matchmaking.

Schema Matching. Schema matching has gained much attention in the past and is related to automatic matchmaking, because automatic matchmaking includes schema matching as an integral part. Most proposed techniques only support the process of manual matching (e.g., via graphical user interfaces or electronic advisors). Automatic schema matching requires additional context data (corresponding instances of schemas for example) besides schema definitions [RB01]. However, automatic schema matching is quite complex and, more importantly, approximate using some distance measure. In contrast, the schema matching part of automatic matchmaking is simple and exact. In fact, it is a problem of join processing rather than a problem of automatic schema matching. For the handling of differences between schemas, automatic matchmaking includes the powerful concept of transformations.

Schema Mapping. Schema mapping deals with the problem of creating a query that maps between two disparate schemas [MHH00]. It is related to automatic matchmaking in two different ways. First, a suitable mapping must be found in schema mapping at first. This is similar to the concept of transformations used by the matchmaker. Our presented matchmaker uses a set of user-defined transformations and tries to combine them accordingly so that two originally disparate schemas match. Second, a found mapping must be applied in schema mapping as final step. In schema mapping, a found mapping is applied in the form of a query. Of course, this approach would also be applicable in case of the matchmaker. But we decided to apply the mappings within the matchmaker component rather than by CQs within the used SPE, because our experimental evaluation showed that mappings can be applied by the matchmaker as efficiently as or even more efficiently as by CQs within a SPE.

Implicit Type Conversions. The concept of transformations is inspired by programming languages that convert types automatically. In almost all programming languages, different numeric built-in types can be combined in computations without explicitly converting them to a common supertype. Moreover, some modern programming languages such as Scala allow to define such implicit type conversions also for non-primitive built-in types (e.g., lists) and arbitrary user-defined types [OSV11].

Semantic Web. In the Semantic Web [BHL01], also designated as the next generation of the Web aka Web 3.0 [Hen09], machines are able to interpret information produced by humans (or other machines) and to make use of it in any kind of processing. One of the most important technologies of the Semantic Web is the resource description framework (RDF). Via RDF, it is possible to represent data in such a way that machines can easily process it. The specific representation is quite simple but powerful. Every atomic information is stored as a triple consisting of an entity (or resource), a key (or predicate) and a value. Because a collection of RDF triples forms a graph, it is easy to determine whether and how an object is related to another object. Automatic matchmaking implements the main idea of RDF in the form of properties. But instead of using a central storage that keeps all context information as triples, properties are stored as key-value pairs directly at the entities in automatic matchmaking.

Publish/subscribe. Publish/subscribe is a paradigm in which distributed elements exchange messages anonymously via a centralized mediator without specifying or knowing the receivers of a message. The result is the entire decoupling of elements in time, space and synchronization [Eug03]. This kind of decoupling and flexibility is necessary in large-scale systems that are distributed across a complex computing network. While the decoupling of data producers and data consumers in space via a central broker is also implemented by automatic matchmaking, the decoupling in time and synchronization would conflict the requirements of real-time processing. In contrast to publish/subscribe where distributed elements exchange arbitrary and potentially heterogenous messages, stream processing requires the transport of masses of data items, which are homogenous per producer, in real-time. Type-based publish/subscribe and topic-based publish/subscribe are quite the same as state-of-the-art stream processing that models data streams to represent a certain type or topic and, thus, are also too inflexible. In contrast, content-based publish/subscribe is attractive due to the resulting flexibility. However, computing all receivers for every single incoming data item individually imposes high runtime overhead that violates the stream processing requirements and is unacceptable therefore [Kal05]. Content-based publish/subscribe also contradicts the fact that stream processing deals with relatively stable connections that change not or only with a low frequency (compared to the frequency of data items). Therefore, automatic matchmaking solves the matchmaking problem per producer-consumer pair only once instead for every singly data item. As long as producer and consumer of a connection do not change, the connection between them keeps established and leads to a very efficient flow of data.

Multisensor Data Fusion. Multisensor data fusion combines data from multiple (and maybe heterogenous) sensors to achieve improved accuracies and more specific inferences than could be achieved by the use of a single sensor alone [HL97]. Because of the fine-grained connections that are allowed to have individual mappings, the matchmaker is also able to merge data coming from multiple (and maybe heterogenous) sensors. However, multisensor data fusion goes further. It also handles uncertainty and conflicts that can arise by combining different sensors [BN09].

TelegraphCQ. Adaptive routing of data items was addressed intensively by TelegraphCQ [Cha03]. So-called *Eddies* are used to dynamically route data items between query operators. This allows for re-optimzation of query execution in an online manner and on per data item basis. However, Eddies are only able to change the execution order at runtime and do not implement a single feature of the proposed matchmaker. Moreover and as already stated two paragraphs before, decision making on per data item basis introduces tremendously overhead.

Complex Event Processing. The concept of matchmaking was previously proposed by Luckham as an important component of complex event processing (CEP) [Luc02]. To the best of our knowledge, this matchmaker has been never implemented.

Summary

This chapter introduces automatic matchmaking that enables stream processing applications to manage all connections automatically and dynamically at runtime in order to achieve self-adaptivity. In strong contrast to content-based publish/subscribe, data consumers select entire streams of data items and not single data items in the concept of automatic matchmaking. We present algorithms and several optimizations for the efficient computation of fine-grained connections between the data producers and data consumers of a stream processing application. Connections are established between not only data producers and data consumers that perfectly match, but also data producers and data consumers that slightly differ in semantics. Therefore, we propose the powerful concept of transformations that resolve semantical differences. Once established, a connection transfers the data items from a producer to a consumer without any significant overhead and respects the important real-time requirements of stream processing therefore. Experiments prove that our proposed approach is able to compute connections in reasonable time.

17

Event Stores

Outline

17.1 Introduction

One of the main characteristics of data stream processing is that data items are kept in main memory and only as long as they are needed for processing. From a technical point of view, this is due to the large (potentially unbounded) volume of data streams as well as the high performance requirements in terms of low latency and high throughput. And from the view of an application, older data items (e.g., log entries) are very often of no or only little interest. For all these reasons, a data item is discarded when it is no longer needed for the evaluation of continuous queries. However, there are also several good reasons for archiving data items:

Data Provenance. Stream processing systems are popular and the first choice in time-critical applications because of their low latency. They are able to detect opportunities and risks in near real-time. However, quick response to such situations requires not only fast detection, but also fast reaction. Usually, the outputs of continuous queries are consumed by software or hardware components that automatically trigger reactions which can have a huge impact. For instance, algorithmic trading applications could buy stocks for billions of dollars or monitoring applications could control public infrastructure autonomously (e.g., shutting down a power plant). Because of the scope of such decisions, it is important to be able to exactly reproduce why an action was triggered by a system (particularly in case of suboptimal decisions). For this reason it must be known which data entered the system and which were the (intermediate) results of computations [BKT00, GD07, SPG05].

High Availability. Some systems must be available at all times (e.g., monitoring systems in safety-critical domains such as nuclear power plants). For achieving high availability, there are two common approaches. In one approach, a second system which is an exact copy of the original system is running in parallel. If one of the two systems fails, the other can take over immediately. The other approach also requires a second system but is more economical, because the second system is not running until the original system fails. Only then it is started and replaces the original system. If the systems have a state such as an EP provider, the rescue system first must reach a consistent state. Until then both systems are not available. In many cases the start-up time of the rescue system can be dramatically reduced, if the most recent history is repeated. This requires to always keep (the most recent) historical data. In Chapter 18, we use historical data to quickly start and update continuous queries.

Historical Analysis. In many application domains, it is necessary to analyze historical data. Some important use cases are forecasting and anomaly detection. For example, an anomaly can be specified (and, thus, detected) only if the normal state is known. Very often, the normal state is defined as the state that was valid in the past [Den87]. Therefore, historical data is needed to compute models describing the normal state. Historical analysis and stream processing are complementary. Derived models can be used to define continuous queries that detect anomalies.

IT Compliance. Enterprises are forced by law to store some data for a certain period of time. For example, data related to financial transactions must be kept in almost all countries. It is needed to check the annual financial statements. If important data is not available, many countries such as Germany punish enterprises and responsible managers personally with harsh penalties according to the law.

JEPC can optionally record external event streams as well as output event streams of EPAs. If recording is enabled, JEPC stores events via an interface that abstracts from so-called *event stores*. This interface specifies simple methods for recording and querying event streams and can be implemented via adapters by a wide range of existing storage systems. In the next section, we describe the event store interface at first. The subsequent sections then present three different implementations of it.

17.2 Event Store Interface

The event store interface is designed to be simple and general in order to enable as many as possible different types of storage systems to (efficiently) implement it. Almost all methods are for the purpose of either recording or querying events. The core of JEPC itself utilizes only a few methods for the recording of event streams. But applications and extensions on top of JEPC have access to a connected event store. Therefore, more methods than needed by JEPC core are specified and provided.

The most important methods for recording event streams are shown in Table 17.1. Before an event stream can be recorded, it must be registered via REGISTERSTREAM. The unique name of the event stream must be given as argument. Also its schema is expected as argument. This is because some types of storage systems (e.g., relational database systems) require this information. Registered event streams can be unregistered (UNREGISTERSTREAM), removed (REMOVESTREAM), suspended (SUSPENDSTREAM) and resumed (RESUMESTREAM). The meanings of these methods are as follows. While UNREGISTERSTREAM makes an event stream unavailable until it

Method	*Description*
REGISTERSTREAM(n,s)	Registers the event stream named n having schema s
UNREGISTERSTREAM(n)	Unregisters the event stream named n
REMOVESTREAM(n)	Removes the event stream named n
SUSPENDSTREAM(n)	Suspends the recording of the event stream named n
RESUMESTREAM(n)	Resumes the recording of the event stream named n
GETSTREAMS(n)	Gets a set containing the names of all registered event streams
GETSCHEMA(n)	Gets the schema of the event stream named n
GETEVENTCOUNT(n)	Gets the total number of stored events for the event stream n
GETLATESTTIMESTAMP(n)	Gets the latest timestamp of the stored event stream named n
PUSHEVENT(n,p,t_s,t_e)	Inserts a new event (p, t_s ,t_e) into the event stream named n

Table 17.1: Methods for recording event streams

Method	*Description*
GETHISTORY(n)	Gets an iterator over the event stream named n
GETHISTORYINVERSE(n)	Gets an inverse-directed iterator over the event stream named n beginning with the latest event
GETHISTORYMOSTRECENT(n,t)	Gets an iterator over the event stream named n beginning at time instant t
GETHISTORYTIMEPOINT(n,t)	Gets an iterator over all events being valid at time instant t of the event stream named n
GETHISTORYTIMERANGE(n,t_1,t_2)	Gets an iterator over all events being valid in $[t_1 : t_2]$ of the event stream named n
QUERYATTRIBUTES(n,t_1,t_2,m)	Gets an iterator over all events being valid in $[t_1 : t_2]$ and fulfilling m of the event stream named n
QUERYSQL(q)	Gets an iterator over the result set of the query q

Table 17.2: Methods for reloading and querying recorded event streams

is registered again, REMOVESTREAM also deletes all recorded events of it. Suspending an event stream means that recording of new events is paused. The event stream itself is still available (e.g., it can be queried). Recording of suspended event streams can be continued by calling RESUMESTREAM. Furthermore, the interface specifies methods to obtain all registered event streams, the schema of a registered event stream, the total number of recorded events and the latest timestamp of a registered event stream. Lastly, new events are inserted via PUSHEVENT. This method expects an event with time-interval semantics as argument. Internal events of JEPC have already the correct format. For external events, we set t_e to $t_s + 1$.

Table 17.2 shows the most important methods for reloading and querying historical data. The method GETHISTORY simply returns an iterator over all recorded events of a stream. Events are iterated in temporal order starting with the oldest recorded event. The method GETHISTORYINVERSE also gets an iterator over all recorded events, but this time events are iterated in inverse direction. If not stated otherwise, all following methods return iterators that give events in temporal order. The method GETHISTORYMOSTRECENT again returns an iterator but it starts at a user-defined point in time. Thus, this method can be used to get all events being valid between some past point in time and now. The method GETHISTORYTIMEPOINT gets an iterator over all events that are valid at a user-defined point in time. GETHISTORYTIMERANGE returns an iterator over all events being valid within a user-defined time interval. An extension of this method is QUERYATTRIBUTES that requires selected events to fulfill user-defined conditions in addition. Therefore, users can define for each attribute an arbitrary set of desired values. Then, only events having the desired values for all restricted attributes are returned. Lastly, QUERYSQL is provided for transmitting arbitrary textual queries. This method has been added, because many storage systems have a declarative query language. Analogues to JDBC, QUERYSQL simply forwards a textual query definition directly to the underlying storage system and returns the result set.

17.3 JDBC Event Store

We implemented one adapter that maps the event store API to JDBC. Since the payloads of events are already tuples, each event stream can be stored in its own database table with the name and schema of the event stream. All query methods of the event store API can be expressed in the form of simple standard SQL queries.

Via this adapter, every standard database system can be used as event store. This gives all the advantages of standard database systems. In particular, recorded event streams can be accessed by almost every data analytics tool via powerful SQL queries. But there is also a remarkable downside. Standard database systems have an extremely poor write performance. This is because they are simply not designed for recording high-speed data streams [SÇ05, Sto07a]. In addition, there is a tremendous overhead imposed by components and features not directly related to data recording and processing. Most of the overhead is caused by logging, locking, latching and buffer management [Har08]. Because of the poor write performance, standard database systems can only serve as event stores in JEPC applications that deal with low-speed event streams. Otherwise, the event store becomes a bottleneck.

17.4 In-Memory Event Store

The in-memory event store is a tailor-made implementation of the event store interface. It keeps each recorded event stream in a standard Java list and, thus, in main memory. Obviously, recorded events are not written to external storage and only small amounts of events can be kept, because the size of main memory is quite limited. Eventually a maximum amount of events has been recorded so that the oldest must be discarded to record new events. For all these reasons, the in-memory event store cannot be used for all purposes that are listed in the introduction. However, it still became a part of the JEPC core and is the default event store. One reason is its very good write performance since no external storage is involved. It does not become a bottleneck in JEPC applications therefore. Also its read performance is good. The in-memory event store is provided mainly for supporting fast query starts and updates (see Chapter 18) in JEPC applications that do not need events to be archived on external storage.

17.5 B^+-Tree Event Store

So far, there are two different event stores. Unfortunately, none of them fulfills the two most important requirements (use of external storage and excellent write performance) at the same time. In this section, we present a high-performance implementation of the event store interface that writes events to external storage with near-optimal write performance and achieves good read and query performance.

The B^+-tree event store is highly optimized for fast recording of event streams and fast reloading of their most recent histories. It was implemented using the XXL library [BDS00, Ber01] that is a toolbox for building custom-tailored databases. Each event stream is stored in its own B^+-tree [GUW08] on the temporal dimension. Because events arrive ordered by time per stream, the B^+-tree event store is designed as an append-only database [Ter92] and exploits the order to insert new events efficiently. For each B^+-tree, a certain number of full disk pages are buffered in main memory. Once a buffer is full, a bulk insertion [BSW97, DeW94, Gra06] moves the buffered disk pages from main memory into the B^+-tree. This leads to the following advantages: random I/Os are avoided because always multiple disk pages of the same index are written in a sequence, the bulk insertion has linear I/O complexity and the space is utilized optimally because leaf nodes can be filled up completely. The existing B^+-tree implementation of the XXL library was slightly extended for the B^+-tree event store. At the leaf level, we are using a double-linked list of disk pages (for time travels in

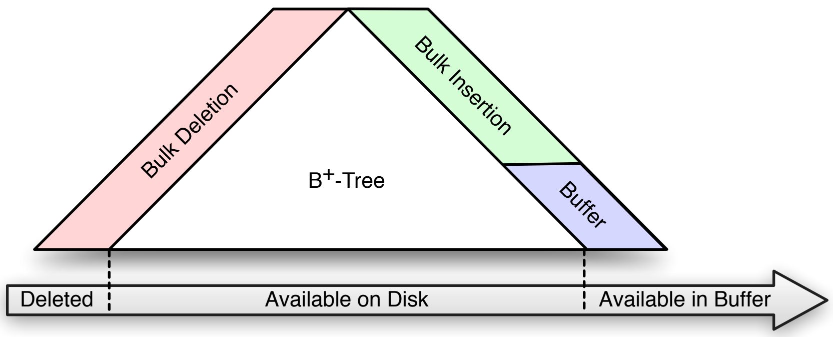


Figure 17.1: Jumping B$^+$-tree on event stream

both directions) and keep a reference to the disk page with the last inserted event. Besides their use for bulk insertions, the buffers also allow to directly access the most recent disk pages without querying the B$^+$-trees. The B$^+$-tree event store is scalable by distributing the indexes across multiple disks. In the best case, each event stream is written to its own and dedicated disk resulting in the highest possible write performance. Because event streams are potentially unbounded, we have to remove the oldest events when the available disk space is completely utilized. A different strategy would be to aggregate them. The indexes support fast temporal point and range queries that can be used to efficiently explore the B$^+$-tree event store.

Figure 17.1 illustrates all important characteristics of our so-called *jumping B$^+$-tree*, which is tailor-made for recording and indexing high-speed event streams. Note that new events are appended right-hand side in the figure. The most recent events of a stream are stored in disk pages that are buffered in main memory. When the buffer overflows, the B$^+$-tree is purged first, if necessary, to free enough space on disk. Therefore, the left branches of the B$^+$-tree are removed by deleting all corresponding disk pages and adapting the root node. Then, all buffered disk pages are moved from main memory to the leaf level of the jumping B$^+$-tree and a bulk insertion is performed to efficiently create the branches at the right side of the tree.

To give some details about the performance of the B$^+$-tree event store, we carried out the following experiment. The test setup consisted of N parallel running event streams. Each test event had a size of 28 bytes. A disk page had a size of 4 KiB and could keep 145 test events besides the metadata. Test events were pushed evenly into all N event streams. The B$^+$-tree event store used only one commodity disk (WD1002FAEX with average access time of 8.9 ms) for recording all streams and accessed it as a raw device. Each buffer had a size of exactly 100 disk pages.

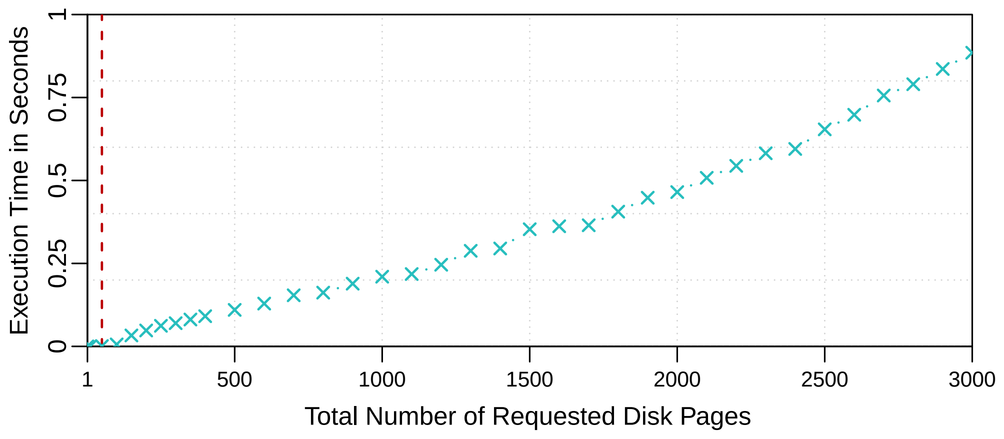


Figure 17.2: Read performance of the event store

We pushed test events at the maximum possible rate and made the following observations and measurements. First, the number of parallel running event streams N (we tested different configurations ranging from 1 to 100 parallel running event streams) had no significant impact on the write performance, even when all indexes were placed on the same disk. Second, the B$^+$-tree event store recorded about three million events per second on average.[1] This means that the B$^+$-tree event store wrote events at a rate of about 80 MiB per second, which is close to the maximum write rate of the used disk. During the experiment, the CPU utilization was quite low so that the disk was clearly the limiting factor. Thus, the write performance can be scaled up by using multiple disks. For examining the read performance, we made travels from the end of a recorded event stream back in time (i.e., disk pages were requested in inverse order of their writing). Figure 17.2 shows the average time for reloading events from the B$^+$-tree event store as a function of the number of requested disk pages. The costs for reading 1,000 disk pages were about 200 milliseconds. This corresponds to a read performance of more than 725,000 events per second. Before the point 50 (that is exactly half the size of the buffers) on the horizontal axis (marked by a dashed line), almost all disk pages were delivered by the buffer located in main memory. After the marked point, disk I/Os were necessary in general. Therefore, the needed time scaled linearly with the number of requested disk pages starting at the point 50.

[1]Relational standard database systems reached a write performance in the order of a few thousand events per second on the same machine.

Summary

Despite the fact that stream processing is a main memory technology, there are good reasons for archiving entire event streams or at least their most recent histories. JEPC meets the requirements of applications that need to archive events by supporting the connectivity to event stores. In this chapter, we introduce the interface used to abstract from event stores and present three different implementations in total. One implementation is based on JDBC and allows to use any standard database system as event store. Another implementation maintains recorded events in list data structures in main memory. Because the first implementation has poor write performance and the second does not durable store events, we present a third implementation that records event streams with respectable write performance on external storage. It is tailor-made and stores events in B$^+$-trees on disk. Because of the append-only nature of streams, we add events efficiently in batches via bulk insertions.

18

A General Approach to Efficiently Updating Continuous Queries On-the-Fly

Outline

18.1 Introduction

Today's SPEs are not able to dynamically adapt to semantic changes in the application context. This becomes a serious problem in context-sensitive applications. For example, the normal total traffic in a computer network may differ noticeably between days during the week and days at weekend so that suspicious values during the week could be totally normal at weekends and vice versa. Because parameters are time-varying in context-sensitive applications, CQs must also be time-varying.

In this chapter, we present a general approach to efficiently updating the definitions of CQs at runtime. It achieves both efficiency and correctness at the same time by utilizing a database for keeping the most recent histories of data streams. In particular, the recorded histories of data streams are exploited to completely fill up operator states of newly created CQs rapidly. Our proposed approach allows to trigger query updates as a reaction to changes in the application context and makes stream processing applications adaptive therefore. Its implementation can be purely on top of a SPE without the need to modify a single line of the source code.

18.2 Background

A serious problem with existing SPEs is that applications implemented with them are static during runtime. The matchmaker (see Chapter 16) already overcomes the problem of static connections. But still, no existing SPE allows to arbitrarily adapt the processing logic (i.e., the set of all running CQs) to semantic changes in the application context on-the-fly. This is due to the lack of functionality for updating CQs.

18.2.1 Problem Description

The problem of static CQs prevents the use of modern SPEs in some interesting application domains. On closer inspection, also many application domains addressed by modern SPEs are context-sensitive. For example, the support of anomaly detection became an indispensable requirement for security monitoring applications [Bau15, Gar12] and many other real-time monitoring applications. Anomalies are defined as a remarkable difference from some normal state [CBK09]. Because a normal state may be valid only within a certain context and change over time, static CQs are not an appropriate foundation to support anomaly detection [HS13]. In fact, because stream processing applications are intended to be long-term running (for weeks, months or even years), most applications will require adjustments sooner or later.

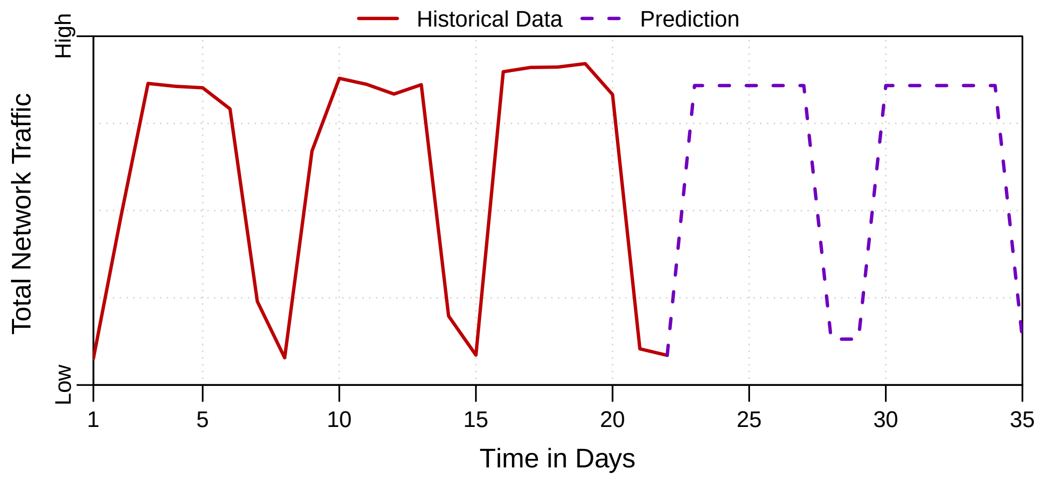


Figure 18.1: Total network traffic of an ISP

Figure 18.1 plots the real total network traffic of an Internet service provider (ISP) in England as a time series.[1] The solid line is a section of the time series in the range of three weeks. Obviously, the time series has different normal states. At five days a week the total network traffic was significantly higher than at the other two days. The two days with lower traffic were the days at weekend. For detecting anomalies (e.g., DDoS attacks), users could wish to deploy the query shown in Listing 18.1.

```
(SELECT *
 FROM (SELECT AVG(traffic) AS avgTraffic
       FROM TotalNetworkTraffic WINDOW(TIME 10 MINUTES)
      ) AS AvgQuery
 WHERE avgTraffic < normalTraffic - tolerance
    OR avgTraffic > normalTraffic + tolerance
) AS AnomalyDetectionQuery
```

Listing 18.1: Anomaly detection query

The listed query first computes the average total traffic within a time window of size 10 minutes. This is simply for smoothing the time series in order to become insensitive to noise. Then, the averaged total network traffic is checked whether it is normal. The normal value is specified by $normalTraffic$ while $tolerance$ is any positive number that allows for small deviations. This example definition is typical for anomaly detection queries that check measurements whether they exceed a threshold. However, normal values are often not static and depend on the context (e.g., time, location,

[1]Data made available by [Cor12].

or states of other objects). Figure 18.1 clearly shows that the value of $normalTraffic$ is dynamic and depends on the time. Therefore, the query definition must be periodically updated. In case of the example query above, the value of $normalTraffic$ can be predicted by a model we created specifically for the presented time series. On basis of the history of the time series, we trained a hidden Markov model (HMM) [RJ86] according to [WWW11]. In short, we segmented the time series via the Douglas-Peucker algorithm [DP73]. The resulting segments were combined by using hierarchical clustering [MR10]. Finally, we used the clusters to train a HMM. This model can be used to forecast the future values of the time series for a given sequence of values. The predictions made by our model are indicated as a dashed line in Figure 18.1.

Because arbitrary query updates are not supported by existing SPEs, anomaly detection queries cannot be executed properly. In practice, there exist workarounds for still executing some of them. One common workaround involves obtaining the current value of $normalTraffic$ from a databases system via a join operation. But this workaround not only decreases the query performance because of the database querying, it is also quite limited. It can only update constant values and only at places in a query definition where database access is supported. Another naïve workaround is substituting query updates by stopping the outdated query and starting a new one that has the updated query definition. But this procedure has serious consequences. Many queries including the query shown in Listing 18.1 are stateful. A newly started stateful query needs some time until it produces semantically correct results. For instance, a newly started query with the definition shown in Listing 18.1 requires 10 minutes until it reports semantically correct results. In the meantime, results must be ignored. A monitoring system would be blind and not able to detect anomalies such as denial of service attacks on the network. The issue becomes even more critical, if the query state requires more time to become consistent than the time between two updates. Then, a query will never reach a consistent state, because it is always replaced by a new one before. In the motivating example, this is true when the normal total network traffic changes from low to high and from high to low. These changes are gradually and require multiple updates of $normalTraffic$ in a very short time.

Note that not only filter values should be updatable. Fully dynamic query definitions must also allow to modify everything else. For example, it must be supported to change the sizes of windows, to replace aggregates, and to completely redefine patterns. Furthermore, it must be possible to replace, add and remove entire operators. In extreme cases, a query definition must be exchanged for a completely different one. Since the problem is vacant in all existing SPEs, a general solution is desirable.

18.2.2 Dynamic Event Processing

In the future, event processing applications will be required to be more and more adaptive at runtime. This, of course, includes the entire interconnection of all elements of an application that can be made adaptive by using the matchmaker. But for fully dynamic EP applications also queries must be adaptive. The combination of both results in a new kind of event processing that we call *dynamic event processing* (DEP). Despite the fact that DEP can be implemented via extensions, we think that every general-purpose SPE should have native functionality for (efficiently) updating CQs at runtime. The importance of update functionality becomes clear when SPEs are compared with their counterpart that are standard database systems. On the one hand, DBMSs and SPEs have common roots. On the other hand, the paradigms of both classes of information systems are inverse to each other. Data items are persistent and queries are volatile in DBMSs. In contrast, the roles of data items and queries are interchanged in SPEs so that queries are persistent and data is volatile. The set of all persistent objects in an information system (data items in DBMSs and CQs in SPEs) represents the current knowledge base and has a lifecycle that should be manageable via basic methods to add, remove and update persistent objects. Table 18.1 compares DBMSs and SPEs with respect to the supported lifecycle management methods.

Database systems	*Stream processing engines*
Adding data items	Adding continuous queries
Removing data items	Removing continuous queries
Updating data items	———

Table 18.1: Natively supported basic lifecycle management methods

As shown in Table 18.1, adding and removing of data items in DBMSs and of CQs in SPEs are an essential part of their basic functionality, but updates are natively supported only by DBMSs. While an update can be substituted by a remove operation followed by an add operation in case of DBMSs, this procedure cannot be used in case of SPEs for the following reasons. First, a running CQ cannot be simply replaced by a new one because CQs are stateful in general. Removing a running stateful CQ removes also its state so that there is a loss of information. In addition, a stateful updated version of a CQ starts with an empty state and produces incorrect results until its state is completely filled up. Second, it is nearly impossible to perform updates deterministically and reproducibly this way. Both problems are not acceptable in business-critical applications. Therefore, CQs definitely require a native update method.

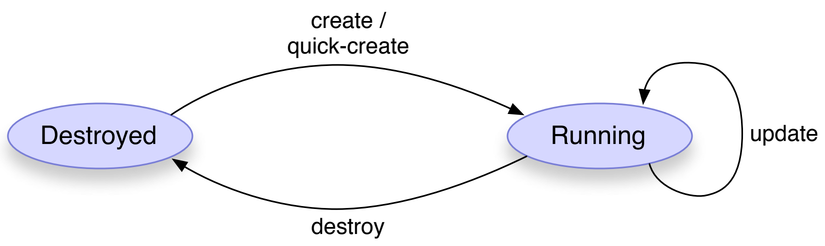


Figure 18.2: States and transitions of continuous queries

Figure 18.2 shows the most important states in which a CQ can be and all possible transitions between them. A CQ starts in the state *Destroyed*. This means that the CQ is neither running nor even known by the SPE. The state changes to *Running*, when the CQ is initially created or quick-created (a technique for quickly starting a new query is part of the update method presented in the next section). A running CQ can be destroyed at any time. Then, its state changes to *Destroyed*. The last transition is for updating a CQ on-the-fly. Thus, the CQ does not leave the state *Running*.

18.3 Update Method

In this section, we present our general update method. The only restriction is that a new definition must preserve the output schema of the query being updated. Note that this restriction ceases to exist when the matchmaker is used (see Chapter 16). When a query is updated, there is usually the requirement to enable its new definition as fast as possible. Therefore, our update method replays the most recent histories of the input streams of a new query definition to quick-load all contained operator states immediately after its creation. The switch from one definition of a query to a another one should be deterministic and reproducible. This aspect also influenced our design of an update method. Furthermore, we wanted our update method to be universally implementable in as well as purely on top of different SPEs. The latter forced us to use only basic data structures and functionality offered by all modern SPEs.

18.3.1 Preliminaries

The main algorithm of our update method needs some preparations that are described in the following. Data streams must be enriched with additional information. A special data sink is needed to maintain different running versions of the same query. We also have to integrate a database for recording and replaying data streams. Finally, a special data source that allows for quick-loading of queries is introduced.

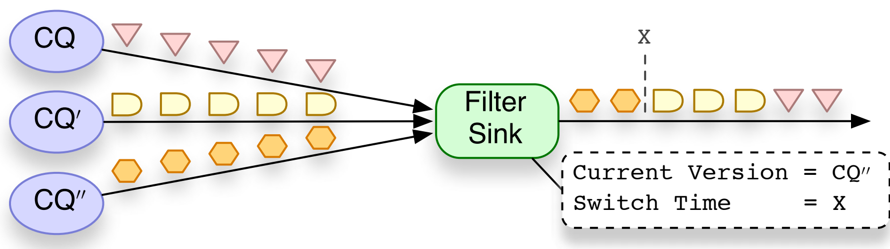


Figure 18.3: Filter sink

18.3.1.1 Data Streams

The first preparations concern the output schemas of all CQs. They are extended by additional attributes. The first attribute that is added is named *version*. It is needed because CQs can exist in different versions and only one of the versions is allowed to be active at a time. Therefore, each output data item of a CQ contains the version number of the CQ by which it was created. To set the version information, a simple map operation is integrated into every query definition automatically. Another attribute must be added only for data streams that do not contain an external time information. In such cases, the missing time attribute is added to the schema and each incoming data item is timestamped by the system clock.

18.3.1.2 Filter Sinks

Whenever a completely new CQ (i.e., the initial version) is created, a so-called *filter sink* is created automatically and added simultaneously. A filter sink is implemented as an ordinary data sink. Its task is to filter incoming data items on basis of its current state. The state of a filter sink consists of a version information (`Current Version`) as well as a temporal information (`Switch Time`). It can be set and updated from the outside. Each filter sink consumes the output stream of its corresponding CQ and also the output streams of all updated versions of the corresponding CQ. The version information of the state of a filter sink determines the latest version of the CQ and the temporal information determines the instant of time when the latest version has or will become valid. With the help of a filter sink, we can forward all output data items that were produced by the valid version of the corresponding CQ and block all output data items coming from an outdated version or a version that is not valid yet. In Figure 18.3, this use of a filter sink is illustrated for a CQ and two updated versions of it. Algorithm 28 describes how a filter sink handles incoming data items.

```
Algorithm 28: PushFilterSink((p, t))
  Input: Data Item: $(p, t)$
  Data: Timestamp: $switchTime$, $clock$,
        Number: $currentVersion$
1 if ($p.version = currentVersion$ and $t \geq switchTime$)
2  or ($p.version = currentVersion - 1$ and $t < switchTime$) then
3     $clock \leftarrow t$;
4     Payload $p' \leftarrow$ RemoveVersionAttribute($p$);
5     Forward(($p'$, $t$));
```

A filter sink works as follows. If the timestamp (in case of time intervals the start timestamp) of an incoming data item is equal to or greater than the point in time specified by *switchTime*, then the incoming data item is forwarded to the output if and only if it was produced by the version that is specified by *currentVersion*. And if the timestamp of an incoming data item is less than the instant of time specified by *switchTime*, then the incoming data item is forwarded to the output if and only if it was produced by the predecessor of the latest version (*currentVersion* - 1). In other words, to achieve a clear, deterministic and reproducible switch from one version to a new one, a certain point in time (*switchTime*) is defined for it. Before this instant of time, the original version is the valid one. And beginning from the defined instant of time, the new version is the valid one. Figure 18.3 shows a filter sink in action with its corresponding CQ, an update of the CQ (CQ') and an update of the updated version (CQ''). The output stream consisted of events from CQ at its beginning. At some later point in time, the filter sink switched from CQ to CQ'. From this point in time, the output stream consisted of data items from CQ' while all data items from CQ were blocked. The latest switch happened at the point in time x and enabled CQ''. Thus, data items coming from CQ as well as CQ' were blocked and data items coming from CQ'' were forwarded starting at the point in time x.

Every filter sink makes available all valid incoming data items in the form of a new data stream and removes the version attribute beforehand. Note that the timestamp of an output data item clearly identifies the version of its corresponding CQ, because at most one version is valid at a point in time in our approach. All data sinks and all CQs that consume the output of a CQ are redirected to the output stream of its corresponding filter sink. The last feature of a filter sink is that it can report the timestamp of the last data item it put into its output stream (*clock*).

18.3.1.3 Stream Store

For efficiency reasons, our update method requires a database system, called *stream store* in the following, capable of recording and reloading data streams. To support fast updates and high update rates, our update method needs access to the most recent histories of data streams. Of course, we use one of the provided event stores in case of JEPC (see Chapter 17). The output streams of all external data sources and the output streams of all filter sinks must be recorded (i.e., all data streams than can be queried). A replay of a data stream always begins at some past point in time t and gives all events that happened since then in temporal order. In case of time-instant semantics, a temporal range query that returns all data items in correct order is sufficient. But in case of time-interval semantics, a replay is a bit more complex. At first, all events that were valid at t must be reloaded in correct order. Note that the start timestamps of those events can be any points in time before t. Then, a temporal range query gives all events that started after the point in time t. The performance of updatable CQs (i.e., the time that is needed to enable a new definition) and the overhead (i.e., how much is the overall throughput decreased) depend on the read and write performance of the used stream store. Thus, the use of a high-performance stream store such as our in-memory and B^+-tree event stores is crucial in challenging applications.

18.3.1.4 Merge Sources

We developed a special kind of data source that we call *merge source*. A merge source consumes exactly one existing output stream of either a CQ or an external data source and forwards every incoming data item to its output stream. Merge sources are connected with the used stream store. After its creation, a merge source loads the most recent historical data items of its input stream from the stream store. A parameter that was set when the merge source was created specifies how much historical data is loaded. Inside a merge source all incoming data items (historical data items as well as data items from the live input stream) are put into a heap data structure that sorts them by time. The result is that the output stream of a merge source consists of historical data at its beginning followed by the live input stream (see Figure 18.4).

The necessary amount of historical data to completely fill up all operator states of a query is specified in the form of a past timestamp t_{min} from which on the most recent history of the corresponding data stream must be replayed. This past point in time must be determined by analyzing the CQ to load and the recorded data streams in general. Moreover, it depends on the query semantics and, thus, on the underlying

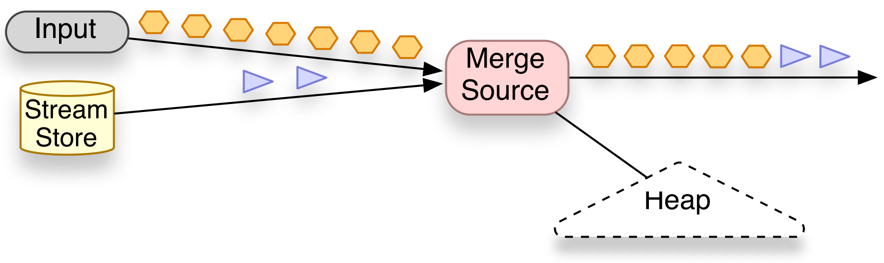


Figure 18.4: Merge source

system. In the special case of JEPC, we initially set t_{min} to the current clock of the corresponding event stream. Then, we propagate it from the input to the output of the operator plan of the query to load and update it at each operator. At a time-based window operator of size w, we reduce t_{min} by the size of the time window w. And at a count-based window operator of size N, we reduce t_{min} by the size of the time range which is the union of the time intervals of the N most recent events of the corresponding stream. At filter EPAs t_{min} is not updated at all and at aggregation EPAs it is reduced by one time instant. This is because filter EPAs are stateless and aggregation EPAs delay the clock by one time instant. At a pattern matching EPA, we reduce t_{min} by the size of the within condition. And at a correlation EPA, we set t_{min} to the minimum clock among the clocks of all input streams of the correlation EPA (the clocks of all other input streams of the query to load are propagated simultaneously and, thus, known). Once t_{min} reaches the output of the query to load, it has its final value that is used to replay the correct amount of historical events.

If the stream store cannot provide all necessary historical data, a merge source waits until the live data stream has delivered the missing amount of data. Obviously, this increases the wall-clock time to load a query. However, stream processing applications are long-term running and it can be assumed that sufficiently many (external) data items have been recorded in general. The only exception is that the input data stream of a merge source is the output stream of another CQ that has been initially created not long ago. But in this special case, a copy of that CQ can be executed on historical data in order to generate the needed historical data items.

Eventually, a merge source pushed all historical data items into its output stream. Then, a heap is no longer necessary because new data items are coming only from the live input stream that is correctly ordered by default. In this case, a merge source can push the entire heap into the output stream, delete the heap and directly connect the input stream to the output stream for performance reasons.

18.3.2 Algorithm

Our method for updating CQs on-the-fly combines the components introduced in the last section as shown in Algorithm 29. It requires the identifier of the CQ to update and the new query definition. At first, merge sources are created for all input data streams (lines 1–4). Each input data stream must be replaced by a new merge source that provides not only the corresponding input data stream, but also its most recent history. Therefore, the algorithm iterates over all input data streams (that can be the output data streams of data sources as well as other CQs) of the new query definition. The needed amount of historical data is determined individually for each input data stream and the result is used to configure a new merge source. Finally, the new query definition is modified by exchanging an input data stream for its corresponding merge source. After all merge sources have been created and integrated into the new query definition, a new CQ on the basis of the new and modified query definition is created (line 5) and connected with the associated filter sink (lines 6–7). Then, the algorithm waits until the new CQ is completely loaded. In particular, this means that every merge source must have pushed sufficiently much data from the stream store and, if necessary, from the live data stream. As soon as the new CQ is completely loaded, the state of the associated filter sink is updated to finish the query update

Algorithm 29: UPDATEQUERY($\mathcal{C}$,$\mathcal{D}$)

```
Input:  Query Identifier: $\mathcal{C}$, Query Definition: $\mathcal{D}$
Output: Timestamp: $switchTime$
1  foreach Input Stream $stream \in \mathcal{D}$ do
2      Timestamp $t_{min} \leftarrow$ DETERMINEREQUIREDHISTORY($stream, D$);
3      Merge Source $mergeSource \leftarrow$ NEWMERGESOURCE($stream, t_{min}$);
4      $\mathcal{D}$.REPLACE($stream, mergeSource$);
5  Continuous Query $\mathcal{CQ}' \leftarrow$ CREATEQUERY($\mathcal{D}$);
6  Filter Sink $filterSink \leftarrow$ GETFILTERSINK($\mathcal{C}$);
7  $\mathcal{CQ}'$.ADDDATASINK($filterSink$);
8  while not ISLOADED($\mathcal{CQ}'$) do
9      WAIT();
10 Timestamp $switchTime \leftarrow filterSink.clock + 1$;
11 $filterSink.currentVersion \leftarrow \mathcal{CQ}'$;
12 $filterSink.switchTime \leftarrow switchTime$;
13 return $switchTime$;
```

(lines 10–12). To update the state of the filter sink, the version information is set to the new CQ and the temporal information is set to the first point in time of the future. Finally, the algorithm returns the point in time at which the switch to the new query definition will happen. On basis of this information, it can be clearly identified which version of a query produced a specific output data item. The choice of the point in time $filterSink.clock + 1$ for the handover is discussed in the following.

Theorem 7. *The point in time $filterSink.clock + 1$ is the earliest one that guarantees a deterministic and reproducible switch to the updated query definition. That is, for each point in time there is at most one version of the query that produced all output data items.*

Proof. The original version of a query being updated has not reached this point in time yet, because otherwise the requested clock of the filter sink would be greater than $filterSink.Clock$. Both the original and the updated version of the query are running in parallel and are in consistent states. So it is correct to set the temporal information to $filterSink.clock + 1$, because this ensures that for each point in time there is at most one version of the query that produced all results. Setting the temporal information to $filterSink.clock$ or to an earlier point in time would lead to a non-deterministic and not reproducible switch. Because the filter sink reported $filterSink.clock$, there are already results produced by the original version with timestamp set to $filterSink.clock$ and forwarded to the output. If the switch to the updated version of the query happens immediately, it might be possible that results of it with timestamp set to $filterSink.clock$ are forwarded to the output too. As a consequence, there might be results from different versions at the same instant of time. This is neither deterministic nor reproducible and therefore not a correct switch. In summary, a switch from the original version to the updated version of a query at the point in time $filterSink.clock + 1$ is deterministic and $filterSink.clock + 1$ is the earliest point in time that ensures determinism. □

Figure 18.5 illustrates the algorithm by example.[2] The query CQ_2 consumes the output streams of a data source and CQ_1. As explained earlier, whenever the output stream of a query is requested, the consumer is redirected to its associated filter sink. The output stream of CQ_2 is consumed by CQ_3 and a data sink. An update of CQ_2 creates CQ'_2. For each of the input streams of the updated version (the new version consumes the same data streams but is not forced to) a merge source is created. Finally, the updated version is connected with the filter sink of the original version. As soon as possible, the output stream of this filter sink will contain results produced by CQ'_2.

[2]Note that the presented update of the query CQ_2 is not supported by the only competing method presented in [She11], because CQ_2 has multiple inputs.

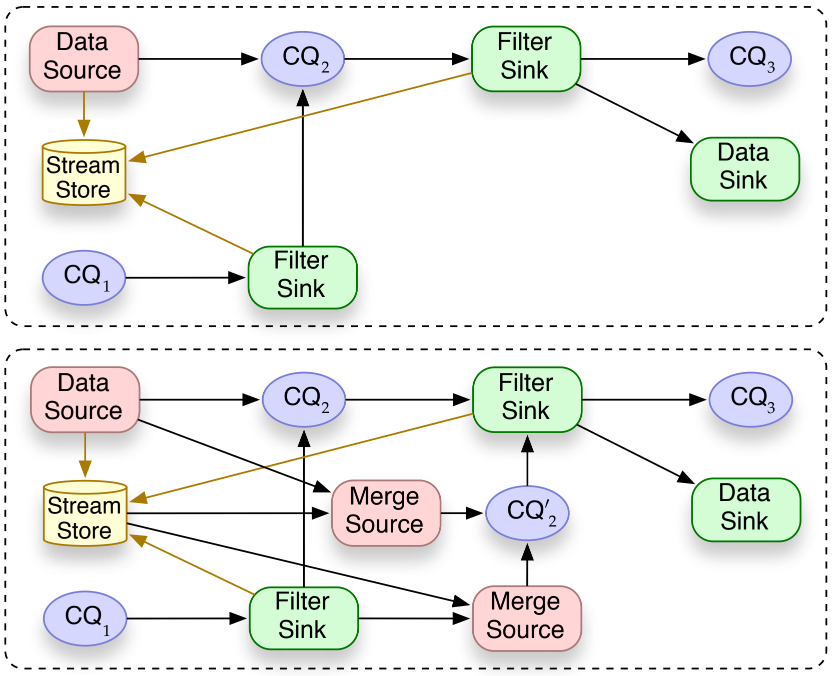


Figure 18.5: Update of CQ_2 at runtime (top before, bottom after the update)

18.3.3 Implementation

The general design of our update method enables it to be implemented purely on top of any modern SPE (this is why we refer to it as *general*). Independent of specific SPEs, data sources and data sinks are external components that can be user-defined via arbitrary code. Therefore, the implementations of the merge source and the filter sink as well as the bidirectional connectivity (recording and reloading) with a stream store are generally possible. The update algorithm itself can also be on top of a SPE. It depends only on a method to create new CQs within the SPE. However, such a method is part of the basic functionality of all SPEs. By now, we have already implemented the filter sink, the merge source and the update algorithm on top of the SPEs Esper [Esp], Odysseus [App12] and webMethods Business Events [web]. For performance reasons, we added a garbage collection (GC) to destroy outdated query versions automatically. A version of a query becomes outdated when the switch to a newer version has happened (i.e., not immediately when the update algorithm terminates). If the outdated version was an update itself (i.e., it is not the initial version), all corresponding merge sources are removed too. We also implemented a slightly modified variant of our update method within the matchmaker extension (see Chapter 16).

In our current design and implementation, the behavior of the update method is greedy. Once an update has been initialized, it is enabled as fast as possible. This behavior has sound semantics, because the update method returns the point in time when the switch between the two versions will be performed. Fast but fuzzy update execution is not only an important requirement of most applications, but also compliant with the behaviors of the add and remove methods in most SPEs. An alternative behavior, which would require only slight modifications of the method, would be to let users specify a concrete future point in time at which the switch will happen.

18.4 Use Cases

A method to arbitrarily exchange definitions and plans of continuous queries at runtime allows to holistically manage their life cycles and has a broad range of applications. Three of them are discussed in this section.

18.4.1 Anomaly-Based Detection

Besides signature-based and behavior-based detection, anomaly-based detection [CBK09] is one of the most important monitoring paradigms and essential for many application domains. In case of anomaly-based detection, the current behaviors of objects are compared to their individual normal behaviors. If the current behavior of an object differs significantly from its normal one, then it is classified as anomalous. One challenge of anomaly-based detection is that the normal behaviors usually depend on the context such as time, location or states of other objects and may evolve over time. There is no static normal behavior of an object in general. A simple but illustrative example is given in Section 18.2.1. To support anomaly-based detection in EP applications adequately, it is important to be able to change the processing logic when the context changes. Therefore, CQs must be updatable on-the-fly [HS13].

18.4.2 Elastic Windows

Today's SPEs force users to specify fixed sizes for all windows. When looking at join queries for example, fixed-sized windows result in variable output sizes. This is due to variable input rates of data items in the special case of time windows and due to the content of incoming data items in conjunction with the user-defined join condition in general. In contrast, provided that the output size shall be fixed, sizes of windows must be variable. This kind of elastic windows is not provided by existing

SPEs [LGP10], but is made possible by a method to update queries at runtime. Fixed output sizes of continuous queries might be a quality of service (QoS) requirement of applications. In particular, too large output sizes of continuous queries not only result in waste of system resources (e.g., CPU, main memory, network, disk space), but also may cause problems for subsequent consumers. For example, screens and humans can process only a certain amount of results at once. Screens are limited by their size and resolution. They can show only a limited number of results at a time. Also humans are limited. They are only able to gather and analyze a few results within a short period of time. In many applications, SPEs can only be used to prepare or preselect data items. Final decisions must be made by a human expert that analyzes the prepared or preselected data items on a screen. A common task of monitoring applications is to find the *k* most interesting elements within a large quantity of different elements. Three examples are discussed in the following. In case of an online auction, a professional trader could constantly seek the most up-to-date *k* most interesting auctions. Or in case of Twitter [Twi], a journalist could invariably look for the most up-to-date *k* most interesting Tweets. We decided to discuss how to find the *k* most interesting stocks as an detailed example. The query in Listing 18.2 shows a reasonable approach to finding the most interesting stocks in a data stream `StockPrices` that contains the latest exchange prices (`price`) of different stocks (`stockID`).

```
(SELECT *
 FROM  (SELECT   stockID, MAX(price)/MIN(price) AS fluc
        FROM     StockPrices WINDOW(TIME x SECONDS)
        GROUP BY stockID) AS ElasticWindow
 WHERE  fluc > 1.025
) AS FixedOutputRate
```

Listing 18.2: Elastic time window

Assume that every second a new data item is pushed into the data stream for each individual stock. In Listing 18.2, the inner query definition contains a time-based sliding window with a size of x seconds on `StockPrices`. The window is used to compute the minimum and maximum exchange prices within the last x seconds per stock. These values are simple but important key performance indicators (KPI) in the field of financial analysis. A stock is classified as interesting if its exchange price fluctuated significantly (more than 2.5 % in case of the listed query) within a short period of time. The outer query definition selects all interesting stocks and forwards them to a screen which shows them to human experts. Ultimately, the human experts are responsible for deciding which of the reported stocks should be sold or bought.

Since screens as well as human experts can process only a limited number of data items at a time, we only want the most up-to-date k most interesting stocks to be reported every second. But the number of results is variable, because it depends on the content of incoming data items as well as the size x of the time window. Additionally, the number of incoming data items at each point in time could be highly variable and this would also influence the output size. The content of incoming data items and the input rates cannot be influenced, but the window size x can be changed in the presence of an update method for continuous queries. Therefore, another continuous query counts all stocks reported by the listed query within the last second. If the second query detects more than k reported stocks within the last second, an update of the listed query is triggered and the size x of the time window is decreased. Similarly, an update of the listed query increases x, if less than k stocks were reported within the last second. Changing the sizes of windows is an appropriate way to express such queries. This approach not only optimizes the utilization of system resources, but also finds an optimal setting for time constraints. Because the content constraint is fixed in the example (i.e., the fluctuation must be greater than 2.5 %), the time constraint (i.e., the size x of the time window) must be tightened and relaxed to control the output size, because the computed minimum and maximum prices of each stock move together when x is decreased and move apart from each other when x is increased.

Another meaningful application of elastic windows is given in [LGP10]. Here, the sizes of time windows are adapted to the individual lengths of periods contained in monitored time series. Such time series that include periods with individual lengths occur, for example, in heart acceleration streams that come from body sensors.

18.4.3 Query Optimization

Because continuous queries are long-term running and important characteristics of incoming data streams can change over time, the query plans of continuous queries must be optimized not only before execution, but also occasionally during execution. Dynamic plan migration (DPM) techniques have been developed to allow for continuous query optimization. In particular, DPM techniques exchange the plan of a running query for an optimized but semantically equivalent one.

The presented update method for continuous queries allows to exchange a query plan for any other query plan. Of course, this includes all semantically equivalent ones. Therefore, DPM is only a special case of the problem addressed by our update method. In fact, the JEPC query optimizer (see Chapter 19) supports the continuous optimization of queries by utilizing our update method for DPM.

18.5 Related Work

Our problem of updating the definitions of CQs on-the-fly is related to the problem of dynamic plan migration [Krä06, Yan07, ZRH04]. Because the workloads of CQs may change over time, their query plans can become suboptimal. DPM replaces the query plan of a CQ by another equivalent one at runtime. This allows to optimize CQs continuously by exchanging active query plans for more efficient ones whenever they become suboptimal. Approaches to dynamic plan migration assume that the input data streams of a query, the operators of a query plan as well as the sizes of windows are not changed so that the new query plan is semantically equivalent to the old one but has a different structure. For example, the order of join operators can be changed via DPM. While dynamic plan migration is at a technical level and does not modify the semantics of CQs, our update problem is at the application level. We explicitly want to change the definitions of CQs as reactions to semantic changes in their context. Query updates include DPM as a special case, but are much wider applicable than DPM. Because the assumptions made by DPM prevent arbitrary query updates, all proposed DPM techniques cannot perform arbitrary query updates.

To the best of our knowledge, there has been proposed only one approach to updating CQs so far. This approach [She11] lets users decide how to deal with the problem of operator states during an update. It provides basically two different strategies to users. One strategy is exactly the naïve one discussed in Section 18.2.1. That is, an update is a completely new CQ that becomes immediately valid. Due to empty operator states, this CQ produces incorrect results until all states are completely filled up. The other strategy delays the handover between the original version and the updated one. That is, both CQs run in parallel until the updated one starts producing correct results. Then, the handover can be performed. So there is always a trade-off between correctness and performance. Either the update is fast and incorrect or it is correct and slow. In contrast, our approach is correct and fast at the same time. There are also other issues with the approach of [She11]. It requires the modification of built-in operators and the feature to push control elements [Tuc03] in data streams. Therefore, a complete rewrite of the SPE is necessary. In contrast, our approach can be easily implemented on top of any modern SPE without the need for modifying a single line of its source code and does not depend on system-specific functionality. An important disadvantage of [She11] is that this approach can only update queries with exactly one input data stream. Already queries consisting of a single join operator cannot be updated. Our approach is able to update any query.

Summary

This chapter presents an approach to updating continuous queries at runtime. Because continuous queries are running for weeks, months or even years, such an approach is needed in order to manage their lifecycles. The proposed approach achieves both high performance and correctness at the same time by utilizing a database system, whereas the state-of-the-art approach is only able to achieve correctness or performance but not both at the same time. Furthermore, the presented approach is general and can be implemented purely on top of existing SPEs for adding the functionality of query updates. As a consequence, SPEs can finally get a rich and complete set of methods to manage the lifecycles of implemented applications. SPEs that have a method for updating queries provide a powerful data stream processing infrastructure which allows for dynamically changing its processing logic. Different use cases show possible applications and the benefits of this kind of adaptive stream processing.

19

Query Optimization

Outline

19.1 Introduction

In expressive query languages, a complex query has multiple semantically equivalent query plans in general. End-users specify queries in languages such as SQL or directly as query plans usually with focus on correctness rather than on performance. Nevertheless, queries should be executed ideally not only semantically correct, but also with high performance. This requires that only efficient query plans are executed. Therefore, a query optimizer that transforms query plans into efficient ones before their execution is part of the query compilers of almost all database systems.

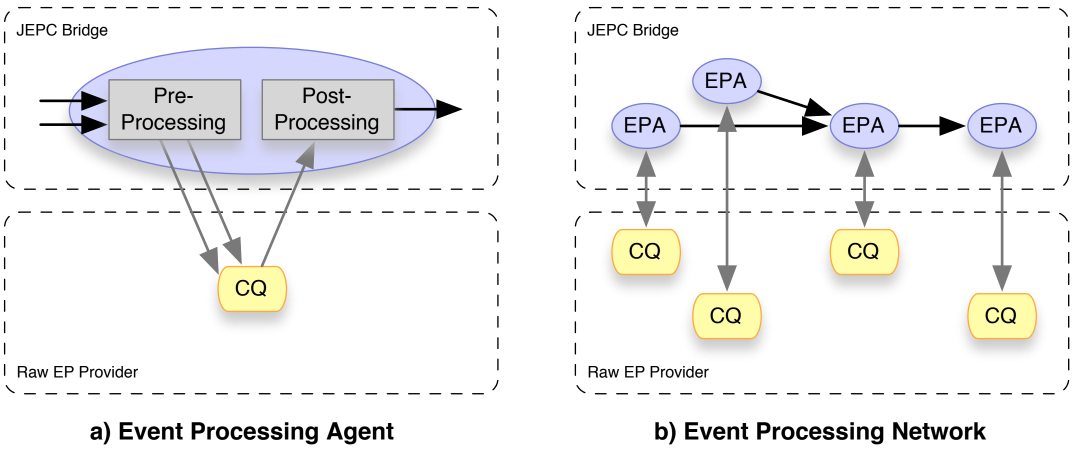


Figure 19.1: Translation and interconnection of EPAs

Although optimization of continuous queries has gained much attention in the past, today's general-purpose stream processing engines support almost no built-in query optimization at the query plan level [GV04, Hei08, Rie08, SHG13]. Moreover, a raw EP provider of JEPC must not necessarily be a stream processing engine (e.g., the JEPC-to-JDBC bridge from Chapter 9 or the native EP provider from Chapter 10) so that the existence of an optimizer for CQs cannot be assumed. But even if a raw EP provider has a sophisticated query optimizer, it cannot be used because it is disabled by JEPC. This is due to the fine-grained translation of EPNs and the interconnection of EPAs completely in the middleware layer as it is illustrated in Figure 19.1. JEPC maps each single EPA of an EPN separately to a CQ running within the raw EP provider. All input events of an EPA are forwarded to the corresponding CQ. And the output stream of the corresponding CQ is directly received by JEPC where it is redirected to subsequent EPAs. Thus, the interconnection of EPAs is only known by JEPC. Raw EP providers just see a set of isolated and simple CQs that must be executed.

In this chapter, we present a powerful optimizer for JEPC queries that implements essential rule-based and cost-based techniques for transforming query plans. We mainly adopted techniques from query optimizers of database systems that are proven to be effective. However, work done in the database area focuses on one-time optimization of ad-hoc queries and on query plans that consist only of operators of the relational algebra. In contrast to traditional optimizers, the JEPC optimizer also works continuously and implements novel techniques to improve pattern matching EPAs.

19.2 Optimization Techniques

Because the semantics of JEPC queries is based on the powerful concept of snapshot-reducibility, well-known transformations of query plans can be performed without changing the output of a query [Sno87, SJS01]. In particular, algebraic laws of the relational algebra [GUW08] can be directly taken over. Those laws serve as a foundation for most techniques implemented by the query optimizer of JEPC.

19.2.1 Predicate Ordering

In JEPC, predicates can be combined via the logical connectors $\wedge$ and $\vee$ to arbitrarily complex Boolean expressions that are used as arguments to configure all basic EPA types except the aggregation EPA. The evaluation of a (nested) Boolean expression is typically done from left to right (and from the inside to the outside). Thus, there is an evaluation order of predicates. On basis of the commutative, associative and distributive laws that all apply to Boolean expressions, the order of predicates can be safely changed within a Boolean expression. The key idea of our first optimization technique is to order the predicates of a Boolean expression such that disqualifying events are detected with a minimum number of predicate evaluations. As soon as an event surely disqualifies, all remaining predicates must not be evaluated. Of course, predicates that discard more events on average than the other predicates should be evaluated first. This property of a predicate is determined by its selectivity *Sel*, which is one of the standard measures in query optimization. The selectivity simply gives the ratio between the number of output events and the number of input events:

$$Sel := \frac{|E^L_{out}|}{|E^L_{in}|}$$

The lower the selectivity, the more events are discarded. Therefore, an optimal evaluation order of predicates is an order according to increasing selectivity. Among others, this kind of optimization is also performed by Eddies [AH00] for example.

19.2.2 Filter Push-Down

Because a filter EPA does not create any new events but potentially discards events and reduces the size of the corresponding stream therefore, it is generally beneficial to place filter EPAs as close as possible to the sources of a query. This is analogous to the selection push-down in the relational algebra. As our evaluation clearly shows, every single EPA of a query plan including the relatively cheap filter EPAs causes costs. The filter push-down technique allows not only to discard events earlier, but also to integrate filter EPAs into other EPAs and, thus, to eliminate them.

19.2.2.1 Filter EPAs

At first, we focus on the case where a filter EPA is adjacent to another filter EPA. The most important algebraic laws regarding to this case are as follows:

$$\sigma_{\varphi_1}(\sigma_{\varphi_2}(E^L)) = \sigma_{\varphi_1 \wedge \varphi_2}(E^L)$$

$$\sigma_{\varphi_1}(\sigma_{\varphi_2}(E^L)) = \sigma_{\varphi_2}(\sigma_{\varphi_1}(E^L))$$

The first law allows to completely integrate a succeeding filter EPA into its predecessor by combining the filter conditions. Alternatively, the second law allows to swap adjacent filter EPAs and, thus, to change their evaluation order. Thus, the second law can be used to order filter EPAs by their selectivities similar to predicate ordering. However, the first law eliminates a filter EPA and our experiments clearly showed the superiority of this kind of transformation in all situations. Therefore, the JEPC optimizer always combines adjacent filter EPAs according to the first law.

19.2.2.2 Correlation EPAs

From an algebraic point of view, every correlation EPA is composed of the Cartesian product[1] followed by a simple filter EPA which checks the correlation condition:

$$\bowtie_{\varphi} (E_1^L, E_2^L) = \sigma_{\varphi}(\times(E_1^L, E_2^L))$$

Hence, the algebraic laws for adjacent filter EPAs can be simply reused:

$$\sigma_{\varphi_1}(\bowtie_{\varphi_2} (E_1^L, E_2^L)) = \sigma_{\varphi_1}(\sigma_{\varphi_2}(\times(E_1^L, E_2^L))) = \sigma_{\varphi_1 \wedge \varphi_2}(\times(E_1^L, E_2^L)) = \bowtie_{\varphi_1 \wedge \varphi_2} (E_1^L, E_2^L)$$

As a consequence, the same effects are achieved. Therefore, a succeeding filter EPA can be integrated into a correlation EPA so that the query plan is reduced by one EPA.

[1]The Cartesian product $\times$ can be simply defined as $\times(E_1^L, E_2^L) := \bowtie_{\texttt{TRUE}} (E_1^L, E_2^L)$.

Besides the above filter push-down technique, the query optimizer of JEPC also supports a second technique that can be applied directly to correlation EPAs. It is based on the following algebraic law:

$$\bowtie_{\varphi_1 \wedge \varphi_2} (E_1^L, E_2^L) = \bowtie_{\varphi_1} (E_1^L, \sigma_{\varphi_2}(E_2^L))$$

This law assumes that φ_2 refers only to attributes of E_2^L. Then, the law allows to push down the correlation condition (or parts of it) to the arguments of the correlation EPA. Of course, both techniques for correlation EPAs can be combined. A following filter EPA can be first integrated into a correlation EPA. Then, as many subexpressions as possible can be moved from the correlation condition to the sources.

19.2.2.3 Aggregation EPAs

The aggregation EPA is the only type of EPA that does not have parameters of the type Boolean expression. Therefore, the integration of succeeding filter EPAs is not possible. But if a succeeding filter EPA refers completely or partly to group attributes, then it can be moved completely or partly in front of a preceding aggregation EPA according to the following algebraic law:

$$\sigma_{\varphi_1 \wedge \varphi_2}(\alpha_a(E^L)) = \sigma_{\varphi_1}(\alpha_a(\sigma_{\varphi_2}(E^L)))$$

Assuming that φ_2 refers only to group attributes, the law allows to check the filter condition before the aggregation. If φ_1 is empty, the entire filter EPA can be moved.

19.2.2.4 Pattern Matching EPAs

Since the pattern matching EPA is not part of the relational algebra, there are no algebraic laws that can be taken over. But because the pattern a pattern matching EPA is looking for is a sequence of symbols each defined via a Boolean expression, a pattern matching EPA can be roughly seen as a sequence of filter EPAs. Moreover, at the position of the last Boolean expression, all values of potential output events are known. This is, because output events consist purely of global variables and their values can only be set within a pattern. On basis of this observation, the following algebraic law holds without any further conditions:

$$\sigma_{\varphi}(\rho_{s^1[\varphi_1]s^2[\varphi_1]\ldots s^n[\varphi_n]}(E^L)) = \rho_{s^1[\varphi_1]s^2[\varphi_2]\ldots s^n[\varphi_n \wedge \varphi]}(E^L)$$

According to this law, a succeeding filter EPA can be eliminated by integrating its filter condition into the Boolean expression that defines the last symbol in a pattern. This transformation can be applied without deep analysis of the pattern definition.

But if the pattern itself is analyzed, filter conditions can be pushed further down the sequence of symbol definitions. In particular, a filter condition φ can be moved to the directly preceding symbol definition, if and only if the directly preceding symbol definition does not set global variables to which φ refers. That is, φ can be safely placed in front of symbol definitions that do not set global variables used in φ:

$$\rho_{s^1[\varphi_1]s^2[\varphi_2]\ldots s^{n-1}[\varphi_{n-1}]s^n[\varphi_n \wedge \varphi]}(E^L) = \rho_{s^1[\varphi_1]s^2[\varphi_2]\ldots s^{n-1}[\varphi_{n-1} \wedge \varphi]s^n[\varphi_n]}(E^L)$$

The above law allows to push down φ from the definition of s^n into the definition of s^{n-1} if no global variable used in φ is set by the symbol s^{n-1}. Successively applied, this law allows to move filter conditions as close as possible to the input of ρ.

The transformations presented in this section altogether allow to push down filter conditions close to the sources in not only simply query plans, but also complex ones. In the latter case, it is often possible to push down a filter condition multiple times. Our experiments showed that the performance improvement of most filter push-down techniques is significant. This is consistent with traditional database systems where the selection push-down technique is one of the most powerful optimizations.

19.2.3 Correlation EPA Ordering

Multiway correlations that comprise three or more event streams can only be implemented in the form of multiple correlation EPAs, because correlation EPAs are binary. In this section, we focus only on left-deep trees of correlation EPAs in which the right child node of every internal node that is a correlation EPA is an input event stream. The advantage of left-deep trees is that they correspond to an order of input event streams and vice versa. For correlating N input event streams, $N-1$ binary correlation EPAs are required. In total, there are $N!$ different orders of the input event streams and, thus, also left-deep trees. The question is how to order event streams such that the left-deep tree representing a multiway correlation is executed with high performance. As in database systems, our strategy is to pick the order which minimizes the sizes of intermediate event streams. This can be done by adopting the algorithms used in database systems. Because this optimization is cost-based, we first have to define a cost model. Our cost model is derived from traditional cost models, but considers also some special properties of the JEPC middleware. The function *Size* gives the estimated size of every event stream that is involved in a multiway correlation:

$$Size(E^L) := \begin{cases} Sel(\varphi) * Size(E_1^L) * Size(E_2^L) & \text{if } E^L = \bowtie_\varphi (E_1^L, E_2^L) \\ AvgValidEvents(E^L) & \text{otherwise} \end{cases}$$

In case of an input event stream that already exists, *Size* returns the average number of valid events at a point in time. This measure corresponds approximately to the average number of events within the state of an correlation EPA on the event stream. The value returned by *AvgValidEvents* is obtained by analyzing the history of the given event stream. In case of an intermediate event stream, we have to estimate its average size because it is not an existing event stream whose history can be analyzed. Therefore, *Size* returns the size of the Cartesian product of the incoming event streams multiplied with the selectivity of the correlation condition.

The final cost function $Cost_{out}$ we use for scoring trees of correlation EPAs is based on the well-known cost function C_{out} [CM95] that is simple but effective:

$$Cost_{out}(E^L) := \begin{cases} Size(E^L) + Cost_{out}(E_1^L) + Cost_{out}(E_2^L) & \text{if } E^L = \bowtie_{\varphi} (E_1^L, E_2^L) \\ 0 & \text{otherwise} \end{cases}$$

This cost function is symmetric and fulfills the ASI property [MS79] that guarantees the optimal solution (according to the cost function) to be computable in polynomial time [AK80, CM95, IK84, KBZ86]. Furthermore and in contrast to many proposed cost models for join ordering, it does not assume a disk-based access model. This is important because we use it in the context of stream processing. Lastly, it captures the most important properties of the JEPC processing model (see Figure 19.1). It assigns no additional costs to the input event streams, because all input event streams must be processed completely in every case. Each intermediate event stream is considered twice. This is a good estimation of the real costs in the special case of JEPC for the following reason. Assume two correlation EPAs in sequence. Then, the intermediate event stream produced by the first correlation EPA is send to the JEPC middleware. Because this event stream is handled by JEPC in the same way as an input event stream, it causes costs for the first time. JEPC sends back the entire intermediate event stream to the raw EP provider where it serves as input event stream of the second correlation EPA. Hence, the intermediate event stream causes costs for the second time.

We adopted three traditional join ordering algorithms [Moe09] for the JEPC query optimizer. Two of them are greedy algorithms that do not guarantee to find a good or even the optimal order, whereas the third algorithm is a dynamic programming approach that always finds the optimal order. The first greedy algorithm is quite simple and makes only partly use of the cost model. It is shown in Algorithm 30 and requires an array *inputStreams* containing all N input event streams as well as the correlation condition φ. At first, the input event streams are ordered according to their average sizes (line 1). Next, the resulting order is directly used to create a tree

Algorithm 30: CORRELATORORDERING1(*inputStreams*, φ)

```
Input: Array of Event Streams: $inputStreams$, Boolean Expression: $\varphi$
Output: EPN: $bestTree$
Data: Function: $Size$
1 $inputStreams \leftarrow$ SORT($inputStreams$, $Size$);
2 EPN $bestTree \leftarrow inputStreams[1]$;
3 for $i \leftarrow 2 \ldots N$ do
4     Boolean Expression $\varphi' \leftarrow$ GETSUBEXPRESSION($\varphi$, $bestTree$, $inputStreams[i]$);
5     $bestTree \leftarrow \bowtie_{\varphi'} (bestTree, inputStreams[i])$;
6 return $bestTree$;
```

(lines 2–5). GETSUBEXPRESSION gives all parts of the correlation condition that can be assigned to the correlation EPA currently being created. The first argument must be an entire correlation condition and the other two must be event streams. Then, GETSUBEXPRESSION returns a subexpression including all predicates that can be evaluated for the given event streams and that are not already assigned to a correlation EPA. Obviously, Algorithm 30 does not utilize the cost function $Cost_{out}$ and takes into account only input event streams but no intermediate event streams.

The second greedy algorithm, shown in Algorithm 31, improves the first one by actually using the cost function and by searching for an order in a significant larger search space (that still does not contain all possible orders). It starts with allocating an array that can store as many different candidate trees as there are input event streams (line 1). For each input event stream, it generates exactly one candidate tree which begins with that stream (lines 2–16). Therefore, every input event stream creates a new candidate tree (line 3). Then, an order for all remaining event streams is generated (lines 4–15). This is done by successively adding the input event stream that minimizes the size of the intermediate event stream resulting from the correlation between itself and the candidate tree generated so far. After all candidate trees have been generated, the cost function is used to determine the best one (lines 17–18).

The final algorithm is based on dynamic programming [Cor09]. With respect to the cost function, it finds always the optimal left-deep tree. This is, because a tree must have optimal subtrees for being optimal [GUW08, Moe09]. The correlation EPA ordering problem has optimal substructure therefore. Using dynamic programming, we can find those optimal subtrees in a bottom up fashion so that the optimal tree is generated at the end. Note that this is true for not only left-deep trees, but also all other types of trees. Moreover, the correlation EPA ordering problem has also over-

Algorithm 31: CorrelatorOrdering2(*inputStreams*, φ)

```
Input: Array of Event Streams: inputStreams, Boolean Expression: $\varphi$
Output: EPN: bestTree
Data: Function: Size, $Cost_{out}$
Array of EPNs candidates ← NewArray($N$);
for $i \leftarrow 1 \ldots n$ do
    EPN epn ← inputStreams[$i$];
    Set of Event Streams streams ← NewSet(inputStreams);
    streams.Remove(inputStreams[$i$]);
    while SizeOf(streams) $> 0$ do
        Event Stream $e' \leftarrow$ NULL;
        Boolean Expression $\varphi' \leftarrow$ TRUE;
        for $e^* \in$ streams do
            Boolean Expression $\varphi^* \leftarrow$ GetSubexpression($\varphi$,epn,$e^*$);
            if $e'$ = NULL or $Size(\bowtie_{\varphi^*} (epn, e^*)) < Size(\bowtie_{\varphi'} (epn, e'))$ then
                $e' \leftarrow e^*$;
                $\varphi' \leftarrow \varphi^*$;
        streams.Remove($e'$);
        $epn \leftarrow \bowtie_{\varphi'} (epn, e')$;
    candidates[$i$] ← epn;
candidates ← Sort(candidates, $Cost_{out}$);
EPN bestTree ← candidates[1];
return bestTree;
```

lapping subproblems. Instead of enumerating and scoring all possible left-deep trees one another, we remember intermediate trees (which take part in multiple larger trees) and reuse them for generating and scoring larger trees. Algorithm 32 shows how the JEPC query optimizer finds the best left-deep tree via dynamic programming. It creates an array that is capable of storing the optimal tree for each subset (except the empty set) of the input event streams (line 1). We use a binary representation of the array index to encode the set of event streams for which the optimal tree is stored. The position of a bit corresponds to the position of an input event stream and its value indicates whether that stream is in the set. If the bit is set to one, then the associated event stream is included. For example, the array holds at position 21 (=10101_2) the optimal tree for the correlation of the first, the third and the fifth input event streams. The algorithm uses GetBit for looking up the bit at the position given as argument

Algorithm 32: CORRELATORORDERING3(*inputStreams*, φ)

```
Input: Array of Event Streams: inputStreams, Boolean Expression: $\varphi$
Output: EPN: bestTree
Data: Function: $Cost_{out}$
1  Array of EPNs trees ← NEWARRAY[$2^N - 1$];
2  for $i \leftarrow 0 \dots N-1$ do
3      trees[$2^i$] ← inputStreams[$i+1$];
4  for $b \leftarrow 1 \dots 2^N - 1$ do
5      if trees[b] = NULL then
6          for $i \leftarrow 1 \dots N$ do
7              if b.GETBIT(i) = 1 then
8                  b.FLIPBIT(i);
9                  Boolean Expression $\varphi'$ ←
                   GETSUBEXPRESSION($\varphi$,trees[b],inputStreams[i]);
10                 EPN epn ← $\bowtie_{\varphi'}$ (trees[b], inputStreams[i]);
11                 b.FLIPBIT(i);
12                 if trees[b] = NULL or $Cost_{out}$(trees[b]) > $Cost_{out}$(epn) then
13                     trees[b] ← epn;
14 EPN bestTree ← trees[$2^N - 1$];
15 return bestTree;
```

and FLIPBIT for flipping the bit at the position given as argument. Initially, the array contains only NULL values. For each subset of input event streams that contains only a single input event stream, the optimal tree is given by the only existing input event stream (lines 2–3). Between the array positions 2^j and 2^{j+1}, the $(j+1)$-th bit is always one. Therefore, the trees at those positions contain always the $(j+1)$-th input event stream and a subset of $\{E_1^L, E_2^L, \dots, E_j^L\}$. All empty positions in the array are filled from left to right (lines 4–13). As a consequence, the optimal trees for the first $k+1$ input event streams are computed after the optimal trees for the k input event streams have been determined (i.e., trees are generated in a bottom-up fashion). This allows to reuse the optimal solutions for the first k input event streams to determine the optimal trees for the first $k+1$ input event streams (lines 6–13). Two alternative trees for the same set of input event streams are compared on basis of the cost function (lines 12–13). Once all optimal trees are finally computed, the last position in the array contains the optimal tree for all input event streams (all bits are one).

19.2.4 Optimization of Pattern Matcher EPAs

We already mentioned and exploited the fact that a pattern matching EPA can be seen as a collection of filters (each individual symbol definition introduces a new filter). Therefore, a pattern matching EPA is more costly than a single filter EPA in general. Due to its cheapness, it is almost always a good idea to push down a filter EPA in front of a more costly EPA such as an aggregation or correlation EPA. Then, the filter EPA reduces the number of events that enter the succeeding costly EPA. In case of a pattern matching EPA, the same positive effect is expected if we can successfully move a filter EPA in front of it. Fortunately, pattern matching EPAs inherently contain filter conditions and allow for extraction of filter conditions therefore.

```
(SELECT  *
 FROM    RunnerLocation
 MATCH_RECOGNIZE_SEQUENTIAL (
 PATTERN  ABCD
 DEFINE    A AS checkpoint = "Staten Island",
           B AS checkpoint = "Queens",
           C AS checkpoint = "Bronx",
           D AS checkpoint = "Manhatten"
 WITHIN    5 HOURS
) AS RunnerTrackingQuery
```

Listing 19.1: Runner tracking query

The pattern matching EPA shown in Listing 19.1 is adopted from [WTA10]. It is a good example to explain our general idea. This EPA checks whether a runner has legally finished a race through New York City. Every time the monitored runner reaches a checkpoint, an event of type `RunnerLocation` is emitted. Such an event contains the name of the checkpoint. A runner must pass through all checkpoints in the correct order to legally finish the race. Then (and only then) the query produces an output event. In this particular example, every symbol is defined by a Boolean expression that only refers to event attributes. Therefore, it could also be evaluated by a filter EPA in front of the pattern matching EPA. Note that symbol definitions that refer to global variables and definitions of symbols which must not necessarily be emitted cannot be extracted. However, there arise several questions and problems. First of all, we need a strategy to decide deterministically which symbol definition should be extracted in case of multiple options. The JEPC query optimizer always chooses the symbol definition with lowest selectivity. This ensures that the number of events entering a pattern matching EPA is minimized. In case of the example query, the symbol

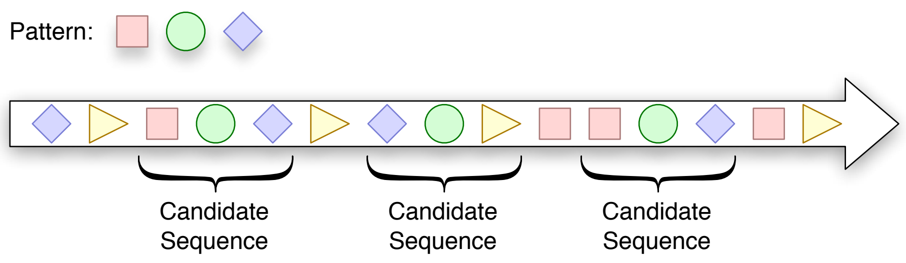


Figure 19.2: Selecting potentially matching event sequences

with lowest selectivity is `D`, because the last checkpoint is most difficult to reach. Furthermore, we need a transformation rule for query plans that does not affect query results. If we simply push down the chosen symbol definition as in the case of correlation EPAs (i.e., the chosen symbol definition is removed from the pattern matching EPA), then the pattern will break. Therefore, a copy of the chosen symbol definition must be used as filter condition for a preceding filter EPA in order to preserve the pattern to detect. Last but not least, selecting only single events affects query results because of the following reason. Suppose a filter EPA that selects only events which fulfill symbol definition `D` is placed in front of the pattern matching EPA. Then, no events that emit one of the symbols `A`, `B` and `C` will enter the pattern matching EPA. Obviously, the output event stream will be empty in every case.

The JEPC query optimizer solves the last problem by using a modified version of the filter EPA. Actually, the extracted filter condition points to an entire sequence of events that potentially matches the pattern rather than to only a single event. Figure 19.2 illustrates this issue. It shows a stream of events each fulfilling exactly one of the definitions of the symbols *circle*, *box*, *diamond*, and *triangle*. The pattern to detect is "*diamond* followed by *circle* followed by *box*". Let the definition of the symbol *circle* be the most selective one (this symbol occurs with lower frequency than the others in the figure). Then, this symbol definition is used as filter condition for a filter EPA being placed in front of the pattern matching EPA. Applied to the shown event stream, this filter EPA selects a *circle* event three times. Because such an event must occur in every matching event sequence, those are the locations of potentially matching sequences. However, not only the selected *circle* events must be forwarded to the pattern matching EPA, but also a certain amount of preceding and succeeding events. Because the pattern to detect has a fixed size of three events and *circle* is the second symbol that must be emitted, a candidate sequence consists of a *circle* event and its adjacent events. Each candidate sequence must be processed by the pattern matching EPA in order to decide whether a candidate sequence is a matching sequence. In the figure, the first and third candidate sequences are matching sequences, whereas the second is not. All events that are not part of a candidate sequence cannot be part of a matching sequence. They can be safely prevented from entering a pattern matching EPA. Therefore, the JEPC query optimizer implements the following novel rule:

$$\rho_P(E^L) = \rho_P(\sigma^*_\varphi(E^L))$$

This rule can be applied if the pattern being detected contains a symbol definition $\varphi \in P$ which can be extracted. Then, an extended filter EPA σ^* can be placed in front of the pattern matching EPA. An extended filter EPA selects not only events that fulfill φ, but also all other events of their corresponding candidate sequences.

Our basic idea to optimize pattern matching EPAs by filtering their input event streams is inspired by the system Z-Stream [MM09]. This system is primarily for detecting sequential event patterns. Instead of using an NFA, which forces a fixed evaluation order of symbol definitions according to the occurrence order within the pattern to detect, Z-Stream utilizes tree-based operator plans that allow for different evaluation orders. Z-Stream exploits this property by evaluating symbol definitions with low selectivity before symbol definitions with high selectivity. However, this method is only applicable in the special case of Z-Stream. It is no universal optimization technique. In particular, it cannot be adopted by NFA-based approaches to pattern matching. On basis of Z-Stream, we developed a universal optimization technique of pattern matching queries that is independent of the implementation of the pattern matching EPA. Furthermore, Z-Stream is only able to speed up single pattern matching queries. In contrast, our technique can be extended to speed up a set of pattern matching queries over the same event stream. A set of pattern matching EPAs may have a common symbol definition that can be extracted. If so, we can extract this common symbol definition and use it to configure an extended filter EPA that is responsible for all pattern matching EPAs at the same time, instead of optimizing each single EPA of the set by extracting its most selective symbol definition. More formally:

$$\{\rho_{P_i}\}(E^L) = \{\rho_{P_i}\}(\sigma^*_\varphi(E^L))$$

The above rule assumes that φ is a symbol definition that can be extracted from all patterns P_i of the set. Then, the event stream which is consumed by all pattern matching EPAs can be filtered by using only a single extended filter EPA. Because the lengths of the patterns P_i as well as the positions of the symbols defined by φ within the patterns may differ, candidate sequences selected by the extended filter EPA must be chosen such that they are candidate sequences for all patterns P_i.

19.3 Implementation

In today's database systems, the query optimizer is an integral part of the query compiler and cannot be disabled or arbitrarily configured in most cases. We decided to not make the query optimizer a permanent feature of the JEPC core. Instead, the JEPC query optimizer was implemented as an extension that may or may not be used. The core of JEPC executes query plans without performing any modifications, which gives users and extensions full control over query execution. In addition, the JEPC query optimizer allows to specify in detail which optimizations to apply to a certain query.

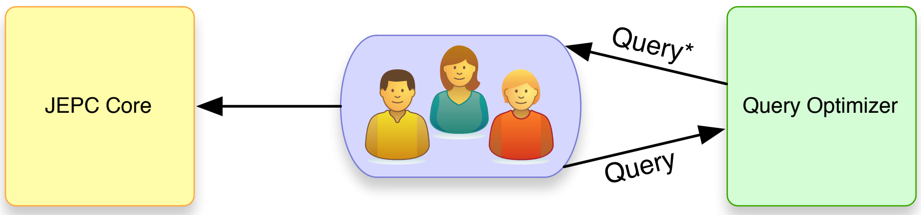


Figure 19.3: Query optimizer as JEPC extension

Figure 19.3 illustrates the implementation of the query optimizer as an extension. The JEPC core executes query plans exactly as they are defined by users and extensions. But users and extensions are free to send a new query plan *Query* to the query optimizer and to execute the returned optimized query plan *Query** instead.

Continuous queries over data streams with potentially time-varying characteristics require not only one-time optimization before execution, but also continuous optimization during execution. Besides that, some of the presented optimization techniques need to know selectivities (e.g., of single predicates or of complete Boolean expressions) or to integrate extended filter EPAs into query plans for the optimization of pattern matching EPAs. We first describe how we support continuous optimization of queries via query monitors. Then, we introduce extended filter EPAs.

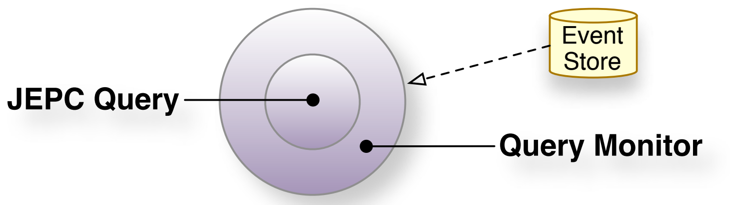


Figure 19.4: Query monitor

If a user wants a JEPC query to be optimized continuously, the JEPC query optimizer returns a one-time optimized query plan wrapped in a so-called *query monitor* as shown in Figure 19.4. While the query monitor is executed within the JEPC middleware, the query plan is translated and executed by the underlying raw EP provider as usual. Thus, JEPC queries with query monitor are still independent of the used raw EP provider. A query monitor sees every event that enters or leaves an EPA of its associated query plan. This allows to continuously determine the current selectivities. Of course, the monitoring imposes little runtime overhead. To reduce the overhead, a query monitor provides two options. First, the determination of selectivities can be done only for sections of event streams. This means that a query monitor updates all selectivities by processing a certain amount of events and shuts down afterwards. It then sleeps for a user-defined period of time. Second, a query monitor must not determine selectivities by itself. If available, it can simply query the event store (see Chapter 17). This option obviously requires an event store to be present, but leads to almost no overhead. If selectivities have changed so that another equivalent query plan is better, the query plan in execution must be modified. A query monitor modifies its query plan on-the-fly via our query update method (see Chapter 18).

An extended filter EPA is integrated into query plans in the form of a user-defined EPA that runs in the middleware layer (see Section 7.3). It primarily looks for incoming events that fulfill its filter condition. Since such an event is just one element in a candidate sequence, a certain amount of preceding events must be buffered and a certain amount of succeeding events must be forwarded unconditionally. For a single pattern matching EPA, the exact sizes of those amounts depend on the pattern. The symbol the extended filter EPA is looking for (pivot symbol) divides the pattern into two parts. The first part begins at the first symbol and extends to the pivot symbol, whereas the second part begins at the pivot symbol and extends to the last symbol. We only discuss how to determine the amount of events for the first part. The second part is handled totally analogous. We need to determine either the exact number of events or the maximum range of time the first part covers. If the first part of the pattern has a fixed length (i.e., no Kleene operators are applied), the exact number of events to buffer can be determined simply by counting the symbols of the first part. Then, the buffer behaves like a count-based sliding window keeping always an exact number of events. But if the first part of the pattern has variable length, the buffer must behave like a time-based sliding window with size set to the size of the within-condition of the pattern. To preserve pattern matching with low latency, an extended filter EPA immediately forwards events when the pivot symbol is detected. At first, it releases the buffer containing all relevant events preceding the pivot symbol. Then, it forwards the event that is associated with the pivot symbol. Lastly, it directly forwards new events until a complete candidate sequence has been passed through. If an extended filter EPA is responsible for multiple pattern matching EPAs at the same time, the first part with maximum size among all patterns determines the size of the buffer and the second part with maximum size among all patterns determines the amount of events that must be forwarded after the pivot symbol. In the case that an event store is available, buffering of events is not necessary. After the pivot symbol has been detected, all preceding events of the candidate sequence can be obtained from the event store. This is particularly important in case of very large within-conditions and variable length patterns. Because pattern matchers are usually implemented in the form of NFAs, they do not keep events. Therefore, within-conditions can cover amounts of events that are significantly larger than the available main memory.

The extended filter EPA passes through candidate sequences one another and discards all events in-between. This leads to a serious problem. The end of a candidate sequence may start new matching instances that are successfully finished by the beginning of the next candidate sequence. To avoid false positive matches, the extended filter EPA outputs dummy events, which reset all active matching instances, between two candidate sequences that do neither overlap nor meet.

19.4 Evaluation

In an experimental evaluation we examined the effects of the implemented optimization techniques that are applied purely in the middleware layer. The experiments were conducted on a single machine with an Intel i7-2600 CPU and 8 GiB main memory running JEPC in an Oracle Java HotSpot VM (1.8.0_25). Furthermore, the JEPC bridge to the SPE Esper (4.9.0) was used in all experiments.

19.4.1 Predicate Ordering

At first, we present the evaluation of the technique that optimizes the order of predicates within Boolean expressions. The setup consisted of a single filter EPA that consumed an event stream with 100 attributes each of type 32-bit integer number. Different total numbers of predicates were combined via conjunctions into Boolean expressions which were used by the EPA as filter conditions. Each predicate referred to two different attributes and had a different pre-defined selectivity. Within the generated Boolean expressions, all predicates were ordered by selectivity from high to low. This order of predicates is the worst case from a performance point of view. The event stream was generated in a way that kept the chosen selectivities of predicates constant. We studied three different configurations of this experiment. In the first configuration, we directly executed the EPA using the worst case order of predicates (Non-Optimized). In the second configuration, we let the JEPC query optimizer rearrange the predicates and apply a query monitor that permanently determined the current selectivity of each predicate (Optimized + Monitor). This configuration was for identifying the runtime overhead imposed by an active query monitor that determines selectivities by itself. However, a query monitor determines selectivities only for sections of the event stream and is inactive for most of the time in practice. Therefore, we shut down the query monitor in the third configuration (Optimized).

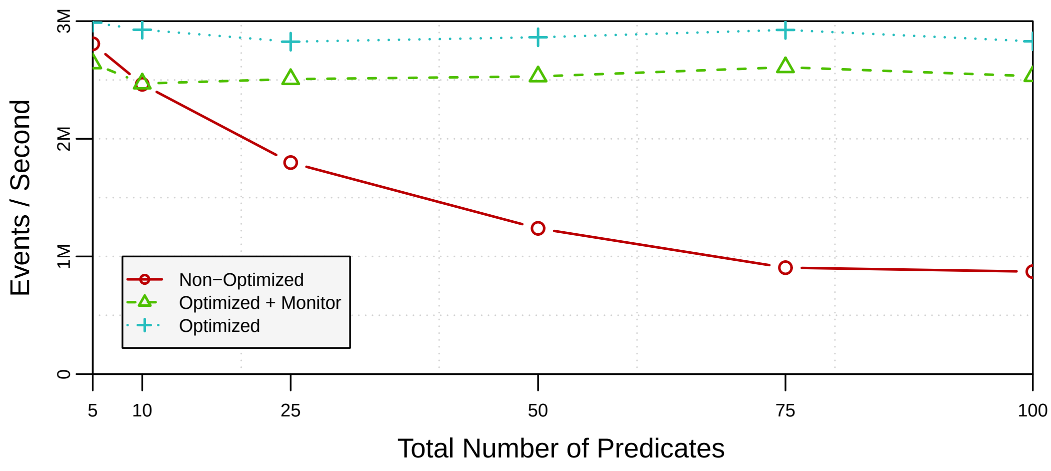


Figure 19.5: Effect of predicate ordering

Figure 19.5 shows the achieved throughputs for different total numbers of predicates. As the number of predicates increased, the throughput of the non-optimized filter EPA decreased because of an increasing number of unnecessary predicate evaluations. The configuration with an optimized predicate order and a permanently active query monitor was nearly not affected by an increasing number of predicates. When the query monitor was shut down, the throughput could be further increased by a constant factor. The difference of the two graphs (Optimized and Optimized + Monitor) gives the overhead that was caused by an active query monitor. Our results show that even for few predicates the additional overhead could be compensated by the effect of optimization. Note that filter EPAs are relatively cheap in contrast to the other types of EPAs. Therefore, the overhead imposed by active query monitors is comparatively higher in case of filter EPAs than in case of the other EPA types.

19.4.2 Filter Push-Down

The JEPC query optimizer provides at least one filter push-down technique for each basic type of EPA. Since most filter push-down techniques are applicable with no and the remaining filter push-down techniques with only little limitations, they play an important role in real-world JEPC applications in general. In the experiments presented in the following, we examined the effects of the filter push-down techniques for filter, correlation, aggregation and pattern matching EPAs.

In case of filter EPAs, the corresponding filter push-down technique transforms EPNs that contain multiple filter EPAs in a row. For studying the effect of this transformation, we executed a query consisting of two filter EPAs σ_{φ_1} and σ_{φ_2}. Both filter EPAs were combined into the EPN $\sigma_{\varphi_2}(\sigma_{\varphi_1}(E^L))$. The event steam E^L consisted of events with six randomly generated 32-bit integer numbers as payload. We ran this experiment several times for different selectivities of the filter conditions φ_1 and φ_2.

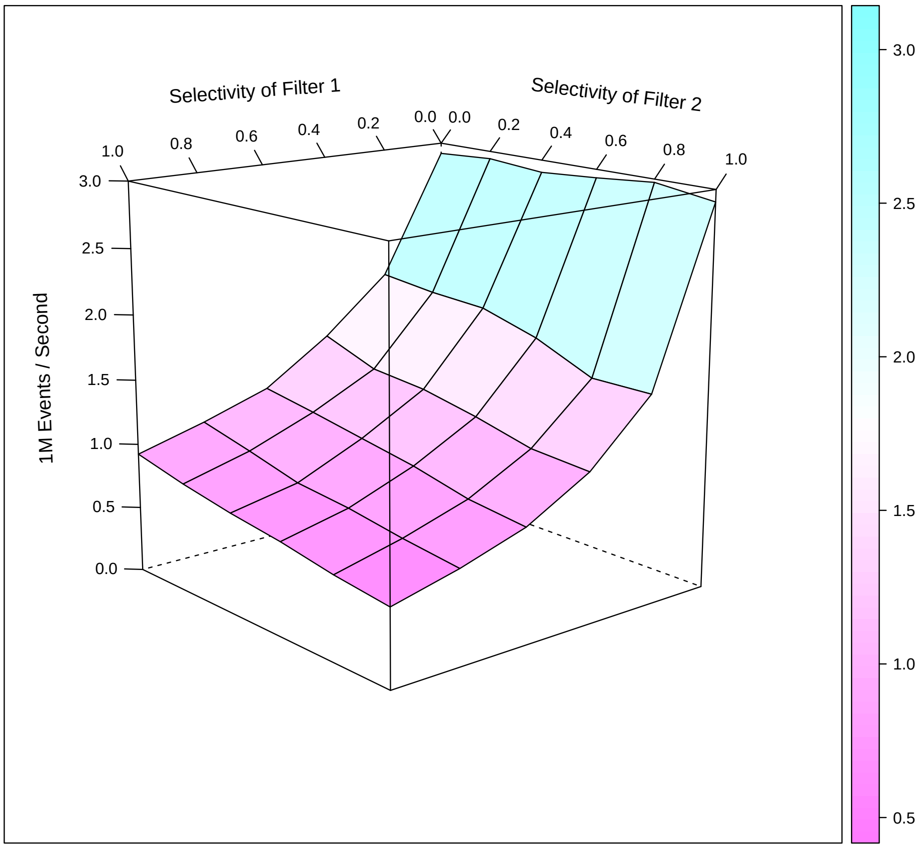


Figure 19.6: Effect of filter push-down on filter EPA (non-optimized)

Figure 19.6 shows the achieved throughputs (z-axis) of all tested configurations (x-axis and y-axis respectively). The overall throughput was heavily affected by the selectivity of the filter condition of the first EPA and only slightly by the selectivity of the filter condition of the second. This was because the more events had passed through the first EPA, the more events had to be processed by the second.

In another setup of the experiment, the JEPC query optimizer transformed every EPN before its execution. After its transformation, each optimized test query consisted of only a single filter EPA with $\varphi_1 \wedge \varphi_2$ as filter condition. Figure 19.7 shows the results of the experiment using the new setup. Except the cases where the selectivity of the first filter condition was zero, the overall throughput was significantly higher and did not decrease so heavily as in the setup with non-optimized test queries. Note that the impacts of both filter conditions were nearly identical to each other in this setup (the graph in Figure 19.7 is approximately symmetric).

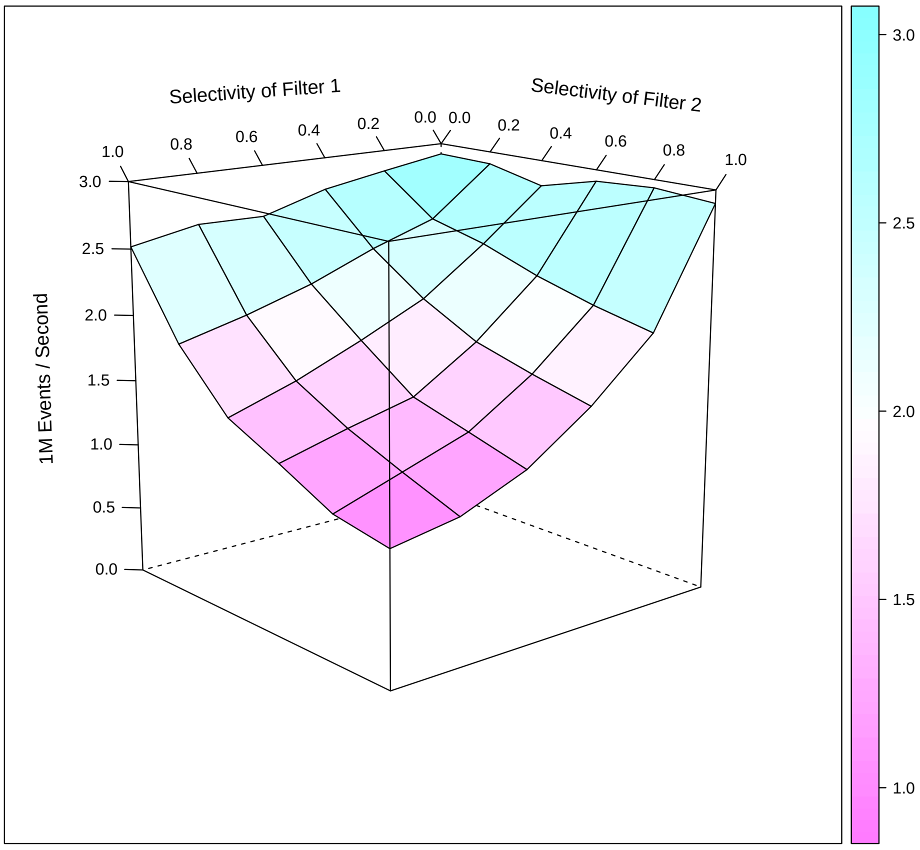


Figure 19.7: Effect of filter push-down on filter EPA (optimized)

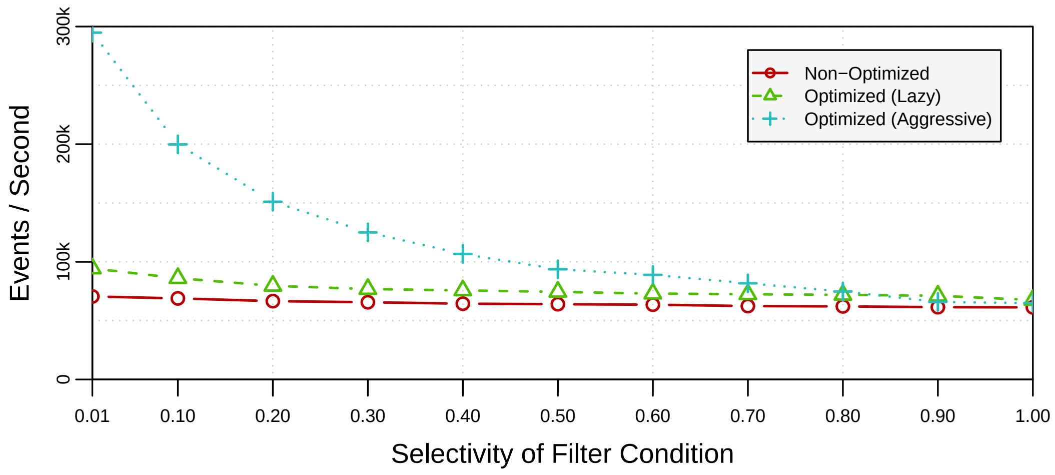


Figure 19.8: Effect of filter push-down on correlation EPA

In our next experiments, we focused on the filter push-down techniques for correlation EPAs. Here, we had to investigate two different applications of it. First, a filter EPA directly succeeding a correlation EPA is completely pushed down and, thus, eliminated. Second, all predicates of a correlation condition that only refer to one input event stream are extracted and evaluated by an additional filter EPA in front of the correlation EPA. The test query used in this experiment was the EPN $\sigma_{\varphi_f}(\bowtie_{\varphi_c} (\omega_{100}^{count}(E_1^L), \omega_{100}^{count}(E_2^L)))$. Each event of the event streams E_1^L and E_2^L had three randomly chosen 32-bit integer numbers as payload. The correlation condition φ_c referred to attributes of both event streams and had a fixed selectivity of 0.01. In contrast, the filter condition φ_f referred only to attributes of E_1 and we tested different selectivities of it. In lazy mode, the JEPC query optimizer integrated the filter condition into the correlation EPA so that the transformed EPN was $\bowtie_{\varphi_c \wedge \varphi_f} (\omega_{100}^{count}(E_1^L), \omega_{100}^{count}(E_2^L))$. And in aggressive mode, the JEPC query optimizer further transformed the EPN. It analyzed the new correlation condition $\varphi_c \wedge \varphi_f$ and further pushed φ_f so that the transformed EPN was $\bowtie_{\varphi_c} (\sigma_{\varphi_f}(\omega_{100}^{count}(E_1^L)), \omega_{100}^{count}(E_2^L))$.

Figure 19.8 shows the achieved throughputs as a function of the selectivity of φ_f. The lazy optimized query performed better than the non-optimized query. However, the effect disappeared for extremely high selectivities of φ_f. For low selectivities of φ_f, the aggressively optimized query achieved throughputs that were higher by multiple times than the achieved throughputs of the other queries. This was due to the fact that the filter EPA prevented most events from entering the costly correlation EPA. Again, its effect also disappeared for extremely high selectivities of φ_f.

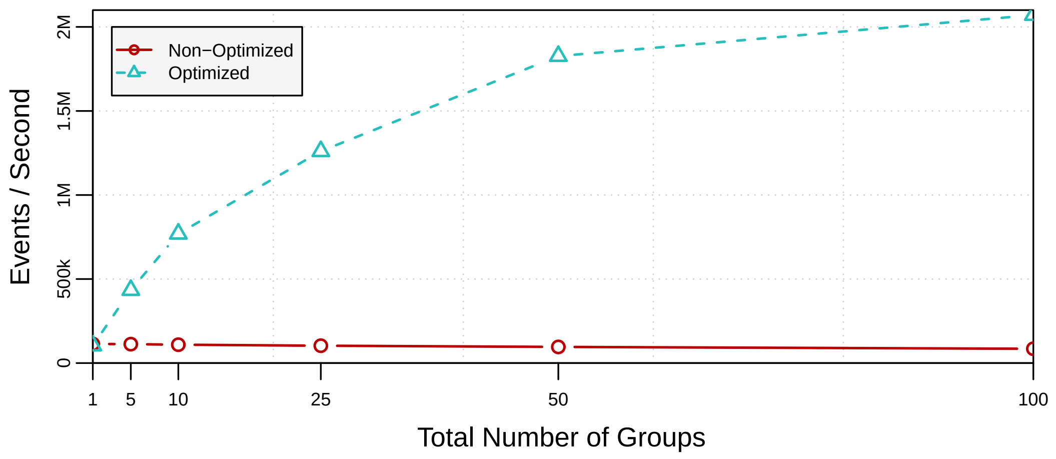


Figure 19.9: Effect of filter push-down on aggregation EPA

For examining the filter push-down technique for aggregation EPAs, we used the EPN $\sigma_{groupID=1}(\alpha_{groupID,max(attr)}(\omega_x^{time}(E^L)))$ as test query over a stream E^L of events that had six 32-bit integer numbers as payload. Five of them (including *attr*) were randomly generated. The values of the attribute *groupID* were not random. This attribute was used to identify the group an event belonged to. At first, the test query computed the maximum value of the attribute *attr* per group within a sliding time window. After the aggregation, a filter EPA selected only the results that belonged to the first group. As a consequence, the selectivity of the filter EPA depended on the total number of groups. We generated several input event streams with different numbers of equally sized groups. In particular, each event stream was generated so that for each group there were constantly 100 events within the sliding time window.

Figure 19.9 shows the achieved throughput of the test query for different numbers of groups (non-optimized). Since event aggregation is a more expensive operation than event filtering, the overall costs were dominated by the costs of the aggregation EPA. At a first glance, the throughput of the aggregation EPA was approximately constant, because every input event led to the computation of a *max* aggregate for a state consisting of exactly 100 events in our experimental setup. But on closer inspection, the throughput decreased slightly with an increasing number of groups (from one group to 100 groups the performance drop was 26 %). This was due to the overhead caused by grouping. Because of its placement, the succeeding filter EPA whose filter condition became more selective when the number of groups was increased could not compensate the overhead. This changed after we let the JEPC query optimizer transform the test query before its execution. Then, the filter EPA was moved to the front of the aggregation EPA (optimized). Obviously, this transformation had no effect for a single active group. The selectivity of the filter EPA was 1 in this case. However, when the number of active groups was increased also the throughput increased significantly, because more and more events were prevented from entering the costly aggregation EPA. As a pleasant side effect, there was no overhead imposed by grouping. The aggregation EPA had to manage only a single group in every case.

For pattern matching EPAs, there are three different applications of the filter push-down technique. First, a succeeding filter EPA is integrated into the symbol condition of the last symbol of a pattern (lazy mode). Second, a filter condition is further pushed down within a pattern (aggressive mode). Third, a filter condition that already was pushed down within a pattern definition is extracted and evaluated before pattern matching. The optimization technique that extracts filter conditions is studied in Section 19.4.4. In our next experiment, we examined only the first two applications. The EPN $\sigma_{\varphi_f}(\rho_P)(E^L)$ served as test query in this experiment. Furthermore, the event stream E^L consisted of events with six randomly chosen 32-bit integer numbers as payload and the pattern to detect was defined as follows:

$$P := a[\varphi_a]b[\varphi_b]c[\varphi_c]d[\varphi_d]e[\varphi_e]$$

The symbol conditions φ_a and φ_b had a fixed selectivity of 1 and all others had a fixed selectivity of 0.5. Every symbol definition also included the setting of exactly one global variable. Symbol a set the only global variable used in the filter condition φ_f. Thus, the filter condition φ_f could be pushed down at most to symbol b.

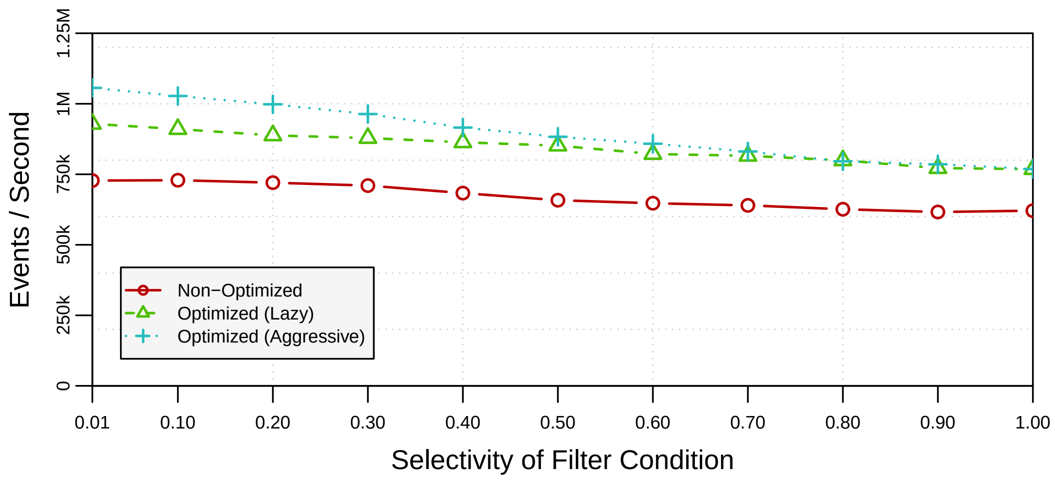


Figure 19.10: Effect of filter push-down on pattern matching EPA

Figure 19.10 presents the achieved throughputs for different selectivities of the filter condition φ_f. In case of the optimized test queries, the maximum event throughput was higher in comparison to the original test query. This was mostly due to the elimination of the filter EPA in the EPNs. For low selectivities, the aggressive optimization that pushed down the filter condition to symbol b was superior to the lazy optimization that pushed down the filter condition only to symbol e. This confirms that a pattern matching EPA can be seen as a series of filters. Therefore, filter conditions should be pushed down as far as possible in order to maximize the performance.

Altogether, the filter push-down techniques should be applied in every case, because they never had a negative effect on the performance. On the contrary, they improve the overall query performance significantly in general. And in the worst case, they simply have no effect. Note that the worst case scenarios require extremely high selectivities. Because filter conditions with selectivities close to 1 contradict the purpose of event filtering, it can be expected that the filter push-down techniques will speed-up query execution in almost all real-world applications.

19.4.3 Correlation EPA Ordering

In order to study the effects of the algorithms for correlation EPA ordering, we generated EPNs in the form of left-deep trees consisting of $N-1$ correlation EPAs for different numbers N of input event streams. Each input event stream contained events with three randomly chosen 32-bit integer numbers as payload. All correlation EPAs of a left-deep tree except the topmost one, which kept the entire correlation condition, had `TRUE` as correlation condition. Note that joins are commonly expressed this way in standard SQL and, thus, our test queries represented the naïve translation of such query specifications. The correlation condition of each test query consisted of $2N$ predicates that were connected via conjunctions. Each of the predicates referred to two randomly chosen attributes and also the type of a predicate was randomly chosen. Generated correlation conditions that always evaluate to `FALSE` were discarded and replaced by a completely new generated correlation condition. On average, every correlation EPA had 70 events of each input event stream in its state.

Figure 19.11 shows the achieved throughputs of the test queries and their optimized versions for different numbers N on the x-axis. The results clearly show that each ordering algorithm of the JEPC query optimizer improved the performance noticeably. In every case, Greedy2 was superior to Greedy1. As expected, the dynamic programming approach always outperformed the two greedy algorithms.

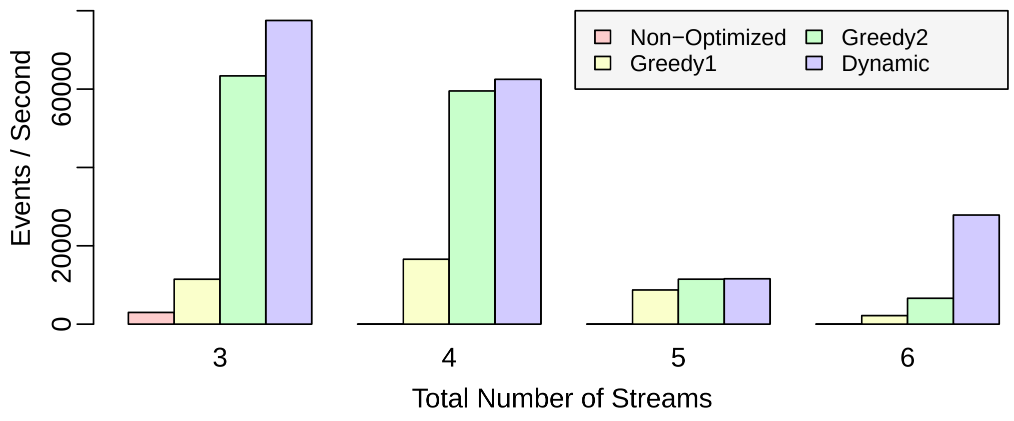


Figure 19.11: Effect of correlation EPA ordering

19.4.4 Optimization of Pattern Matching EPAs

In this final section, we present experiments which we conducted to examine the optimization technique for pattern matching EPAs. We applied one or more pattern matching EPAs on an event stream that contained exactly one event per time instant in this experiments. Each single event had six randomly chosen 32-bit integer numbers as payload. At first, we investigated the optimization technique for a single pattern matching EPA. For this purpose, exactly one pattern matching EPA was executed in each run of the experiment. The pattern to detect was defined as follows:

$$P := a[\varphi_a]b[\varphi_b]c[\varphi_c]d[\varphi_d]e[\varphi_e]$$

All symbol conditions except φ_c had a fixed selectivity of 0.5. For the symbol condition φ_c, which could be extracted by the JEPC query optimizer, we tested different selectivities ranging from 0 to 1. Furthermore, every symbol definition included the assignment of global variables. The within condition was set to 100 time instants.

Figure 19.12 shows the achieved throughputs of the non-optimized and the optimized query plans for different selectivities of φ_c. In case of low selectivities which are the rule rather than the exception in meaningful pattern definitions, the optimization technique was able to speed up the pattern matching query by multiple times. However, the performance was slightly worse for high selectivities. Beginning from a selectivity of about 0.3, the optimized query plan achieved a throughput that was lower by a few percent in comparison to the non-optimized query plan. This was due to the fact that a large amount of events passed through the additional filter EPA in front of the pattern matching EPA. The additional filter EPA could not discard sufficiently many events to compensate the overhead imposed by it.

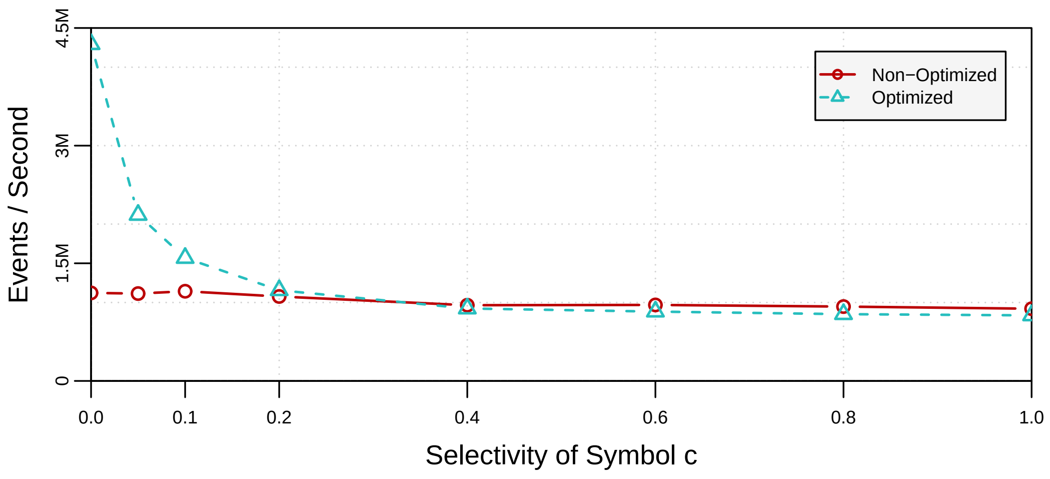


Figure 19.12: Effect of filter extraction on a single pattern matching EPA

In our last experiment, we studied the effect of the optimization technique for a set of pattern matching EPAs. Therefore, the query workload of the following experiment consisted of multiple pattern matching EPAs that had a symbol condition in common. The pattern of the i-th pattern matching EPA of N pattern matching EPAs in total was defined as follows:

$$P_i := a[\varphi_a^i]b[\varphi_b^i]c[\varphi_c^i]d[\varphi_d^i]e[\varphi_e^i]$$

All symbol conditions except φ_c^i were randomly generated, had a randomly chosen selectivity in $[0.5 : 1.0]$, and referred to global variables that were selected by random. The symbol condition φ_c^i was fixed in all patterns, had a fixed selectivity of 0.01, and did not refer to global variables. In other words, each pair of pattern matching EPAs had pattern definitions in which the conditions of symbol c were identical while all other symbol conditions differed from each other. Furthermore, every symbol definition also included the assignment of exactly one global variable. The within-condition of each pattern matching EPA was set to 100 time instants. According to this specification of the query workload, the JEPC query optimizer was able to extract the condition of symbol c for every generated set of pattern matching EPAs.

Figure 19.13 shows the achieved throughputs for different numbers of pattern matching EPAs on the x-axis. At a first glance, it is obvious that the application of the optimization technique for multiple pattern matching EPAs had a strong effect. When the total number of active pattern matching EPAs was increased, the throughput decreased in both cases. However, the performance drop of the optimized query workload was significantly lower. For example, the maximum event throughput of the

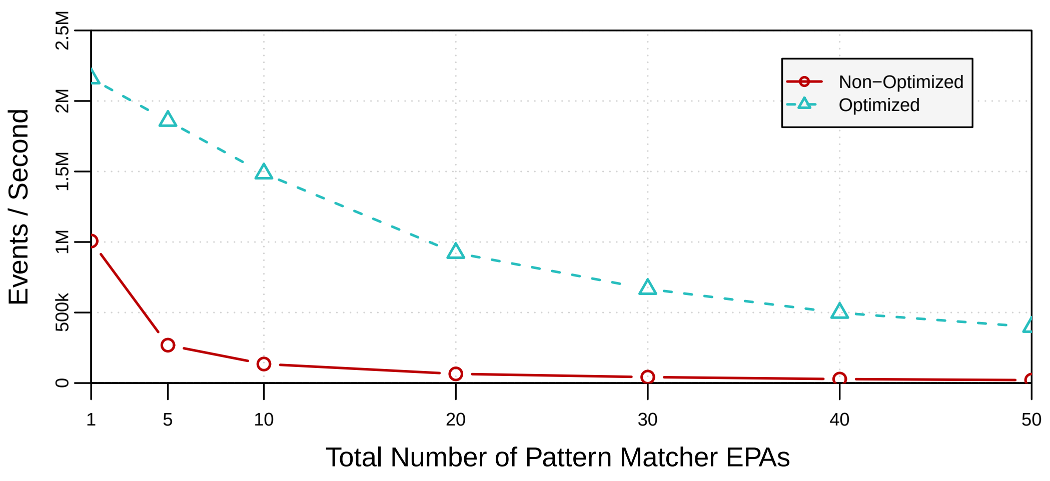


Figure 19.13: Effect of filter extraction on a set of pattern matching EPAs

optimized query workload consisting of 50 active pattern matching EPAs was about 20 times higher than the maximum event throughput of the non-optimized query workload that also consisted of 50 active pattern matching EPAs. Note that the optimized query workload consisting of 20 active pattern matching EPAs achieved almost the same maximum event throughput as the non-optimized query workload consisting of only a single active pattern matching EPA.

Summary

This chapter presents a query optimizer for the JEPC middleware. The JEPC query optimizer is designed as an external adviser that may or may not be used. It supports not only one-time optimization before the execution of a query, but also continuous optimization during its execution, and takes over well-established optimization techniques from the area of database systems such as the powerful selection push-down and the ordering of multiway joins. In addition, it also implements novel optimization techniques for pattern matching queries. An experimental examination confirmed the positive effect of each implemented optimization technique.

20

Parallel and Federated Event Processing

Outline

20.1 Introduction

Generally speaking, data is processed in parallel mostly for one of two reasons. One reason is to improve the overall performance in order to handle large and huge workloads. And the other reason is achieving high availability by introducing redundancy. Unfortunately, both objectives of parallelization are contradictory. A parallel infrastructure designed for high performance has poor availability and a parallel infrastructure designed for high availability has poor performance. But recently, hybrid approaches such as are implemented by MapReduce [DG08, Läm07] and many NoSQL/NewSQL database systems [Gro13] became popular. Hybrid approaches are proven to achieve a reasonable trade-off between performance and availability.

In this chapter, we focus on increasing the overall performance of JEPC by processing events in parallel (high availability is out of the scope of this thesis). The key idea is to use multiple instances of EP providers (EPP) that run in parallel. This general concept can be further extended if we allow the instances to be of different types of EP providers. Such heterogenous configurations are called *federations* in the following. The advantage of a federation is that the different strengths and weaknesses of different types of EP providers can be exploited. Our experimental evaluation clearly shows that "one size fits all" [Sto07b, LHB13] is also not true for EP providers.

20.2 Parallelization of JEPC

We describe in Section 1.2 and illustrate in Figure 1.1 that the query workload of any stream processing system in general and of JEPC in particular can be modeled as a directed operator graph. In case of JEPC, the nodes of an operator graph are single EPAs and the directed edges represent the flow of events. Basically, there are three different methods to partition operator graphs and to distribute the resulting parts across a set of multiple workers running on, e.g., different machines connected to the same network or different cores of a multi-/many-core CPU [Gul10, Gul12]. First, subgraphs of an operator graph that represent entire queries are distributed across workers (query parallelization). Second, single operators of an operator graph (i.e., EPAs) are distributed across workers (operator parallelization). Third, arbitrary but disjoint subgraphs of an operator graph (i.e., EPNs that are not necessarily identical to entire queries) are distributed across workers (operator-set parallelization). The latter approach has been proven to be superior, because it is able to optimally balance the degree of parallelization and the communication overhead [Gul10, Gul12].

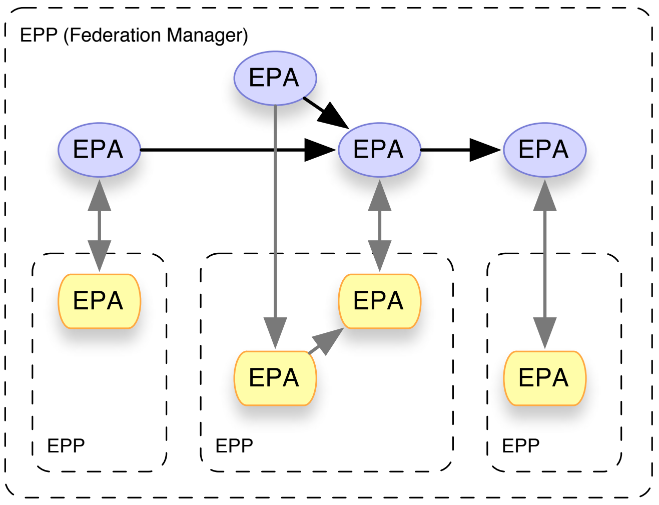


Figure 20.1: Parallel processing of EPAs using JEPC

We developed a so-called *federation manager* that automatically distributes queries according to the operator-set parallelization as a JEPC extension. At first, the best worker is determined for each EPA of a given operator graph. The federation manager assumes workers to be instances of EPPs that potentially have different performance profiles. Then, EPAs which are adjacent in the operator graph and assigned to the same EPP instance are merged into a subgraph to reduce communication overhead. The responsibility for each resulting partition is given to its assigned EPP instance. Figure 20.1 gives a high-level overview of the federation manager. The entire operator graph that shall be executed is illustrated in the upper part. Since the federation manager is not able to execute EPAs by itself, it delegates the execution to available instances of EPPs shown at the bottom. During execution, the federation manager redirects events and forwards output events to the outside world.

The federation manager has full control over multiple instances of ideally different types of EPPs that run somehow in parallel (e.g., on different CPU cores, on different machines, or on different components of a machine such as CPU and GPU). At the same time, the federation manager itself acts as an EPP. This leads to several advantages. Most importantly, the federation manager can be used like any other EPP, while the internal complexity of the federation is completely hidden from users and extensions (decorator design pattern [Fre04, Gam95]). Moreover, a federation manager can also use other federation managers as workers so that hierarchies of federations can be created. The only remarkable difference to other types of EPPs supported by JEPC is that events can be concurrently injected into the federation manager.

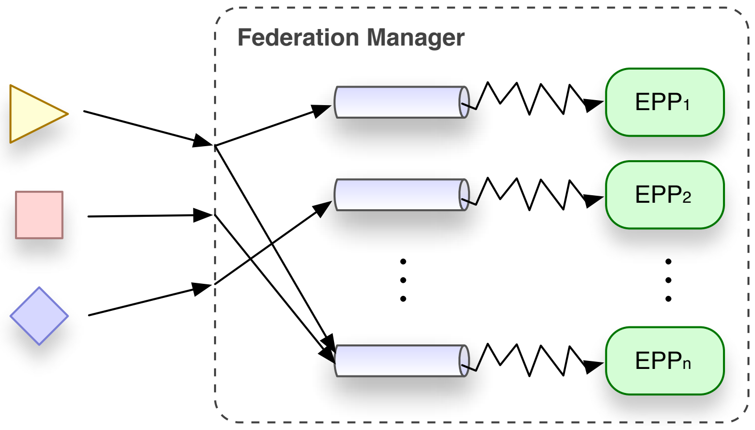


Figure 20.2: Overall architecture of the federation manager

Figure 20.2 shows the overall architecture of our implemented federation manager. All available instances of EP providers EPP_1 to EPP_n are executed in parallel. Each available instance has an assigned queue for transferring events. An event is allowed to arrive concurrently with other events. Every input event is added to the queues of all instances that execute partitions which require the event. At the other side of a queue, an extra thread pulls events out of it and pushes them into the corresponding EPP instance. This ensures that the EPP instances get events concurrently. The same threads are also used for receiving events from the instances. Output events are simply pushed back into the federation manager as new input events.

20.3 Distribution of EPAs

The main task of the federation manager is to distribute EPAs across a set of potentially heterogeneous instances of EPPs in a way that maximizes the overall event throughput. We define the optimal distribution as follows. First, every single EPA is executed by the type of EPP that is best suited from a performance point of view. Second, the load is balanced. Third, the communication overhead is minimal. Note that these three optimization goals contradict each other. The communication overhead is minimal if we place all EPAs on the same EPP instance. But then some EPAs are executed poorly and the EPP instance is overloaded. If we assign every EPA to the best suited EPP, the communication overhead is not minimal and the load is not balanced. And if we perfectly balance the load, EPAs are executed poorly and the communication overhead is not minimal. So, the federation manager is in charge to find a good trade-off between the three optimization goals.

The federation manager does not transform operator graphs. This is the task of a logical query optimizer which, of course, may be applied before (see Chapter 19). The federation manager performs physical query optimization by selecting a good implementation for each EPA of an operator graph. Different implementations to choose from are provided by different available types of EPPs. For the assignment of EPAs to EPP types that execute them efficiently, a method to determine the best type of EPP for a given EPA is needed. There are two different approaches common in physical query optimization [Moe09]. The performance a given EPA is expected to achieve when executed by a certain type of EPP can be either estimated or determined via a simulation. While the estimation approach is fast but can lead to supoptimal decisions, the simulation approach always converges to the optimal decision but is time consuming (especially in case of a large search space). However, both approaches can be combined [Moe09]. An estimation can be made to quickly find a good starting point. Through simulation this decision can be successively improved until it is optimal. In the case of the federation manager, we preferred to implement the estimation approach in order to be able to immediately start new CQs. Since CQs are long-term running, a simulation-based approach can be used in addition to successively improve the assignments of EPAs during runtime. The distribution of EPAs can be changed on-the-fly via dynamic plan migration [Krä06, Yan07, ZRH04] using the update method of JEPC (see Chapter 18) and is out of the scope of this chapter therefore.

20.3.1 Classification

We use a classifier in the form of an ordinary decision tree to assign EPAs to types of EPPs. The specific decision tree which is used by default (and which was also used in our experiments) is based on both expert knowledge and benchmarking. Because of our experience with JEPC, its bridges and the supported raw EP providers, we already had some knowledge about their individual strengths and weaknesses. We then performed a series of benchmarks in order to concretize our impressions and to potentially reveal hitherto unknown strengths and weaknesses.

In total, we performed more than 200 different benchmarks for four different types of EPPs. Those four types of EPPs were taken from the set of available JEPC bridges, but are not named because also commercial products were included. We simply refer to them as EPP_1, EPP_2, EPP_3 and EPP_4 in the following. Not all benchmarks led to new findings. But still, it is not possible to report the results of all relevant benchmarks in their full breadth. Instead, we present a simplified version of our final decision tree that is an aggregation of the benchmarking results. There were two important criteria

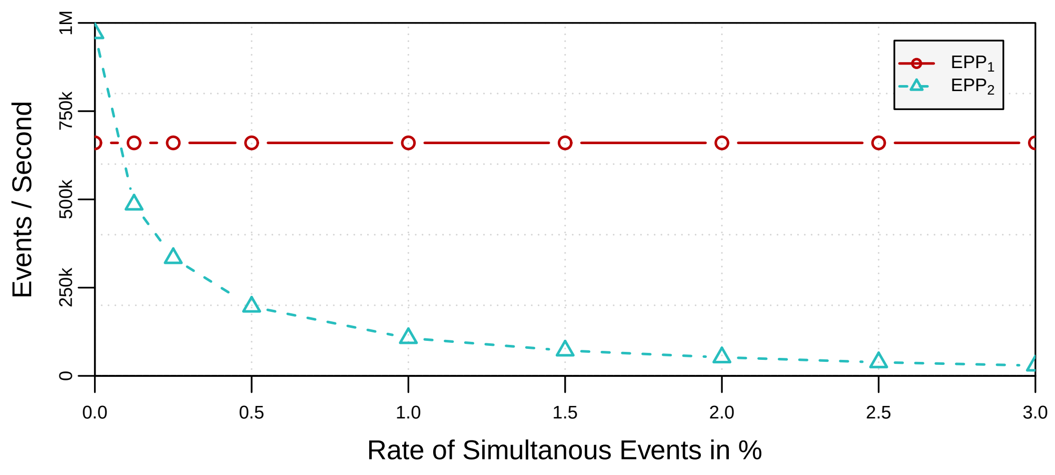


Figure 20.3: Effect of simultaneous events on pattern matchers of different EPPs

that mainly separated the tested types of EPPs. First, the size of an operator state was an influencing factor. This was because different raw EP providers contain different algorithms, different implementations, or/and different optimization techniques. In many cases, the performances of different EPP types changed differently when the size of a state was varied. Second, the performances of some types of EPPs were affected by simultaneous events. On the one hand, some raw EP providers exploit simultaneous events. On the other hand, the corresponding JEPC bridge might introduce noticeable overhead due to the alignment of semantics (at present this is true for a few types of EPPs in case of the pattern matching EPA, see Section 8.3.2).

In the following, we present the results of two specific benchmarks that not only revealed different performances of different types of EPPs, but also determined a concrete threshold value simultaneously. We show and discuss the results of each benchmark only for selected types of EPPs. In the first benchmark, a pattern matching EPA searching for a simple sequential event pattern was executed over a generated test event stream. The results are shown in Figure 20.3 for two different types of EPPs. We ran the benchmark for different rates of simultaneous events shown on the x-axis. For example, a value of 1 % means that with probability of one percent there was an additional event at a time instant at that an event already occurred. The graphs show clearly that EPP_2 was strongly affected while EPP_1 was not affected at all. But in case of almost no simultaneous events, EPP_2 was superior to EPP_1. So, EPP_2 should be used if there are (almost) no simultaneous events and EPP_1 should be used otherwise. This outcome led to the creation of a new node in our decision tree (note that we use only abstract thresholds throughout this chapter for ease of presentation).

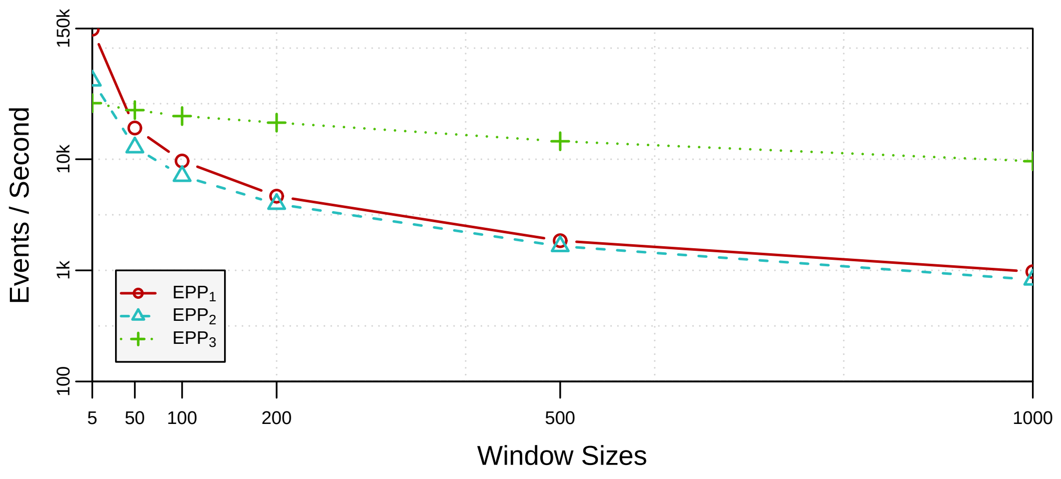


Figure 20.4: Effect of window sizes on correlators of different EPPs

Figure 20.4 shows the results of the second benchmark that also led to a node in our decision tree. It plots the achieved performances of a correlation EPA executed by different types of EPPs. The correlation EPA checked events of two different test event streams for a simple correlation condition and had two equally sized count-based sliding windows on its input event streams. We ran this benchmark for different sizes of the windows shown on the x-axis. Of course, as the sizes increased also the state of the EPA increased. But the benchmark further revealed that different types of EPPs reacted differently on an increasing size of the state. In case of a small state EPP_1 and EPP_2 performed much better than EPP_3 and in case of a large state the opposite was true. For instance, for window sizes of 5 events EPP_1 achieved a throughput that was more than 4 times higher than the throughput of EPP_3. Also EPP_2 was almost twice as fast as EPP_3. But when the window size was increased to 1,000 events, EPP_3 outperformed EPP_1 by factor 10 and EPP_2 by factor 12. These results are a good example of a performance gap caused by the raw EP providers themselves, because the corresponding JEPC bridges introduced no notable overhead.

On basis of all benchmarks, we manually created a decision tree that is shown in simplified form in Figure 20.5. It gets a single EPA as input and returns an ordered list of EPP types. The list keeps the EPP types in decreasing order of their expected performances. Therefore, an EPA is placed best on the first type of EPP that is available. Available means that there are instances of that EPP type in the federation and at least one of them has enough free resources for executing the EPA. If a certain EPP type is not available, then the next type of EPP in the list is tried. At the root of the

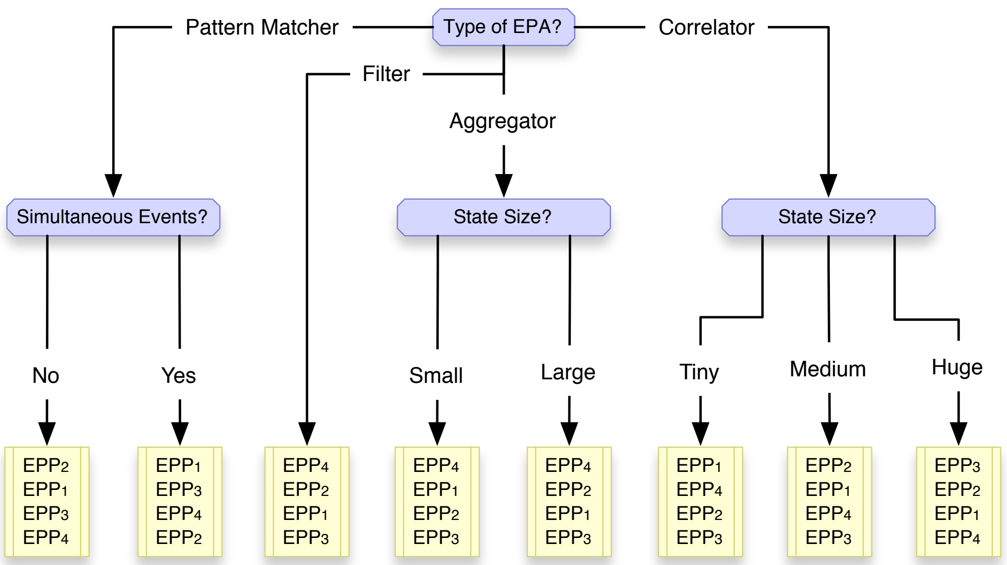


Figure 20.5: Classifier for assigning EPAs to EPP types

decision tree, incoming EPAs are separated by their types. Depending on its type, an EPA might be further classified until a leaf node that holds the ordered list of EPP types is reached. In case of filter EPAs there are no further criteria, neither of the EPA itself nor of the underlying event stream, that influence the ranking of types of EPPs. Thus, there is always the same order of EPP types. For all other types of EPAs, one further criterion is taken into account. As argued before, the expected performance of a pattern matching EPA may depend on the fact whether there are simultaneous events possible. According to this parameter, the order of EPP types differs significantly (e.g., EPP_2 is in one case the best choice and in the others the worst). For aggregation and correlation EPAs the size of the state influences the order of EPP types.

Ignoring a few results that were influenced by overhead due to JEPC and its bridges, we identified remarkable and potentially context-dependent differences in the performances of existing systems. The revealed performance gaps clearly show the potential that lies in federated EP infrastructures. In fact, already the presented benchmarks prove that a federation is superior to the parallelization of a single type of EPP. Federated EP becomes even more important, if we consider special implementation platforms such as GPUs or FPGAs in addition. Some types of EPAs can be expected to perform extremely well on those platforms (e.g., correlation EPAs with large states and complex correlation conditions [TM11]), while others might perform poorly (e.g., filter EPAs) in comparison to CPU-based implementations.

20.3.2 Load Balancing

The classifier presented in the last section determines only the type of EPP that is considered to be the best for a given EPA. But within a concrete federation, there might be no, one, or multiple instances of that type available. In the first case, the EPA cannot be placed on the best suited type of EPP. Then, the next best EPP type must be tried until an available EPP type has been found. In the second case, there are no further choices to make. The EPA is directly placed on the only available instance. In the last case, there are multiple instances by which the EPA could be executed. To clearly determine one of the eligible instances, we tried the most common selection strategies including random selection and round robin selection. Unfortunately, those strategies often resulted in a very uneven distribution of the load.

Therefore, we developed and implemented a selection strategy which better balances the load. In contrast to the common selection strategies, our so-called *lowest-load-least-selected* strategy requires continuous monitoring of all EPP instances of a federation. In particular, all resources that are limited and exclusively used by an EPP instance must be monitored. What resources that are is determined by the deployment model of a federation. If all EPP instances are running on different cores of a multi-/many-core CPU of the same machine, only the CPU utilization is important, because all other resources are shared. But if a federation runs on a cluster of interconnected machines, also the memory and network utilizations must be taken into account. The lowest-load-least-selected strategy chooses the EPP instance with the lowest load among all EPP instances that are selectable. If multiple EPP instances qualify because they have identical loads, then the EPP instance running the least number of EPAs is selected. However, if still no clear selection can be made, an EPP instance is chosen via round robin selection from the remaining EPP instances.

20.3.3 Clustering

Up to now, EPAs are deployed and executed individually. In particular, the federation manager deploys every single EPA on a highly suited and not overloaded EPP instance. During execution, the federation manager redirects all relevant incoming events to the EPP instance of an EPA and receives all its output events that are then forwarded to succeeding EPAs and output processors. Figure 20.6a illustrates this kind of deployment of EPAs. Because every deployed EPA needs at least two connections (at least one for receiving events from the federation manager and one for sending results), there are $2N$ or more connections in total for N deployed EPAs. This

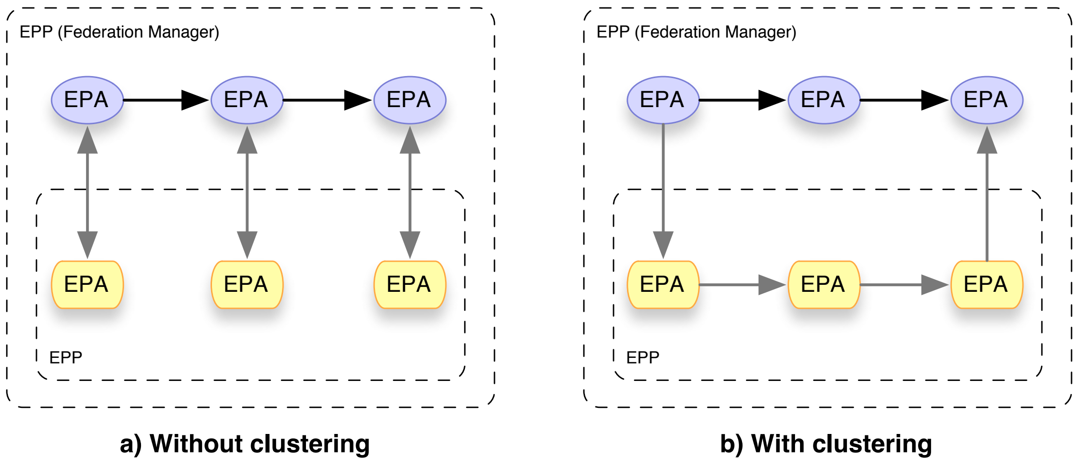


Figure 20.6: Clustering of adjacent EPAs being deployed on the same EPP instance

approach is fine in general and due to the distributed character of parallel and federated event processing. However, this kind of deployment is not optimal for EPAs being adjacent in the operator graph and deployed on the same EPP instance as in Figure 20.6a. As long as the output of one EPA is required only by succeeding EPAs deployed on the same EPP instance, there is no reason why the federation manager should receive the output events, because it would simply send back all events to the EPP instance, which causes unnecessary data transfer. Instead, a subgraph of EPAs such as the one in Figure 20.6a can be handled autonomously by the corresponding EPP instance in order to minimize communication overhead.

The federation manager clusters an operator graph before it is finally deployed and executed. A cluster is created for each set of EPAs that are adjacent in the operator graph and assigned to the same EPP instance. Such clusters are then deployed and executed as an atomic unit. The transfer of events changes as follows. The federation manager is only in charge of forwarding all input events of the root EPAs of a cluster and receives only output events from the leaf EPAs of a cluster. Figure 20.6b shows how the three adjacent EPAs are clustered by the federation manager. As a result, the communication costs reduce significantly. After clustering, only 2 instead of 6 connections between the federation manager and the EPP instance are necessary. The impact of this optimization depends on the deployment model of a federation. On a single machine, a connection is established between two threads and, thus, relatively cheap. But on a cluster of interconnected machines, a connection is established between two machines and relatively costly. However, reducing the transfer of events has a noticeably positive effect on the overall performance in every case.

20.4 Implementation

We implemented the federation manager as an additional EP provider of JEPC. This means that it can be used like any other EP provider of JEPC. Our implementation is arbitrarily configurable, but comes also with a default configuration that works well in most situations. Since the federation manager cannot execute EPAs by itself, it must be wrapped around a set of EPP instances that are able to execute EPAs. In contrast to the other EP providers of JEPC, the federation manager is thread-safe and, thus, allows events to be concurrently injected (see Figure 20.2).

```
EPProvider federation = new FederatedEngine(
        TypeSelector typeSelector,
        InstanceSelector instanceSelector,
        EPProvider epp1, EPProvider epp2, ..., EPProvider eppn
);
```

Listing 20.1: Creating a federation of EP providers in JEPC

Listing 20.1 shows how to create a new federation in JEPC. Of course, a federation consists of multiple EPP instances $epp_1, epp_2, \ldots, epp_n$ that potentially are of different types. Such an EPP instance can be any EP provider of JEPC, even another instance of `FederatedEngine`. The parameter *typeSelector* is a user-defined classifier that assigns every EPA to a list of types of EPPs (see Section 20.3.1). If it is not specified, our default decision tree is used. The parameter *instanceSelector* is a user-defined selection strategy for choosing a concrete EPP instance in the case that multiple instances of the same type are available (see Section 20.3.2). If it is not specified, EPP instances are chosen according to our lowest-load-least-selected strategy.

20.5 Evaluation

We conducted several experiments that all confirmed our claim that a federated EP infrastructure is able to execute queries more efficiently than a single type of EPP running in parallel. In the following, we present one of those experiments. On the basis of a simple test query, it clearly shows the superiority of a federation. The experiment was performed on a single machine with an i7-2600 CPU consisting of 4 cores, 8 GiB main memory and running 64-bit HotSpot VM (1.7.0_13). Each tested configuration consisted of one federation manager, which used the default parameters and had control over exactly four EPP instances running in parallel on the multi-core CPU.

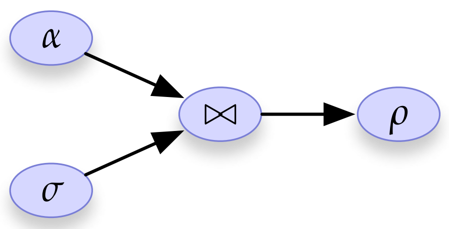


Figure 20.7: Test query

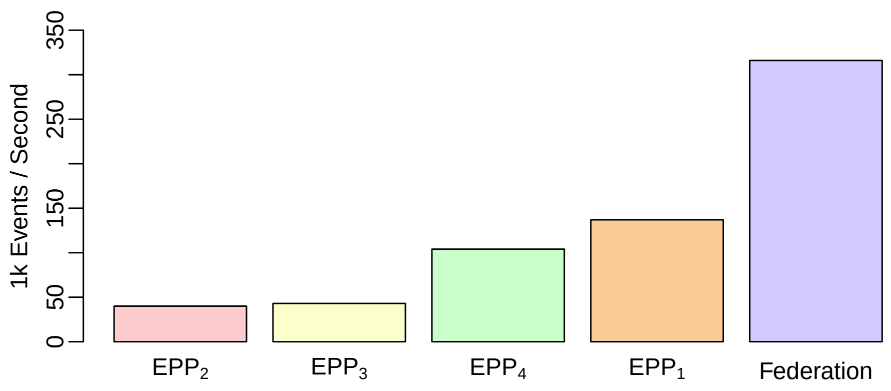


Figure 20.8: Performance of the test query for different configurations

The selected test query we present the detailed results for was one query of a larger set of queries we tested and is depicted in Figure 20.7. It was quite simple but already sufficient to let a federation clearly outperform all homogenous configurations. The test query consisted of four different EPAs that were combined into a single EPN. At first, an aggregation EPA and a filter EPA consumed exactly the same test event stream that contained events which had three 32-bit integer numbers as payload. There were no simultaneous events. A succeeding correlation EPA joined the two output streams and forwarded its own results to a pattern matching EPA. We executed this test query for five different configurations of the federation manager. In four configurations, all four EPP instances were of the same type (EPP_1, EPP_2, EPP_3, or EPP_4) and, thus, represented the parallelization of a single type of EPP. The fifth configuration consisted of a federation that had exactly one instance of each EPP type.

Figure 20.8 plots the achieved throughputs of all configurations. The results clearly show that the federation outperformed every homogenous configuration by many times. Note that if only one EPA of a query is executed poorly, it becomes a bottleneck and the performance of the entire query decreases.

20.6 Related Work

The federation manager touches on several different topics of the parallel and federated data processing areas in which a lot of interesting work has been done in the past. Therefore, the related work to discuss is diverse.

Cyclops. The Cyclops platform [LHB13] is intended to be used on top of different types of data processing and data storage systems. For example, the configuration used in the cited paper integrates a centralized stream processing system, a distributed batch processing system, and a distributed stream processing system. The optimizer of Cyclops delegates the execution of a continuos query to that system which is expected to perform best. As in the case of the federation manager, the optimizer of Cyclops is implemented as a classifier that is based on performance benchmarks. The motivating examples focus only on continuous jumping window aggregation queries. In its evaluation, Cyclops demonstrates that depending on the event stream characteristics and the specification of the jumping window (it can be defined being close to stream processing, close to batch processing, or somewhere in-between), the best performing processing paradigm differs. In contrast to the federation manager, Cyclops does not deal with arbitrary multi-operator queries. Moreover, it focuses on different processing paradigms and not on different implementations of systems belonging to the same paradigm. Nevertheless, Cyclops supports our main statement that implementations among different existing EP providers, different processing paradigms (e.g., batch processing and stream processing), and different implementation platforms (e.g., CPUs and GPUs) can be combined into very powerful infrastructures.

Federated DBMS. A federated DBMS is a meta database system which has control over multiple and potentially heterogenous DBMSs that are called *component DBMSs* [SL90]. Thus, the overall picture is the same as in our definition of a federated EP system. But in the case of a federated DBMS, the integration of the different component DBMSs is in focus rather than the optimization of queries. This is mainly because of the assumption that the distribution of data is naturally given due to the existence of several component DBMSs and not done on purpose with the aim to optimize the overall performance or availability. The analogous problem to federated DBMS is the integration of different existing EP providers. However, this problem has been already solved by the core of JEPC.

Intra-operator Parallelization. One approach to parallelization often implemented by parallel database systems is intra-operator parallelization [DG92]. In this approach, the data a single operator has to process is partitioned and each partition is processed in parallel by multiple instances of that operator. This approach is proven to work fine for operators of the relational algebra and can be taken over to the filter, the aggregation and the correlation EPAs therefore. In addition, recent research results showed how also the pattern matching EPA can be parallelized [Bal13]. Intra-operator parallelization of the EPAs of JEPC is on our agenda for future work. However, intra-operator parallelization is a completely different kind of query parallelization and does not enable federated EP. Note that the distribution of EPAs for parallel execution and intra-operator parallelization can be used in combination without any problems. Therefore, they are complementary and no competing approaches.

MaxStream. There has been already done research close to federated stream processing in the MaxStream project [Bot09, Bot10]. MaxStream is a platform mainly for integrating DBMSs and DSMSs that operate on time-varying relations. Therefore, it is comparable to cyclops that also integrates different processing paradigms into a federation. MaxStream is designed and implemented as an extension to SAP MaxDB, which is a federation engine for DBMSs. Since MaxStream is based on a federation engine for databases, DSMSs are integrated in the same way as databases and used to continuously execute traditional database queries. However, MaxStream focuses purely on the integration of DBMSs and DSMSs. The possibility to optimize query execution in the context of a federation is not considered at all [LHB13].

StreamCloud. The framework StreamCloud [Gul10, Gul12] parallelizes a DSMS by running multiple instances distributed across a set of machines. StreamCloud splits the operator graphs of queries according to the operator-set approach and deploys the resulting parts on different nodes. The strategy used to partition query graphs is simple but effective. Before each stateful operator the graph is cut and falls into pieces. Contrary to StreamCloud that supports only a single type of DSMS at a time, the federation manager of JEPC assumes a federation of EP providers each potentially having different strengths and weaknesses. For this reason, the federation manager implements the operator-set approach differently in order to fully exploit the different characteristics of different EP providers. It first assigns every single EPA to the instance that is best suited and then creates clusters of EPAs.

Summary

In this chapter, we present a JEPC extension that manages parallel and federated EP infrastructures on basis of JEPC. We introduce a federation manager that decomposes operator graphs and distributes the resulting parts across multiple parallel running and potentially heterogenous (i.e., a federation) EP providers. Each part of a decomposed operator graph, which can be a single EPA, multiple connected EPAs or an entire query plan, is deployed on the EP provider that is expected to execute it with highest possible performance. Simultaneously, the federation manager also balances the load. Experimental evaluations confirmed that a query can be executed more efficiently by a federation than by parallel running EP providers of the same type.

21 Conclusions

This part presented several extensions to EP technology on basis of the JEPC middleware. JEPC was extended by a novel pattern matching EPA named TPStream, which is more expressive than traditional pattern matchers. We showed how to enhance JEPC to support the processing of spatial data. A matchmaker was introduced to create self-adaptive EP applications. This component establishes all connections between the elements of an EP application automatically and is able to handle differences in meanings. We proposed a method to update continuous queries on-the-fly which is the first one that is safe and efficient at the same time. It relies on a high-performance storage system for recording and reloading event streams. Therefore, we introduced the B^+-tree event store that is a storage system being capable of recording millions of events per second by exploiting the append-only nature of streams. The optimizer we presented improves the performance of JEPC queries significantly. It applies not only techniques known from databases, but also novel techniques for pattern matching queries that were confirmed to be highly effective. Lastly, a framework was presented to further improve the performance by distributing the workload across parallel running instances of JEPC. It also supports federations of heterogenous instances thanks to the unification of JEPC. We showed that a federation can execute queries with a performance that is not achievable via the parallelization of a single type of system.

Part IV

Thesis Conclusions and Future Research

22

Thesis Conclusions

The constantly growing diversity of data models, query languages, and entire systems in the area of event processing (EP) over the past years leads to serious problems today. Applications often need multiple different components of EP technology. Besides this, it is a frequent occurrence that a component must be exchanged for a different one. At present, the necessary integration work is purely manual, highly error-prone and extremely costly. Because it cannot be expected that there will be established any standards in the near future, a reasonable solution lies in platforms that are able to integrate different components of EP technology.

In this thesis, we presented the design and implementation of a novel middleware for EP named *Java Event Processing Connectivity* (JEPC). This middleware abstracts from different EP providers, including their data models and query languages, and offers a uniform platform for developing EP applications. It is the first of its kind and proves the feasibility of powerful integration platforms for EP. We specified JEPC as a virtual EP provider that has its own data model and query language. Existing EP providers are integrated via individual bridges that map the JEPC specification to the corresponding specifications of existing EP providers. We showed that JEPC bridges allow to integrate not only different existing stream processing engines, but also standard database systems. Moreover, we used a native implementation of the

JEPC specification to bring in high-performance algorithms and implementations for important components of EP. Thanks to the unification, this native EP provider can seamlessly work together with every integrated EP provider. This approach can also be used to perform selected tasks on special implementation platforms (e.g., GPUs). Note that not only applications benefit from the advantages of the middleware. Also components developed on top benefit from JEPC to the same extent.

We extended the state-of-the-art method for event pattern matching. Existing approaches capture the temporal nature of EP only insufficiently. Therefore, we presented a novel approach to event pattern matching that allows to derive and correlate high-level events with time-interval semantics. On basis of the derived high-level events, arbitrarily complex temporal event patterns can be detected. We implemented the novel event patten matcher as a JEPC extension so that it can be used in conjunction with all other components that are supported by the middleware.

Since applications are becoming more and more dynamic, today's static EP providers are not able to adequately support many emerging applications. Therefore, we developed the concept of dynamic event processing and implemented it in the form of JEPC extensions. Dynamic event processing comprises two features. First, we presented a matchmaker that fully automatically manages the entire interconnection of all elements of a dynamic EP application. Second, because the context changes constantly in dynamic EP applications, also the processing logic must change constantly. However, existing EP providers do not support adaptive processing logics. Therefore, we introduced an approach to efficiently and safely updating arbitrary parts of the processing logic at runtime. Because the efficiency of our approach relies on a fast storage system for recording and reloading event streams, we presented a high-performance event store that writes event streams with nearly maximum possible write rate to disk. Simultaneously and almost for free, an index is created to support fast replaying and querying of recorded event streams.

We developed a query optimizer that transforms the query plans of JEPC queries at the middleware level. Because different EP providers can be used at the same time and in parallel, JEPC allows to choose from different implementations of the same component in the first place. We found out that different implementations perform quite differently in certain situations. In order to exploit this fact, we presented a novel optimizer that assigns every part of a query plan to the EP provider that is best suited from a performance point of view. An experimental evaluation clearly showed that a federation of different EP providers is more powerful than a parallel infrastructure consisting of one type of EP provider only.

23

Future Research

Outline

23.1 Introduction

Because this thesis presents the design and implementation of a novel EP middleware providing unique functionality via its extensions, it opens a wide range of interesting future research. In the next sections, we give an overview of the most attractive topics we could not work on intensively enough owing to the lack of time.

23.2 Additional JEPC Bridges

In principle, future work on the core of JEPC can be done in two different ways. First, the core itself can be extended in order to get a more powerful and expressive foundation of JEPC. Second, the existing set of JEPC bridges can be extended by additional JEPC bridges to currently unsupported EP providers. However, we think the first way is not very promising and advisable. An extension of the core itself by new features (e.g., new types of EPAs) will cause serious consequences. Extending the core itself means extending the JEPC specification. Thus, all existing JEPC bridges must be extended simultaneously by implementing the new features. Furthermore, the effort for implementing new bridges will grow. In the worst case, a new feature forbids a JEPC bridge to some EP providers that can be supported without that new feature. We are convinced that our specification of the JEPC core is a good compromise between compatibility and expressive power. For instance, the current set of basic EPAs contains all types of EPAs the event processing community considers to be essential [EB09, EN10]. This ensures both sufficient expressiveness and wide support by raw EP providers, which enables the implementation of JEPC bridges to them. Instead of extending the JEPC core, the development of additional JEPC bridges seems more worthwhile. Additional JEPC bridges can be developed for not only dedicated EP systems and general-purpose stream processing engines, but also more exotic EP providers. For example, distributed stream processing infrastructures such as Storm [Tos14] are becoming increasingly popular nowadays. Furthermore, highly parallel hardware architectures such as many-core CPUs [Vaj11], graphics cards [Ven03] and FPGAs [WTA10] become more and more important and are promising platforms for high-performance event processing [TM11]. The native EP provider of JEPC is designed for traditional CPUs and cannot be ported to fundamentally different architectures. Instead, EPA implementations that exploit highly parallel hardware are needed. To fully benefit from such hardware, research on inter- and especially intra-operator parallelism [DG92, MD95] with respect to the basic EPAs of JEPC is needed.

23.3 What-If Analysis and Query Quality

It is possible to execute JEPC queries over recorded or synthetically generated event streams stored in a standard database, because the JEPC semantics is compatible with standard database systems. The EPAs filter, aggregator and correlator are snapshot-reducible to operators of the relational algebra. Pattern matching is not part of the relational algebra. However, in Chapter 9 we show how to perform pattern matching purely via SQL. Note that there arise possibilities for optimization, when an event stream or the relevant part of it is known in advance. For instance, an index can be created in order to read and process only events of candidate sequences (see page 301 in Section 19.2.4). In addition, JEPC provides a query language and supports the connectivity to several event stores. Therefore, everything that is needed to support the evaluation of JEPC queries over stored event streams is already present. Together with the sound JEPC semantics, this is an ideal basis for the design and implementation of a simulation environment in which JEPC queries can be safely, isolated and efficiently executed over historical events of a real EP application. One important use case of such a simulation environment is What-If analysis. Going one step further, the overall quality of JEPC queries can be measured and improved in such an environment. We think that the use of a simulation environment raises interesting research questions (rather than its design and implementation) and is important for the integration of event processing technology into business-critical applications.

What-If analysis is performed in many different ways in the context of databases today [LÖ09] and it can also be performed in many different ways in the context of event processing. On basis of recorded event streams of a real EP application, different queries and slightly different definitions of a query can be tested and successively improved. For example, precision and recall of a query with respect to the detection of SoIs can be determined in a test phase. In a following optimization phase, different configurations of a query (e.g., different window sizes, different thresholds in filter conditions) can be compared in order to find the best one. The other way around, recorded event streams can be modified to check existing queries for the ability to detect SoIs on different data. For instance, additional SoIs can be added by hand or event streams of high quality can be enriched with noise. However, working out different useful types of What-If analysis in the context of event processing is part of future research. In addition, future research should also develop methodologies and techniques that allow for the implementation of electronic assistants which guide users or even perform some tasks (semi-)automatically.

In the last paragraph, we indicated that it is currently not possible to give measurement methods for the quality of continuous queries and advices for the management of the life cycles of continuous queries, quite simply because there has not been done intensive research on these topics yet [HS13]. Since in stream processing the roles of data and queries are interchanged in comparison to database systems, the quality of continuous queries running in a stream processing system is as important as the quality of data stored in a database systems. There has been done much work on managing data quality [Lee06]. The management of query quality should be elaborated in the same way. We think that interesting new research is possible due to the unique properties of continuous queries. However, existing work focusing on data quality can serve as a starting point. For example, most dimensions of data quality [PLW02] can also be used for a set of continuous queries. Just as the set of all data stored in a database, the set of all running queries should be complete, up-to-date and free of error.

23.4 Application of Dynamic Event Processing

In this thesis, we proposed dynamic event processing to enable the use of event processing technology in applications with constantly changing contexts. Dynamic event processing makes applications (self-)adaptive and comprises two novel features from a technical point of view. First, an additional matchmaker component manages the entire interconnection of all elements fully automatically during runtime. Second, an extra method allows, in conjunction with a high-performance event store, to safely and quickly update continuous queries on-the-fly. However, the latter subject was only addressed at a technical level in this thesis. For getting the best out of dynamic event processing, it is crucial to utilize the update method properly.

We outlined a general and powerful way of exploiting the update method on page 274 in Section 18.2.1. Basically, models (particularly context models [Bet10, SL04] and statistical models [McC02]) can be used to capture the behavior of the application context including individual monitored objects to deploy and update queries for the detection of SoIs. Especially behavior not in conformity with the expected behavior can be detected this way and is of high interest in anomaly detection. However, two important questions that require extensive investigation arise. First, how to get and maintain suitable models? One possible research direction would be applying data mining methods to historical data recorded by the event store [Bis06, FPS96, WFH11]. Second, how to utilize the models? A simple but effective way is the use of prepared query definitions whose parameters are maintained by the models [HS13].

Appendices

A

Development and Evaluation Details

Outline

A.1 Introduction

While we were working on JEPC, several bridges to different raw EP providers have been implemented and manifold extensions have been developed. To ensure the quality of our implementations and novel concepts, we conducted extensive and profound checks for errors as well as performance evaluations.

A.2 Unit Tests

On basis of the sound and formal foundation of the JEPC semantics, the output and the behavior of JEPC can be deterministically and exactly predicted for a given set of queries and a given set of input event streams. This allows for the implementation of unit tests that execute certain workloads and check the output and behavior of JEPC for correctness. While there are unit tests for almost every component of JEPC, we particularly focused on the JEPC bridges for two reasons. First, the JEPC bridges are the most critical and error-prone part, because they map the semantics of JEPC to the potentially different semantics of raw EP providers. Hence, a JEPC bridge must possibly align the different semantics of a raw EP provider. The problem is that the alignment techniques a JEPC bridge applies must work correctly in every setup. Therefore, we created a lot of different test setups in order to cover as many situations as possible. Second, JEPC is expected to get additional JEPC bridges in the future. The design and implementation of a JEPC bridge is much easier and faster when comprehensive unit tests that can reveal errors immediately at development time are available.

A.3 Evaluation Framework

The method of unit testing is perfect for revealing errors, but it is not suitable for performance evaluations. Unit tests are as compact as possible and have usually small as well as specially prepared workloads that do not correspond to real-world workloads in general. For plausible performance evaluations, massive workloads with the characteristics of real-world applications are required. Therefore, we developed an evaluation framework specifically for performance evaluations of JEPC and its components. This framework became a toolbox that can be used to generate massive workloads which have certain characteristics. Its cornerstones are described in the following.

A.3.1 Random Number Generator

At the lowest level of the evaluation framework, there is a random number generator (RNG) which can be used in many different ways to create workloads for experiments. The RNG is for obtaining a set of random numbers that follow a certain probability distribution within a user-defined range of values. In particular, users have to specify a range of values $[Low : High]$, a probability distribution and the total number of random numbers to generate. Then, the RNG returns a set of numbers that fulfill the given specifications. In the following, all available and used probability distributions are listed. Most of them are parameterized and, of course, it is possible to freely set parameters in the evaluation framework. However, for each available probability distribution there is at least one predefined parameter setting provided. Note that there are different types of probability distributions. Some are discrete, others are continuos. Also the domains vary. Some have a fixed and finite range of numbers as domain (e.g., $[0 : 1]$), others have all real numbers as domain. Because we forced all distributions to implement our RNG interface, there are some small side-effects. First, if the underlying probability distribution is discrete, the returned random numbers are also discretized. There is no interpolation or other procedure applied. Second, the domain of the underlying probability distribution is mapped to $[Low : High]$ via a transformation consisting of a translation and a scaling operation. Third, if the domain of the underlying probability definition is unbounded at one or both sides, the transformation includes not only a scaling and a translation, but also a clipping.

Uniform and Fixed Distributions. The uniform and fixed distributions do not have any parameters. Therefore, our presets `UNNIFORM` and `FIXED` must be used. When choosing the uniform distribution [Bis06, ES10, For11], the numbers in the returned set have been generated using the same constant probability. The probability density function (PDF) is given by $f_{uniform}$.

$$f_{uniform}(x) = \begin{cases} 1/(High - Low) & \text{if } x \in [Low : High] \\ 0 & \text{if } x \notin [Low : High] \end{cases}$$

The fixed distribution is a special case. Here, always the number $Low/2 + High/2$ is returned. The corresponding probability density function is given by f_{fixed}. Figure A.1 shows these two non-parameterized probability distributions.

$$f_{fixed}(x) = \begin{cases} 1 & \text{if } x = Low/2 + High/2 \\ 0 & \text{if } x \neq Low/2 + High/2 \end{cases}$$

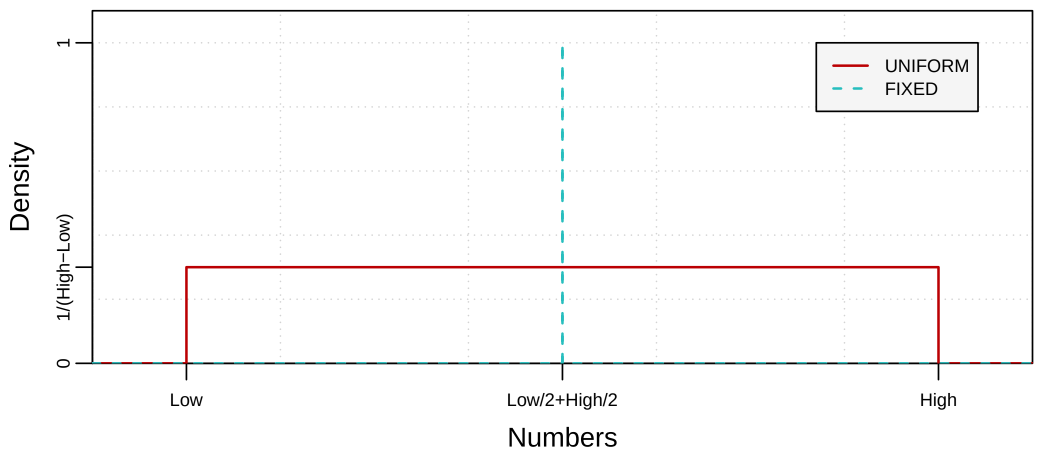


Figure A.1: Uniform and fixed distributed numbers

Triangular Distribution. In the case of the triangular distribution [ES10, For11], the probability density function is a triangular defined by *Low*, *High* and a free parameter *c*. From *Low* to *c* the function increases linearly, and from *c* to *High* it decreases linearly. The PDF of the triangular distribution is given by $f_{triangular}$.

$$f_{triangular}(x) = \begin{cases} 0 & \text{if } x < Low \\ \frac{2(x-Low)}{(High-Low)(c-Low)} & \text{if } Low \leq x < c \\ \frac{2(High-x)}{(High-Low)(High-c)} & \text{if } c \leq x \leq High \\ 0 & \text{if } x > High \end{cases}$$

Besides the option to freely set *c* to any value in $(Low : High)$, there are three predefined configurations provided. In `TRIANGULAR1`, *c* is set to the value that is one quarter away from *Low* and three quarters away from *High*. The preset `TRIANGULAR3` uses the exactly opposite weights. In `TRIANGULAR2`, the middle of the user-defined value range is used to set *c*. Figure A.2 shows all three presets.

Normal Distribution. The normal distribution [Bis06, ES10, For11] is a symmetric and bell-shaped probability distribution. It is the most widely used probability distribution in statistics [Bis06, For11]. The corresponding PDF f_{normal} has the mean μ and the standard deviation σ as parameters.

$$f_{normal}(x) = \frac{1}{\sigma\sqrt{2\pi}} e^{-\frac{1}{2}(\frac{x-\mu}{\sigma})^2}$$

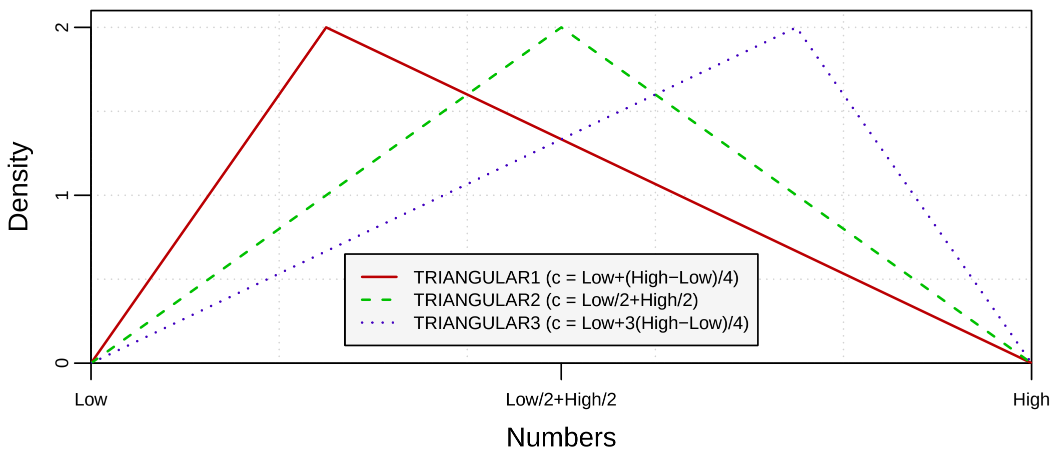


Figure A.2: Triangular distributed numbers

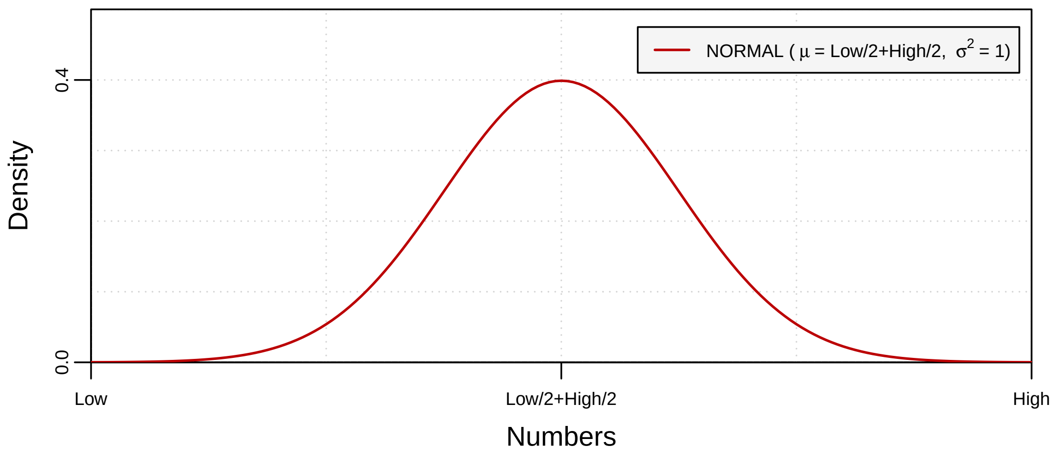


Figure A.3: Normal distributed numbers

While the standard deviation σ can be freely set, the mean μ defines the center and is always given by $Low/2 + High/2$ in our evaluation framework. Figure A.3 shows the only provided preset `NORMAL` in which σ is set to one.

Beta Distribution. Another widely used probability distribution is the beta distribution [Bis06, ES10, For11] whose corresponding PDF is given by f_{beta}. It is based on and named after the beta function $B(\alpha, \beta)$. The parameters α and β can be freely set. Figure A.4 shows our three predefined configurations of the beta distribution.

$$f_{beta}(x) = \frac{x^{\alpha-1}(1-x)^{\beta-1}}{B(\alpha,\beta)}\text{, with } B(\alpha,\beta) = \int_0^1 u^{\alpha-1}(1-u)^{\beta-1}\,\mathrm{d}u$$

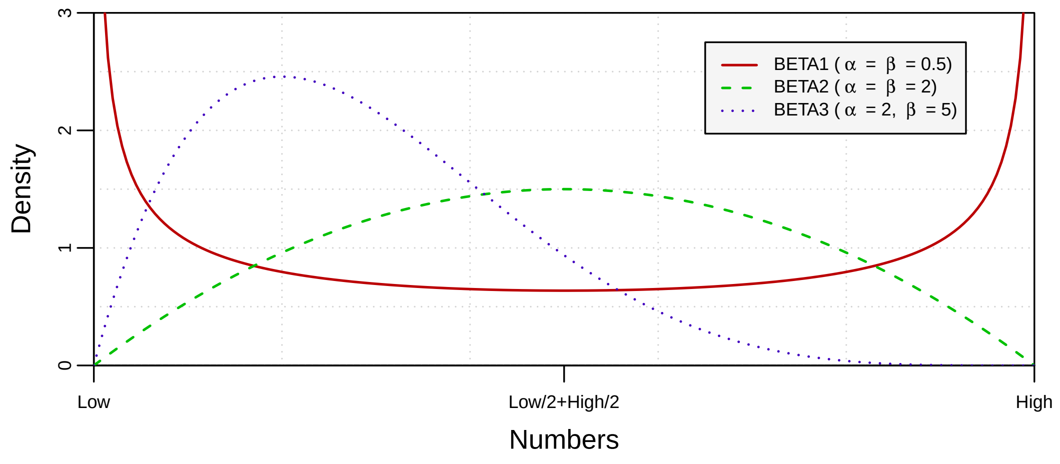


Figure A.4: Beta distributed numbers

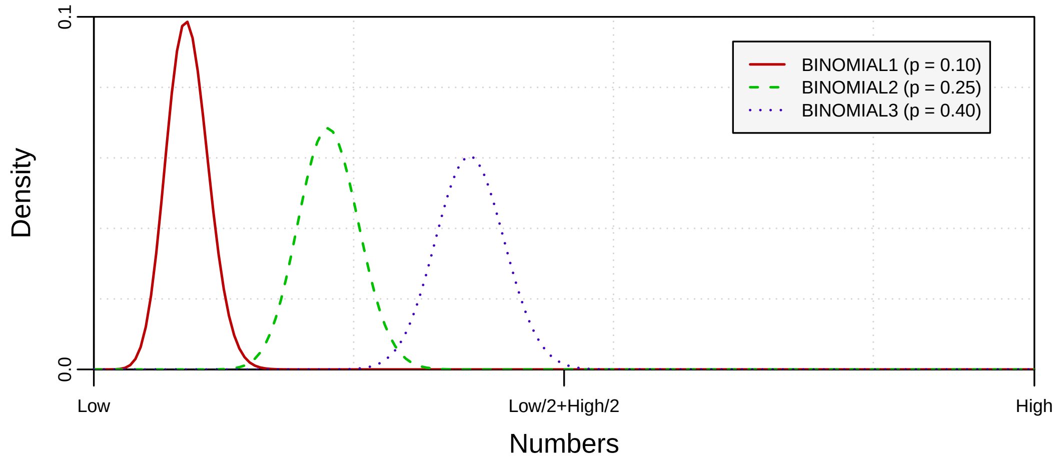


Figure A.5: Binomial distributed numbers

Binomial Distribution. The binomial distribution [Bis06, ES10, For11] is important for the modeling of stochastic processes. Its PDF is given by $f_{binomial}$. While n is fixed in our implementation, the parameter p can be freely chosen from $(0:1)$. Figure A.5 shows all predefined configurations of the binomial distribution.

$$f_{binomial}(x) = \binom{n}{x} p^x (p-1)^{n-x}$$

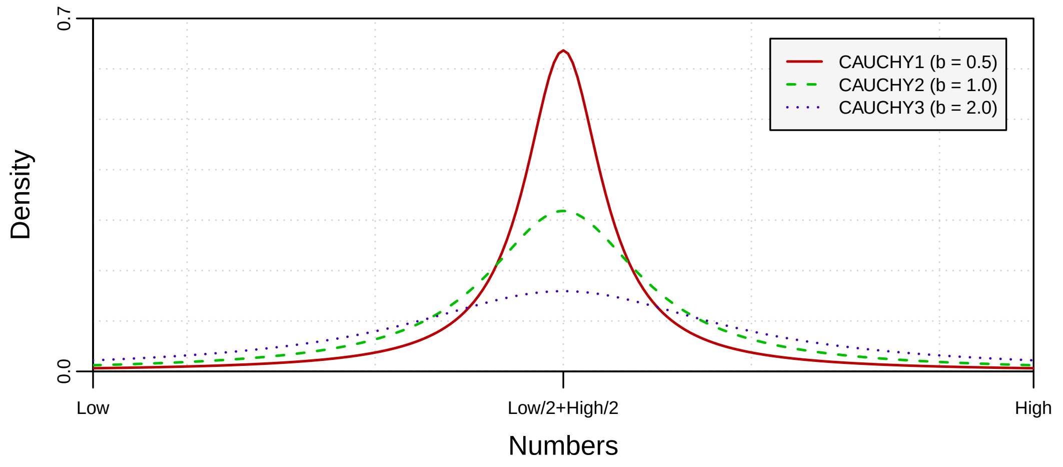


Figure A.6: Cauchy distributed numbers

Cauchy Distribution. The Cauchy distribution [Bis06, ES10, For11] is important in physics and has the PDF f_{cauchy} with free parameter b. Figure A.6 shows our three predefined configurations.

$$f_{cauchy}(x) = \frac{1}{b\pi\left(1+\left(\frac{x-Low/2+High/2}{b}\right)^2\right)}$$

Chi-Squared Distribution. In statistical inference, the chi-squared distribution [ES10, For11], which has the probability density function $f_{chi-squared}$ with free parameter v, is widely used. Figure A.7 shows three configurations being predefined in our evaluation framework.

$$f_{chi-squared}(x) = \frac{1}{2^{v/2}\Gamma(v/2)} x^{(v-2)/2} \mathrm{e}^{-x/2} \, ,$$

$$\text{with } \Gamma(r) = \int_0^\infty u^{r-1}\mathrm{e}^{-u} \, \mathrm{d}u$$

Zipf Distribution. The Zipf distribution [ES10] is capable of modeling many real-world phenomena. Its PDF is given by f_{zipf} with free parameter s. Figure A.8 shows the only provided preset in which s is set to 1.0.

$$f_{zipf}(x) = \frac{x^{-(s+1)}}{\xi(s+1)}, \text{with } \xi(w) = \frac{1}{\Gamma(w)} \int_0^\infty \frac{u^{w-1}}{\mathrm{e}^w - 1} \, \mathrm{d}u$$

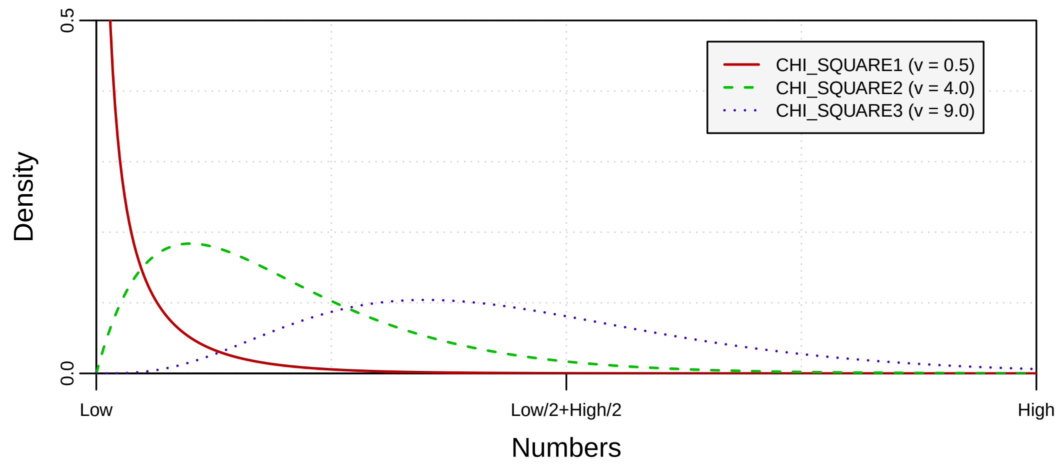


Figure A.7: Chi-squared distributed numbers

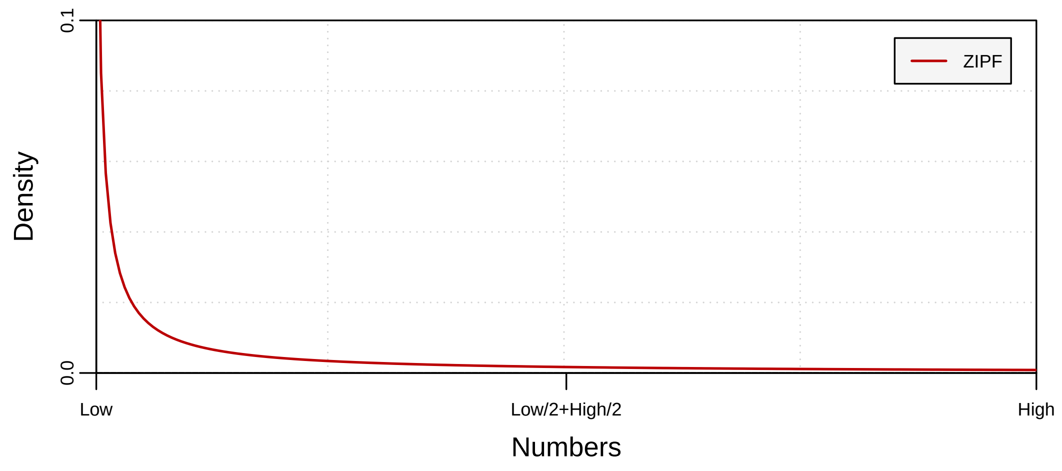


Figure A.8: Zipf distributed numbers

Many of the available probability distributions are not symmetric. In order to increase the number of options, we provide a higher-order function f_{mirror} that mirrors a given probability density function at $Low/2 + /High/2$.

Note that in all experiments which are presented in this thesis and made use of the evaluation framework, only the presented presets were used. In general, we tested all available predefined probability distributions whenever the underlying distribution of a parameter had an effect.

A.3.2 Event Generators

The random number generator following a user-defined probability distribution is perfectly suited for generating the payloads of events. Therefore, we defined event schemas such that they consisted only of numeric attributes in general. Then, the payloads of events were points in a hypercube whose positions could be generated in a controlled way by using one instance of the random number generator for each single event attribute. In addition, also the timestamps of events can be generated by random resulting in a certain distribution of events along the timeline. All event generators we implemented for our experiments became part of the evaluation framework.

A.3.3 Query Generators

The generation of continuous queries is not less important than the generation of events. But the possible applications of the random number generator are more diverse in this case. We mostly used the random number generator to create parameters of EPAs in a controlled way. For instance, we used three different random number generators to create filter conditions for evaluating the query indexes in Chapter 11. One instance of the generator was in charge for selecting a subset of all event attributes for that *between* predicates were defined. Because *between* predicates restrict the attribute domain to a certain range being identical to a one-dimensional interval, we used a second instance for generating the positions of intervals and a third instance for generating the sizes of intervals. For each of the three instantiated generators, the underlying probability distribution as well as the parameters *Low* and *High* can be set separately. This results in a large range of different query workloads that we could use in our experiments. As in the case of event generators, also all our implemented query generators became part of the evaluation framework.

B

Matchmaker Example

Outline

B.1 Introduction

Chapter 7 of this thesis outlines the API of JEPC that is nothing extraordinary since the JEPC middleware is a virtual EP provider designed to meet the behavior and feel of existing stream processing systems which are familiar to informed readers. Things change when our matchmaker extension is used. Then, the notion of a data or event stream is completely removed from the API. In order to illustrate the new behavior and feel, we present a complete and detailed example of an EP application that utilizes the matchmaker in this chapter. The example application is oriented towards the examples used in Chapter 16. Because we want to give a good impression in a nutshell, we do not use our high-level query language JEPC-QL.

B.2 Wrapping Instances of JEPC

The matchmaker extension has been designed as a wrapper for single instances of JEPC. Listing B.1 shows how to create a new matchmaker for an existing JEPC instance. In the first line, a new JEPC instance is created (an EP provider of type Esper in the concrete example). Next, a new matchmaker is created for the recently created JEPC instance in line 2. Note that the matchmaker can be used with any type of an EP provider. This explicitly includes federations of multiple and potentially different EP providers that are EP providers again.

```
1 EPProvider epProvider = new EsperEngine();
2 Matchmaker matchmaker = new Matchmaker(epProvider);
```

Listing B.1: Wrapping an instance of JEPC

B.3 Transformations

We introduced transformations to increase the power and flexibility of the matchmaker as well as the quality and robustness of resulting applications. At first, we show how to create basic transformations that are the only supported types of transformations in JEPC. Then, we demonstrate how the matchmaker automatically composes basic transformations to arbitrarily complex transformations.

B.3.1 Basic Transformations

In our formal specification as well as in our implementation of the matchmaker we only support basic transformations (namely conversion, merge and split). Beginning on page 236 in Section 16.4.1, we give a concrete example for each type of basic transformation. The listings presented in the following show how to express these concrete example transformations in JEPC.

```
 1  Attribute requiredAttribute = new Attribute("temperature", DataType.FLOAT)
 2                              .setProperty("Unit of measurement", "Degrees Celsius");
 3  Attribute imageAttribute = new Attribute("temperature", DataType.FLOAT)
 4                              .setProperty("Unit of measurement", "Degrees Fahrenheit");
 5
 6  ConversionFunction<Float,Float> celsiusToFahrenheitFunc =
 7    new ConversionFunction<Float, Float>() {
 8      @Override
 9      public Float apply(Float input) {
10          return (input * 9) / 5 + 32;
11      }
12  };
13
14  Conversion celsiusToFahrenheit = new Conversion("Celsius -> Fahrenheit",
15                            requiredAttribute, imageAttribute, celsiusToFahrenheitFunc);
```

Listing B.2: Conversion in JEPC

Recall that every transformation consists by definition of two components, a schema transformation and a data item transformation. Listing B.2 shows the example conversion being expressed in JEPC. The first four lines define the schema transformation. In the case of a conversion, exactly one required attribute (`requiredAttribute`) and exactly one attribute of the image (`imageAttribute`) must be defined. The meaning is that the transformation is only applicable to schemas that have an attribute which matches the required attribute. Then, the transformation exchanges the matching attribute of the schema for the attribute of its image. Besides the schema transformation, we must define an additional function which performs a corresponding transformation of all values belonging to the attribute being converted. In the concrete example, this is the function `celsiusToFahrenheitFunc` that converts temperature values from degrees Celsius into degrees Fahrenheit. Finally, all parts are put together for creating the complete conversion in lines 14–15 (`"Celsius -> Fahrenheit"` is the user-defined identifier of the transformation that is needed to remove the transformation from the matchmaker at any time).

Listing B.3 shows the example merge in JEPC. To be applicable to a schema, it requires the schema to have a matching attribute for every single attribute in `requiredAttributes`. Note that `requiredAttributes` has set semantics during matchmaking. Due to the permutation being mandatory in every mapping, the order of attributes does not matter in general. However, the order of attributes in `requiredAttributes` determines the order of arguments of the data item transformation `addAreaFunc` (i.e., the order of values in `Object[] input`). After the merge has been applied to a suitable schema, the resulting schema has a new attribute `imageAttribute`. The data item transformation `addAreaFunc` defines how the values of that new attribute are computed.

```java
Attribute[] requiredAttributes = new Attribute[] {
        new Attribute("width", DataType.INTEGER)
            .setProperty("Unit of measurement", "Inch"),
        new Attribute("height", DataType.INTEGER)
            .setProperty("Unit of measurement", "Inch"),
};
Attribute imageAttribute = new Attribute("area", Attribute.DataType.LONG)
                               .setProperty("Unit of measurement", "Square inch");

MergeFunction addAreaFunc = new MergeFunction() {
    @Override
    public Object apply(Object[] input) {
        long width  = (long) input[0];
        long height = (long) input[1];
        return width * height;
    }
};

Merge addArea = new Merge("Add area",
                                 requiredAttributes, imageAttribute, addAreaFunc);
```

Listing B.3: Merge in JEPC

Lastly, the implementation of the example split is shown in Listing B.4. It is applicable to all schemas that have a matching attribute for `requiredAttribute`. Then, the three attributes of `imageAttributes` are added to the schema in addition. The data item transformation `extractCoordinatesFunc` defines how the values of the new attributes are computed on the basis of the values of the required attribute. Again, the order of attributes of `imageAttributes` does not affect matchmaking, but determines the order in which values of `extractCoordinatesFunc` are assigned to the attributes of the image.

```java
Attribute requiredAttribute = new Attribute("point", DataType.STRING)
                                  .setProperty("Encoding", "Well-Known-Text")
                                  .setProperty("Dimensions", "3");
Attribute[] imageAttributes = new Attribute[] {
        new Attribute("x", DataType.INTEGER),
        new Attribute("y", DataType.INTEGER),
        new Attribute("z", DataType.INTEGER),
};

SplitFunction extractCoordinatesFunc = new SplitFunction() {
    Pattern pattern = Pattern.compile("POINT\\(([\\d]+),([\\d]+),([\\d]+)\\)");
    Matcher matcher;
    @Override
    public Object[] apply(Object input) {
        matcher = pattern.matcher((String)input);
        matcher.find();
        return new Object[] {
                Integer.valueOf(matcher.group(1)),
                Integer.valueOf(matcher.group(2)),
                Integer.valueOf(matcher.group(3))
        };
    }
};

Split extractCoordinates = new Split("Extract coordinates", requiredAttribute,
                                         imageAttributes, extractCoordinatesFunc);
```

Listing B.4: Split in JEPC

```java
Attribute requiredAttribute = new Attribute("temperature", DataType.FLOAT)
                         .setProperty("Unit of measurement", "Degrees Fahrenheit");
Attribute imageAttribute = new Attribute("temperature", DataType.FLOAT)
                         .setProperty("Unit of measurement", "Degrees Kelvin");

ConversionFunction<Float,Float> fahrenheitToKelvinFunc =
  new ConversionFunction<Float, Float>() {
    @Override
    public Float apply(Float input) {
        return (input - 32) / 1.8f + 273.15f;
    }
};

Conversion fahrenheitToKelvin = new Conversion("Fahrenheit -> Kelvin",
                       requiredAttribute, imageAttribute, fahrenheitToKelvinFunc);
```

Listing B.5: Another conversion in JEPC

B.3.2 Combining and Adding Basic Transformations

As mentioned before, complex transformations cannot be achieved via a single basic transformation, but via a combination of multiple basic transformations. According to the motivating example, the sensor of milling machine 42 delivers temperature values in degrees Celsius, but Alice wants temperature values to be in degrees Fahrenheit and Bob needs temperature values in degrees Kelvin. We already created the transformation `celsiusToFahrenheit` for converting temperature values from degrees Celsius into degrees Fahrenheit. This transformation fits perfectly the needs of Alice, but not the needs of Bob. In the case of Bob, we have to create an additional transformation. We could create another transformation that converts the temperature values from degrees Celsius directly into degrees Kelvin. However, we decided to create another transformation that converts temperature values from degrees Fahrenheit into degrees Kelvin. It is shown in Listing B.5. Note that among all four basic transformations there is no single basic transformation that fits the needs of Bob. But when the two conversions are combined, temperature values can be converted from degrees Celsius (over degrees Fahrenheit) into degrees Kelvin.

After the definition of a basic transformation, it must be added to the matchmaker in order to enable it. Listing B.6 shows how to add all four basic transformations. For the rest of this chapter, we assume that all those transformations are active.

```
matchmaker.addTransformation(celsiusToFahrenheit);
matchmaker.addTransformation(addArea);
matchmaker.addTransformation(extractCoordinates);
matchmaker.addTransformation(fahrenheitToKelvin);
```

Listing B.6: Adding basic transformations

B.4 Creating Queries

In the traditional stream processing paradigm, data sources must be defined and registered before continuous queries can be created. This is due to the fact that every query definition must contain an exact specification of all queried data streams which are required to be already registered. Note that because of Algorithm 22 the order in which producers and consumers are registered does not matter when the matchmaker is used. In order to illustrate this flexibility, we decided to create the continuous queries of Alice and Bob before the registration of any external producer.

Listing B.7 shows the definition and creation of Alice's query that is an ordinary aggregation query. However, its input is not an existing data stream. Alice uses the match-operator to define a virtual data stream, whose schema is precisely specified before and corresponds to the individual view of Alice. Note that `setStaticValue`(x) is a shortcut for `setProperty("Static value", x)`. We provide this extra method, because the property with key `"Static value"` has a special meaning in our implementation (see Section 16.5 beginning on page 248). Listing B.8 shows the definition and creation of Bob's query that is similar to Alice's query. At present, we have two running queries without any external data producers being registered yet.

```java
Attribute[] aliceInput = new Attribute[] {
        new Attribute("temperature", DataType.FLOAT)
                .setProperty("Unit of measurement", "Degrees Fahrenheit"),
        new Attribute("mid",   DataType.SHORT)
                .setStaticValue(42)
};

EPA alice = new Aggregator(
        "Alice",
        new Match(aliceInput, new CountWindow(50)),
        new Average("temperature", "avgTemperature")
);

matchmaker.createQuery(alice);
```

Listing B.7: Query of Alice

```java
Attribute[] bobInput = new Attribute[] {
        new Attribute("place",   DataType.SHORT)
                .setStaticValue(10),
        new Attribute("temperature", DataType.FLOAT)
                .setProperty("Unit of measurement", "Degrees Kelvin")
};

EPA bob = new Aggregator(
        "Bob",
        new Match(bobInput, new CountWindow(50)),
        new Average("temperature", "avgTemperature")
);

matchmaker.createQuery(bob);
```

Listing B.8: Query of Bob

B.5 Registering External Producers

In contrast to the traditional paradigm in which a data stream abstracts from all individual data producers of the same type, individual data producers are registered separately in case of the matchmaker. The matchmaker needs the exact schema of an individual data producer being registered and returns a unique identifier for it.

```java
Attribute[] millingMachine42Schema = new Attribute[] {
        new Attribute("temperature", DataType.FLOAT)
                .setProperty("Unit of measurement", "Degrees Celsius")
                .setProperty("Minimum value", "-273.15"),
        new Attribute("consumption", DataType.INTEGER)
                .setProperty("Unit of measurement", "Watt")
                .setProperty("Minimum value", "0"),
        new Attribute("mid",   DataType.SHORT)
                .setProperty("Is key", "true")
                .setStaticValue(42),
        new Attribute("place",        DataType.SHORT)
                .setStaticValue("10")
                .setProperty("Is unique", "true"),
        new Attribute("type",  DataType.STRING)
                .setStaticValue("Milling machine")
};

int millingMachine42 = matchmaker.registerProducer(millingMachine42Schema);
```

Listing B.9: Registration of a producer

Listing B.9 shows the registration of the sensor of milling machine 42. We first define the individual schema of the producer being registered and remember the returned identification. It is needed to send data items, to update the schema of the registered producer and to unregister the producer. Immediately after registration, matchmaking is automatically performed and two new connections are established in total. One connection is established between the new producer and the query of Alice for the following reasons. The producer schema contains the requested attributes `mid` and `temperature`. Furthermore, `mid` has the requested static value and for converting temperature values from degrees Celsius into degrees Fahrenheit there is an active conversion. Another connection is established between the new producer and the query of Bob. The mapping of that connection is slightly different. It selects the requested attributes `temperature` as well as `place` and converts temperature values in degrees Kelvin as requested by Bob via the combination of the two active conversions. Other producers can be registered in the same way as shown in Listing B.9.

B.6 Processing of Events

At this point, we have created a minimum but working application on basis of the matchmaker. In order to make the established fine-grained connections with individual mappings visible, we push some data for the sensor of M42 in the following.

```java
int i = 0;
while(doSend)
    matchmaker.pushEvent(millingMachine42, new Object[] { 22.3f, 2_000 }, i++);
```

Listing B.10: Sending events

Listing B.10 simulates the sensor of M42 by using its identification for pushing events that have the constant payload `temperature=22.3f`, `consumption=2_000` and increasing timestamps. For all other attributes, the sensor must not send values since they have been declared to be static. They are added automatically to each event after arrival at the matchmaker. Note that `temperature` and `consumption` are variable in practice. In this example, we use constant values for illustration purposes.

```
Alice:
avgTemperature=72.13999938964844 tstart=0 tend=1
avgTemperature=72.13999938964844 tstart=1 tend=2
avgTemperature=72.13999938964844 tstart=2 tend=3
avgTemperature=72.13999938964844 tstart=3 tend=4
avgTemperature=72.13999938964844 tstart=4 tend=5
...

Bob:
avgTemperature=295.4499816894531 tstart=0 tend=1
avgTemperature=295.4499816894531 tstart=1 tend=2
avgTemperature=295.4499816894531 tstart=2 tend=3
avgTemperature=295.4499816894531 tstart=3 tend=4
avgTemperature=295.4499816894531 tstart=4 tend=5
...
```

Listing B.11: Output of queries

Listing B.11 shows the first events of the output streams of the queries of Alice and Bob. The fact that there exist output events proves that connections have been established. Furthermore, the different values of `avgTemperature` among Alice and Bob clearly show that the fine-grained connections have different mappings.

B.7 Runtime Adaptivity

Finally, we want to give an impression of the self-adaptivity of the example application that is achieved by using the matchmaker. For this purpose, we exchange the sensor of M42 for a different but still suitable (from the queries' perspective) sensor.

```
1  millingMachine42Schema = new Attribute[] {
2          new Attribute("place",        DataType.SHORT)
3                  .setStaticValue("10")
4                  .setProperty("Is unique", "true"),
5          new Attribute("mid",   DataType.SHORT)
6                  .setProperty("Is key", "true")
7                  .setStaticValue(42),
8          new Attribute("consumption", DataType.INTEGER)
9                  .setProperty("Unit of measurement", "Watt")
10                 .setProperty("Minimum value", "0"),
11         new Attribute("manufacturer", DataType.STRING)
12                  .setStaticValue("Gyro Gearloose"),
13         new Attribute("temperature", DataType.FLOAT)
14                 .setProperty("Unit of measurement", "Degrees Fahrenheit")
15                 .setProperty("Minimum value", "-459.67")
16 };
17
18 matchmaker.updateProducer(millingMachine42, millingMachine42Schema);
19
20 while(doSend)
21     matchmaker.pushEvent(millingMachine42, new Object[] { 2_000, 72.14f }, i++);
```

Listing B.12: Update of a producer

Listing B.12 shows the schema of the new sensor that replaces the sensor of M42. It still provides information about the place, the machine identification and the current temperature so that it is a suitable replacement source for the two queries. However, temperature values are now provided in degrees Fahrenheit instead of degrees Celsius. Moreover, there is no longer an attribute `type` but a new attribute `manufacturer`. Lastly, the order of attributes has completely changed. Nevertheless, after the schema has been updated at the matchmaker (line 18) both connections are re-established with new mappings. The new sensor seamlessly continues the processing by sending data items. Because the order of `temperature` and `consumption` has changed, the order of values in the payload must change too (also temperature values must now be in degrees Fahrenheit). Due to the new mappings, the input of both queries keeps exactly the same as before so that the payloads of output events do not change. In fact, the exchange of the sensor was completely hidden from the queries by the matchmaker.

Bibliography

[Aba03] Daniel J. Abadi, Don Carney, Ugur Çetintemel, Mitch Cherniack, Christian Convey, Sangdon Lee, Michael Stonebraker, Nesime Tatbul and Stan Zdonik: "Aurora: A New Model and Architecture for Data Stream Management".
In: *The VLDB Journal*, Volume 12, Issue 2, 2003, pp. 120–139.

[ABW03] Arvind Arasu, Shivnath Babu and Jennifer Widom:
"An Abstract Semantics and Concrete Language for Continuous Queries over Streams and Relations". In: *Proceedings of the International Conference on Data Base Programming Languages (DBPL)*, 2003, pp. 1–19.

[Agr08] Jagrati Agrawal, Yanlei Diao, Daniel Gyllstrom and Neil Immerman:
"Efficient Pattern Matching over Event Streams". In: *Proceedings of the International Conference on Management of Data (SIGMOD)*, 2008, pp. 147–160.

[Agu99] Marcos K. Aguilera, Robert E. Strom, Daniel C. Sturman, Mark Astley and Tushar D. Chandra:
"Matching Events in a Content-Based Subscription System".
In: *Proceedings of the Annual Symposium on Principles of Distributed Computing (PODC)*, 1999, pp. 53–61.

[AH00] Ron Avnur and Joseph M. Hellerstein:
"Eddies: Continuously Adaptive Query Processing". In: *Proceedings of the International Conference on Management of Data (SIGMOD)*, 2000, pp. 261–272.

[AIM10] Luigi Atzori, Antonio Iera and Giacomo Morabito:
"The Internet of Things: A Survey".
In: *Computer Networks*, Volume 54, Issue 15, 2010, pp. 2787–2805.

[AK80] Hussein M. Abdel-Wahab and Tiko Kameda: “On Strictly Optimal Schedules for the Cumulative Cost-Optimal Scheduling Problem”. In: *Computing*, Volume 24, Issue 1, 1980, pp. 61–86.

[Ali10] Mohamed Ali, Badrish Chandramouli, Balan Sethu Raman and Ed Katibah: “Spatio-Temporal Stream Processing in Microsoft StreamInsight”. In: *Data Engineering Bulletin*, Volume 33, Issue 2, 2010, pp. 69–74.

[All83] James F. Allen: “Maintaining Knowledge About Temporal Intervals”. In: *Communications of the ACM*, Volume 26, Issue 11, 1983, pp. 832–843.

[AM04] Arvind Arasu and Gurmeet Singh Manku: “Approximate Counts and Quantiles over Sliding Windows”. In: *Proceedings of the Symposium on Principles of Database Systems (PODS)*, 2004, pp. 286–296.

[And11] Lance Andersen: “JDBC 4.1 Specification”. Technical report. Oracle America, 2011.

[App12] H.-Jürgen Appelrath, Dennis Geesen, Marco Grawunder, Timo Michelsen and Daniela Nicklas: “Odysseus: A Highly Customizable Framework for Creating Efficient Event Stream Management Systems”. In: *Proceedings of the International Conference on Distributed Event-Based Systems (DEBS)*, 2012, pp. 367–368.

[Ari86] Gad Ariav: “A Temporally Oriented Data Model”. In: *Transactions on Database Systems (TODS)*, Volume 11, Issue 4, 1986, pp. 499–527.

[AW04] Arvind Arasu and Jennifer Widom: “Resource Sharing in Continuous Sliding-Window Aggregates”. In: *Proceedings of the International Conference on Very Large Data Bases (VLDB)*, 2004, pp. 336–347.

[Bab02] Brian Babcock, Shivnath Babu, Mayur Datar, Rajeev Motwani and Jennifer Widom: “Models and Issues in Data Stream Systems”. In: *Proceedings of the Symposium on Principles of Database Systems (PODS)*, 2002, pp. 1–16.

[Bal13] Cagri Balkesen, Nihal Dindar, Matthias Wetter and Nesime Tatbul: “RIP: Run-Based Intra-Query Parallelism for Scalable Complex Event Processing”. In: *Proceedings of the International Conference on Distributed Event-Based Systems (DEBS)*, 2013, pp. 3–14.

[Ban99] Guruduth Banavar, Tushar Chandra, Bodhi Mukherjee, Jay Nagarajarao, Robert E. Strom and Daniel C. Sturman: “An Efficient Multicast Protocol for Content-Based Publish-Subscribe Systems”. In: *Proceedings of the*

International Conference on Distributed Computing Systems (ICDCS), 1999, pp. 262–272.

[Bar07] Roger S. Barga, Jonathan Goldstein, Mohamed Ali and Mingsheng Hong: “Consistent Streaming Through Time: A Vision for Event Stream Processing”. In: *Proceedings of the Biennial Conference on Innovative Data Systems Research (CIDR)*, 2007, pp. 363–374.

[Bau15] Lars Baumgärtner, Christian Strack, Bastian Hoßbach, Marc Seidemann, Bernhard Seeger and Bernd Freisleben: “Complex Event Processing for Reactive Security Monitoring in Virtualized Computer Systems”. In: *Proceedings of the International Conference on Distributed Event-Based Systems (DEBS)*, 2015.

[BBC] BBC: “Plastic card fraud goes back up”. `http://news.bbc.co.uk/2/hi/business/7289856.stm` (visited on 10/15/2014).

[BDS00] Jochen van den Bercken, Jens-Peter Dittrich and Bernhard Seeger: “Javax.XXL: A Prototype for a Library of Query Processing Algorithms”. In: *Proceedings of the International Conference on Management of Data (SIGMOD)*, 2000, p. 588.

[Ber01] Jochen Van den Bercken, Björn Blohsfeld, Jens-Peter Dittrich, Jürgen Krämer, Tobias Schäfer, Martin Schneider and Bernhard Seeger: “XXL - A Library Approach to Supporting Efficient Implementations of Advanced Database Queries”. In: *Proceedings of the International Conference on Very Large Data Bases (VLDB)*, 2001, pp. 39–48.

[Ber08] Mark de Berg, Otfried Cheong, Marc van Kreveld and Mark Overmars: *Computational Geometry: Algorithms and Applications*. 3rd Edition, Springer-Verlag, 2008.

[Bet10] Claudio Bettini, Oliver Brdiczka, Karen Henricksen, Jadwiga Indulska, Daniela Nicklas, Anand Ranganathan and Daniele Riboni: “A Survey of Context Modelling and Reasoning Techniques”. In: *Pervasive and Mobile Computing*, Volume 6, Issue 2, 2010, pp. 161–180.

[BH06] Jürgen Beringer and Eyke Hüllermeier: “Online Clustering of Parallel Data Streams”. In: *Data and Knowledge Engineering*, Volume 58, Issue 2, 2006, pp. 180–204.

[BHL01] Tim Berners-Lee, James Hendler and Ora Lassila: “The Semantic Web”. In: *Scientific American*, Issue May, 2001.

[Bis06] Christopher M. Bishop: *Pattern Recognition and Machine Learning*. Springer-Verlag, 2006.

[Bis94] Gary Bishop, Henry Fuchs, Leonard McMillan and Ellen J. Scher Zagier: "Frameless Rendering: Double Buffering Considered Harmful". In: *Proceedings of the Annual Conference on Computer Graphics and Interactive Techniques (SIGGRAPH)*, 1994, pp. 175–176.

[BKT00] Peter Buneman, Sanjeev Khanna and Wang-chiew Tan: "Data Provenance: Some Basic Issues". In: *Conference on the Foundations of Software Technology and Theoretical Computer Science (FST TCS)*, 2000, pp. 87–93.

[Blo10] Marion Blount, Carolyn McGregor, Andrew James, Daby Sow, Rishikesan Kamaleswaran, Sascha Tuuha, Jennifer Percival and Nathan Percival: "On the Integration of an Artifact System and a Real-Time Healthcare Analytics System". In: *Proceedings of the International Health Informatics Symposium (IHI)*, 2010, pp. 647–655.

[BM72] Rudolf Bayer and Edward M. McCreight: "Organization and Maintenance of Large Ordered Indices". In: *Acta Informatica*, Volume 1, Issue 3, 1972, pp. 173–189.

[BN09] Jens Bleiholder and Felix Naumann: "Data Fusion". In: *ACM Computing Surveys*, Volume 41, Issue 1, 2009, 1:1–1:41.

[Bot09] Irina Botan, Younggoo Cho, Roozbeh Derakhshan, Nihal Dindar, Laura M. Haas, Kihong Kim, Chulwon Lee, Girish Mundada, Ming-Chien Shan, Nesime Tatbul, Ying Yan, Beomjin Yun and Jin Zhang: "Design and Implementation of the MaxStream Federated Stream Processing Architecture". Technical report. ETH Zürich, 2009.

[Bot10] Irina Botan, Younggoo Cho, Roozbeh Derakhshan, Nihal Dindar, Ankush Gupta, Laura M. Haas, Kihong Kim, Chulwon Lee, Girish Mundada, Ming-Chien Shan, Nesime Tatbul, Ying Yan, Beomjin Yun and Jin Zhang: "A Demonstration of the MaxStream Federated Stream Processing System". In: *Proceedings of the International Conference on Data Engineering (ICDE)*, 2010, pp. 1093–1096.

[BS03] Ahmet Bulut and Ambuj K. Singh: "SWAT: Hierarchical Stream Summarization in Large Networks". In: *Proceedings of the International Conference on Data Engineering (ICDE)*, 2003, pp. 303–314.

[BS11] Gurinder Singh Brar and Gagan Saini: "Milk Run Logistics: Literature Review and Directions". In: *Proceedings of the World Congress on Engineering (WCE)*, 2011.

[BSW97] Jochen Van den Bercken, Bernhard Seeger and Peter Widmayer: "A Generic Approach to Bulk Loading Multidimensional Index Structures". In: *Proceedings of the International Conference on Very Large Data Bases (VLDB)*, 1997, pp. 406–415.

[CBK09] Varun Chandola, Arindam Banerjee and Vipin Kumar: "Anomaly Detection: A Survey". In: *ACM Computing Surveys*, Volume 41, Issue 3, 2009, 15:1–15:58.

[CBN07] Eric Chu, Jennifer Beckmann and Jeffrey Naughton: "The Case for a Wide-Table Approach to Manage Sparse Relational Data Sets". In: *Proceedings of the International Conference on Management of Data (SIGMOD)*, 2007, pp. 821–832.

[Cha03] Sirish Chandrasekaran, Owen Cooper, Amol Deshpande, Michael J. Franklin, Joseph M. Hellerstein, Wei Hong, Sailesh Krishnamurthy, Samuel Madden, Vijayshankar Raman, Frederick Reiss and Mehul A. Shah: "TelegraphCQ: Continuous Dataflow Processing for an Uncertain World." In: *Proceedings of the Biennial Conference on Innovative Data Systems Research (CIDR)*, 2003.

[Chr11] Martin Christopher: *Logistics and Supply Chain Management*. 4th Edition, Financial Times Press, 2011.

[Cis14] Cisco Systems Inc.: "Cisco Visual Networking Index: Global Mobile Data Traffic Forecast Update, 2013–2018". Technical report. 2014.

[CM10] Gianpaolo Cugola and Alessandro Margara: "TESLA: A Formally Defined Event Specification Language". In: *Proceedings of the International Conference on Distributed Event-Based Systems (DEBS)*, 2010, pp. 50–61.

[CM12] Gianpaolo Cugola and Alessandro Margara: "Processing Flows of Information: From Data Stream to Complex Event Processing". In: *ACM Computing Surveys*, Volume 44, Issue 3, 2012, 15:1–15:62.

[CM95] Sophie Cluet and Guido Moerkotte: "On the Complexity of Generating Optimal Left-Deep Processing Trees with Cross Products". In: *Proceedings of the International Conference on Database Theory (ICDT)*, 1995, pp. 54–67.

[Cod70] Edgar F. Codd: "A Relational Model of Data for Large Shared Data Banks". In: *Communications of the ACM*, Volume 13, Issue 6, 1970, pp. 377–387.

[Com79] Douglas Comer: "The Ubiquitous B-Tree". In: *ACM Computing Surveys*, Volume 11, Issue 2, June 1979, pp. 121–137.

[Con04] Constant Data Inc.: "Managing the Costs of Downtime". Technical report. 2004.

[Cor09] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest and Clifford Stein: *Introduction to Algorithms*. 3rd Edition, The MIT Press, 2009.

[Cor12] Paulo Cortez, Miguel Rio, Miguel Rocha and Pedro Sousa: "Multi-Scale Internet Traffic Forecasting using Neural Networks and Time Series Methods". In: *Expert Systems*, Volume 29, Issue 2, 2012, pp. 143–155.

[CR87] James Clifford and Ahobala Rao: "A Simple, General Structure for Temporal Domains". In: *Proceedings of the Conference on Temporal Aspects in Information Systems*, 1987, pp. 17–28.

[Cra03] Chuck Cranor, Theodore Johnson, Oliver Spataschek and Vladislav Shkapenyuk: "Gigascope: A Stream Database for Network Applications". In: *Proceedings of the International Conference on Management of Data (SIGMOD)*, 2003, pp. 647–651.

[Dem07] Alan J. Demers, Johannes Gehrke, Biswanath Panda, Mirek Riedewald, Varun Sharma and Walker M. White: "Cayuga: A General Purpose Event Monitoring System". In: *Proceedings of the Biennial Conference on Innovative Data Systems Research (CIDR)*, 2007, pp. 412–422.

[Den87] Dorothy E. Denning: "An Intrusion-Detection Model". In: *Transactions on Software Engineering*, Volume 13, Issue 2, 1987, pp. 222–232.

[DeW94] David J. DeWitt, Navin Kabra, Jun Luo, Jignesh M. Patel and Jie-Bing Yu: "Client-Server Paradise". In: *Proceedings of the International Conference on Very Large Data Bases (VLDB)*, 1994, pp. 558–569.

[DG08] Jeffrey Dean and Sanjay Ghemawat: "MapReduce: Simplified Data Processing on Large Clusters". In: *Communications of the ACM*, Volume 51, Issue 1, 2008, pp. 107–113.

[DG92] David DeWitt and Jim Gray: "Parallel Database Systems: The Future of High Performance Database Systems". In: *Communications of the ACM*, Volume 35, Issue 6, 1992, pp. 85–98.

[DGK82] Umeshwar Dayal, Nathan Goodman and Randy H. Katz: "An Extended Relational Algebra with Control over Duplicate Elimination".

In: *Proceedings of the Symposium on Principles of Database Systems (PODS)*, 1982, pp. 117–123.

[DIG07] Yanlei Diao, Neil Immerman and Daniel Gyllstrom: "SASE+: An Agile Language for Kleene Closure over Event Streams". Technical report. University of Massachusetts, 2007.

[Dil03] Stephen Dill, Nadav Eiron, David Gibson, Daniel Gruhl, R. Guha, Anant Jhingran, Tapas Kanungo, Sridhar Rajagopalan, Andrew Tomkins, John A. Tomlin and Jason Y. Zien: "SemTag and Seeker: Bootstrapping the Semantic Web via Automated Semantic Annotation". In: *Proceedings of the International Conference on World Wide Web (WWW)*, 2003, pp. 178–186.

[Din13] Nihal Dindar, Nesime Tatbul, Renée J. Miller, Laura M. Haas and Irina Botan: "Modeling the Execution Semantics of Stream Processing Engines with SECRET". In: *The VLDB Journal*, Volume 22, Issue 4, 2013, pp. 421–446.

[DP73] David H. Douglas and Thomas K. Peucker: "Algorithms for the Reduction of the Number of Points Required to Represent a Line or Its Caricature". In: *The Canadian Cartographer*, Volume 10, Issue 2, 1973, pp. 112–122.

[DS01] Jens-Peter Dittrich and Bernhard Seeger: "GESS: A Scalable Similarity-Join Algorithm for Mining Large Data Sets in High Dimensional Spaces". In: *Proceedings of the International Conference on Knowledge Discovery and Data Mining (SIGKDD)*, 2001, pp. 47–56.

[EB09] Michael Eckert and Francois Bry: "Complex Event Processing (CEP)". In: *Datenbank Spektrum*, Volume 32, Issue 2, 2009, pp. 163–167.

[EN10] Opher Etzion and Peter Niblett: *Event Processing in Action*. 1st Edition, Manning Publications Co., 2010.

[Erw04] Martin Erwig: "Toward Spatio-Temporal Patterns". In: *Spatio-Temporal Databases* 2004, pp. 29–53.

[ES10] Brian S. Everitt and Anders Skrondal: *The Cambridge Dictionary of Statistics*. 4th Edition, Cambridge University Press, 2010.

[Esp] Esper. `http://www.espertech.com/esper/`.

[Etz10] Opher Etzion: "Event Processing: Past, Present and Future". In: *Proceedings of the VLDB Endowment (PVLDB)*, Volume 3, Issue 1-2, 2010, pp. 1651–1652.

[Eug03] Patrick Th. Eugster, Pascal A. Felber, Rachid Guerraoui and Anne-Marie Kermarrec: "The Many Faces of Publish/Subscribe". In: *ACM Computing Surveys*, Volume 35, Issue 2, 2003, pp. 114–131.

[Fab01] Françoise Fabret, Hans-Arno Jacobsen, François Llirbat, Joăo Pereira, Kenneth A. Ross and Dennis Shasha: "Filtering Algorithms and Implementation for Very Fast Publish/Subscribe Systems". In: *Proceedings of the International Conference on Management of Data (SIGMOD)*, 2001, pp. 115–126.

[FB74] Raphael A. Finkel and Jon Louis Bentley: "Quad Trees: A Data Structure for Retrieval on Composite Keys". In: *Acta Informatica*, Volume 4, 1974, pp. 1–9.

[Fie00] Roy T. Fielding: "Architectural Styles and the Design of Network-Based Software Architectures." PhD thesis. University of California, Irvine, 2000.

[For11] Catherine Forbes, Merran Evans, Nicholas Hastings and Brian Peacock: *Statistical Distributions*. 4th Edition, Wiley, 2011.

[FPS96] Usama Fayyad, Gregory Piatetsky-Shapiro and Padhraic Smyth: "From Data Mining to Knowledge Discovery in Databases". In: *AI Magazine*, Volume 17, Issue 3, 1996, pp. 37–54.

[Fre04] Elisabeth Freeman, Eric Freeman, Bert Bates and Kathy Sierra: *Head First Design Patterns*. O' Reilly & Associates, 2004.

[Fre14] Freescale: "What the Internet of Things (IoT) Needs to Become a Reality". Technical report. 2014.

[Gam95] Erich Gamma, Richard Helm, Ralph Johnson and John Vlissides: *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley Professional, 1995.

[Gar12] Gartner Inc.: "Effective Security Monitoring Requires Context". Technical report G00227893. 2012.

[GD07] Boris Glavic and Klaus R. Dittrich: "Data Provenance: A Categorization of Existing Approaches". In: *Datenbanksysteme für Business, Technologie und Web (BTW)*, 2007, pp. 227–241.

[Gei95] Kyle Geiger: *Inside ODBC*. Microsoft Press, 1995.

[Gen99] Jose Alvin G. Gendrano, Bruce C. Huang, Jim M. Rodrigue, Bongki Moon and Richard T. Snodgrass: "Parallel Algorithms for Computing Temporal Aggregates". In: *Proceedings of the International Conference on Data Engineering (ICDE)*, 1999, pp. 418–427.

[Geo] GeoTools - The Open Source Java GIS Toolkit. `http://geotools.org`.

[Gha07] Thanaa M. Ghanem, Moustafa A. Hammad, Mohamed F. Mokbel, Walid G. Aref and Ahmed K. Elmagarmid: "Incremental Evaluation of Sliding-Window Queries over Data Streams". In: *Transactions on Knowledge and Data Engineering (TKDE)*, Volume 19, Issue 1, 2007, pp. 57–72.

[Glo13] Nikolaus Glombiewski, Bastian Hoßbach, Andreas Morgen, Franz Ritter and Bernhard Seeger: "Event Processing on Your Own Database". In: *Datenbanksysteme für Business, Technologie und Web (BTW)*, 2013, pp. 33–42.

[GÖ03] Lukasz Golab and M. Tamer Özsu: "Issues in Data Stream Management". In: *SIGMOD Record*, Volume 32, Issue 2, 2003, pp. 5–14.

[GÖ10] Lukasz Golab and M. Tamer Özsu: *Data Stream Management (Synthesis Lectures on Data Management)*. Morgan and Claypool Publishers, 2010.

[Gor] Gorilla Factsheet. `http://seaworld.org/en/animal-info/animal-infobooks/gorilla/behavior/` (visited on 06/30/2014).

[GR83] Adele Goldberg and David Robson: *Smalltalk-80: The Language and Its Implementation*. Addison-Wesley, 1983.

[Gra06] Goetz Graefe: "B-Tree Indexes for High Update Rates". In: *SIGMOD Record*, Volume 35, Issue 1, 2006, pp. 39–44.

[Gra94] Jim Gray, Prakash Sundaresan, Susanne Englert, Ken Baclawski and Peter J. Weinberger: "Quickly Generating Billion-Record Synthetic Databases". In: *Proceedings of the International Conference on Management of Data (SIGMOD)*, 1994, pp. 243–252.

[Gro13] Katarina Grolinger, Wilson Higashino, Abhinav Tiwari and Miriam Capretz: "Data Management in Cloud Environments: NoSQL and NewSQL Data Stores". In: *Journal of Cloud Computing: Advances, Systems and Applications*, Volume 2, Issue 1, 2013, 22:1–22:24.

[GSC06] Jane Greenberg, Kristina Spurgin and Abe Crystal:
"Functionalities for Automatic Metadata Generation Applications: A Survey of Metadata Experts' Opinions". In: *International Journal of Metadata, Semantics and Ontologies*, Volume 1, Issue 1, 2006, pp. 3–20.

[Gul10] Vincenzo Gulisano, Ricardo Jimenez-Peris, Marta Patino-Martinez and Patrick Valduriez:
"StreamCloud: A Large Scale Data Streaming System". In: *Proceedings of the International Conference on Distributed Computing Systems (ICDCS)*, 2010, pp. 126–137.

[Gul12] Vincenzo Gulisano, Ricardo Jimenez-Peris, Marta Patino-Martinez, Claudio Soriente and Patrick Valduriez:
"StreamCloud: An Elastic and Scalable Data Streaming System".
In: *Transactions on Parallel and Distributed Systems (TPDS)*, Volume 23, Issue 12, 2012, pp. 2351–2365.

[Güt00] Ralf Hartmut Güting, Michael H. Böhlen, Martin Erwig, Christian S. Jensen, Nikos A. Lorentzos, Markus Schneider and Michalis Vazirgiannis:
"A Foundation for Representing and Querying Moving Objects".
In: *Transactions on Database Systems (TODS)*, Volume 25, Issue 1, 2000, pp. 1–42.

[Gut84] Antonin Guttman:
"R-Trees: A Dynamic Index Structure for Spatial Searching".
In: *Proceedings of the International Conference on Management of Data (SIGMOD)*, 1984, pp. 47–57.

[GUW08] Hector Garcia-Molina, Jeffrey D. Ullman and Jennifer Widom:
Database Systems: The Complete Book. 2nd Edition,
Prentice Hall Press, 2008.

[GV04] Janusz R. Getta and Ehsan Vossough:
"Optimization of Data Stream Processing".
In: *SIGMOD Record*, Volume 33, Issue 3, 2004, pp. 34–39.

[H2] H2 Database Engine. `http://www.h2database.com/`.

[Ham03] Moustafa Hammad, Walid Aref, Michael Franklin, Mohamed Mokbel and Ahmed Elmagarmid:
"Efficient Execution of Sliding-Window Queries Over Data Streams".
Technical report. Purdue University, 2003.

[Har08] Stavros Harizopoulos, Daniel J. Abadi, Samuel Madden and Michael Stonebraker: "OLTP Through the Looking Glass, and What We Found There". In: *Proceedings of the International Conference on Management of Data (SIGMOD)*, 2008, pp. 981–992.

[Hei08] Christoph Heinz, Jürgen Kramer, Tobias Riemenschneider and Bernhard Seeger: "Toward Simulation-Based Optimization in Data Stream Management Systems". In: *Proceedings of the International Conference on Data Engineering (ICDE)*, 2008, pp. 1580–1583.

[Hen09] Jim Hendler: "Web 3.0 Emerging". In: *IEEE Computer*, Volume 42, Issue 1, 2009, pp. 111–113.

[Her11] John R. Herring: "OpenGIS® Implementation Standard for Geographic Information - Simple Feature Access - Part 1: Common Architecture". Technical report OGC 06-103r4. Open Geospatial Consortium Inc., 2011.

[HFS12] Bastian Hoßbach, Bernd Freisleben and Bernhard Seeger: "Reaktives Cloud Monitoring mit Complex Event Processing". In: *Datenbank Spektrum*, Volume 12, Issue 1, 2012, pp. 33–42.

[HKS15] Bastian Hoßbach, Michael Körber and Bernhard Seeger: "Raumzeitliche Ereignisverarbeitung mit JEPC". In: *Geoinformationssysteme 2015: Beiträge zur 2. Münchner GI-Runde*, Wichmann-Verlag, 2015, pp. 60–69.

[HL97] David L. Hall and James Llinas: "An Introduction to Multisensor Data Fusion". In: *Proceedings of the IEEE*, Volume 85, Issue 1, 1997, pp. 6–23.

[HMU00] John E. Hopcroft, Rajeev Motwani and Jeffrey D. Ullman: *Introduction to Automata Theory, Languages, and Computation*. 2nd Edition, Pearson, 2000.

[Hoß11] Bastian Hoßbach: *Cloud Monitoring mit Complex Event Processing*. Diploma thesis. University of Marburg, 2011.

[Hoß13] Bastian Hoßbach, Nikolaus Glombiewski, Andreas Morgen, Franz Ritter and Bernhard Seeger: "JEPC: The Java Event Processing Connectivity". In: *Datenbank Spektrum*, Volume 13, Issue 3, 2013, pp. 167–178.

[HRP06] Hyoil Han, Han C. Ryoo and Herbert Patrick: "An Infrastructure of Stream Data Mining, Fusion and Management for Monitored Patients". In: *International Symposium on Computer-Based Medical Systems (CBMS)*, 2006, pp. 461–468.

[HS13] Bastian Hoßbach and Bernhard Seeger:
"Anomaly Management using Complex Event Processing".
In: *Proceedings of the International Conference on Extending Database Technology (EDBT)*, 2013, pp. 149–154.

[HSC02] Siegfried Handschuh, Steffen Staab and Fabio Ciravegna:
"S-CREAM - Semi-Automatic CREAtion of Metadata".
In: *Proceedings of the International Conference on Knowledge Engineering and Knowledge Management (EKAW)*, 2002, pp. 358–372.

[IK84] Toshihide Ibaraki and Tiko Kameda:
"On the Optimal Nesting Order for Computing N-Relational Joins".
In: *Transactions on Database Systems (TODS)*, Volume 9, Issue 3, 1984, pp. 482–502.

[Jai08] Namit Jain, Shailendra Mishra, Anand Srinivasan, Johannes Gehrke, Jennifer Widom, Hari Balakrishnan, Ugur Çetintemel, Mitch Cherniack, Richard Tibbetts and Stan Zdonik:
"Towards a Streaming SQL Standard". In: *Proceedings of the VLDB Endowment (PVLDB)*, Volume 1, Issue 2, 2008, pp. 1379–1390.

[Jes] Jesse Alpert and Nissan Hajaj: "We knew the Web was big ...".
`http://googleblog.blogspot.de/2008/07/we-knew-web-was-big.html` (visited on 04/09/2014).

[JS96] Christian S. Jensen and Richard T. Snodgrass:
"Semantics of Time-Varying Information".
In: *Information Systems*, Volume 21, Issue 4, 1996, pp. 311–352.

[JTS] JTS Topology Suite.
`http://www.vividsolutions.com/jts/JTSHome.htm`.

[Kal05] Satyen Kale, Elad Hazan, Fengyun Cao and Jaswinder Pal Singh:
"Analysis and Algorithms for Content-Based Event Matching".
In: *Proceedings of the International Conference on Distributed Computing Systems Workshops (ICDCSW)*, 2005, pp. 363–369.

[Kan08] Prathaben Kanagasingham: "Data Loss Prevention". Technical report. SANS Institute, 2008.

[Kar12] Tony Karlsson: "An Observational Study of the Characteristics of Taxi Floating Car Data Compared to Radar Sensor Data."
MA thesis. Royal Institute of Technology, 2012.

[KBZ86] Ravi Krishnamurthy, Haran Boral and Carlo Zaniolo:
"Optimization of Nonrecursive Queries". In: *Proceedings of the*

International Conference on Very Large Data Bases (VLDB), 1986, pp. 128–137.

[Kir04] Atanas Kiryakov, Borislav Popov, Ivan Terziev, Dimitar Manov and Damyan Ognyanoff: “Semantic Annotation, Indexing, and Retrieval”. In: *Journal of Web Semantics*, Volume 2, Issue 1, 2004, pp. 49–79.

[Krä06] Jürgen Krämer, Yin Yang, Michael Cammert, Bernhard Seeger and Dimitris Papadias: “Dynamic Plan Migration for Snapshot-Equivalent Continuous Queries in Data Stream Systems”. In: *Proceedings of the International Conference on Extending Database Technology (EDBT)*, 2006, pp. 497–516.

[Krä07] Jürgen Krämer: “Continuous Queries over Data Streams - Semantics and Implementation.” PhD thesis. University of Marburg, 2007.

[KS04] Jürgen Krämer and Bernhard Seeger: “PIPES: A Public Infrastructure for Processing and Exploring Streams”. In: *Proceedings of the International Conference on Management of Data (SIGMOD)*, 2004, pp. 925–926.

[KS05] Jürgen Krämer and Bernhard Seeger: “A Temporal Foundation for Continuous Queries over Data Streams”. In: *Proceedings of the International Conference on Management of Data (COMAD)*, 2005, pp. 70–82.

[KS09] Jürgen Krämer and Bernhard Seeger: “Semantics and Implementation of Continuous Sliding Window Queries over Data Streams”. In: *Transactions on Database Systems (TODS)*, Volume 34, Issue 1, 2009, 4:1–4:49.

[KS95] Nick Kline and Richard T. Snodgrass: “Computing Temporal Aggregates”. In: *Proceedings of the International Conference on Data Engineering (ICDE)*, 1995, pp. 222–231.

[Kum92] Vipin Kumar: “Algorithms for Constraint-Satisfaction Problems: A Survey”. In: *AI Magazine*, Volume 13, Issue 1, 1992, pp. 32–44.

[KWF06] Sailesh Krishnamurthy, Chung Wu and Michael Franklin: “On-the-Fly Sharing for Streamed Aggregation”. In: *Proceedings of the International Conference on Management of Data (SIGMOD)*, 2006, pp. 623–634.

[Läm07] Ralf Lämmel: “Google’s MapReduce Programming Model - Revisited”. In: *Science of Computer Programming*, Volume 68, Issue 3, 2007, pp. 208–237.

[Lam78] Leslie Lamport: “Time, Clocks, and the Ordering of Events in a Distributed System”. In: *Communications of the ACM*, Volume 21, Issue 7, 1978, pp. 558–565.

[Lar97] Per-Åke Larson: “Grouping and Duplicate Elimination: Benefits of Early Aggregation”. Technical report. Microsoft Research, 1997.

[Lee06] Yang W. Lee, Leo L. Pipino, James D. Funk and Richard Y. Wang: *Journey to Data Quality*. The MIT Press, 2006.

[LGP10] Morten Lindeberg, Vera Goebel and Thomas Plagemann: “Adaptive Sized Windows to Improve Real-Time Health Monitoring: A Case Study on Heart Attack Prediction”. In: *Proceedings of the International Conference on Multimedia Information Retrieval (MIR)*, 2010, pp. 459–468.

[LHB13] Harold Lim, Yuzhang Han and Shivnath Babu: “How to Fit when No One Size Fits”. In: *Proceedings of the Biennial Conference on Innovative Data Systems Research (CIDR)*, 2013.

[Li05] Jin Li, David Maier, Kristin Tufte, Vassilis Papadimos and Peter A. Tucker: “No Pane, No Gain: Efficient Evaluation of Sliding-Window Aggregates over Data Streams”. In: *SIGMOD Record*, Volume 34, Issue 1, 2005, pp. 39–44.

[Li07] Ming Li, Mo Liu, Luping Ding, Elke A. Rundensteiner and Murali Mani: “Event Stream Processing with Out-of-Order Data Arrival”. In: *Proceedings of the International Conference on Distributed Computing Systems Workshops (ICDCSW)*, 2007, p. 67.

[Li08] Jin Li, Kristin Tufte, Vladislav Shkapenyuk, Vassilis Papadimos, Theodore Johnson and David Maier: “Out-of-Order Processing: A New Architecture for High-Performance Stream Systems”. In: *Proceedings of the VLDB Endowment (PVLDB)*, Volume 1, Issue 1, 2008, pp. 274–288.

[Li11] Ming Li, Murali Mani, Elke A. Rundensteiner and Tao Lin: “Complex Event Pattern Detection over Streams with Interval-Based Temporal Semantics”. In: *Proceedings of the International Conference on Distributed Event-Based Systems (DEBS)*, 2011, pp. 291–302.

[Lin14] Tim Lindholm, Frank Yellin, Gilad Bracha and Alex Buckley: "The Java® Virtual Machine Specification (Java SE 8 Edition)". Technical report. Oracle America, 2014.

[LÖ09] Ling Liu and M. Tamer Özsu: *Encyclopedia of Database Systems*. 4th Edition, Springer Science+Business Media, 2009.

[Luc02] David C. Luckham: *The Power of Events: An Introduction to Complex Event Processing in Distributed Enterprise Systems*. 1st Edition, Addison-Wesley Professional, 2002.

[LWZ04] Yan-Nei Law, Haixun Wang and Carlo Zaniolo: "Query Languages and Data Models for Database Sequences and Data Streams". In: *Proceedings of the International Conference on Very Large Data Bases (VLDB)*, 2004, pp. 492–503.

[LWZ11] Yan-Nei Law, Haixun Wang and Carlo Zaniolo: "Relational Languages and Data Models for Continuous Queries on Sequences and Data Streams". In: *Transactions on Database Systems (TODS)*, Volume 36, Issue 2, 2011, 8:1–8:32.

[Mac08] Ashwin Machanavajjhala, Erik Vee, Minos Garofalakis and Jayavel Shanmugasundaram: "Scalable Ranked Publish/Subscribe". In: *Proceedings of the VLDB Endowment (PVLDB)*, Volume 1, Issue 1, 2008, pp. 451–462.

[McC02] Peter McCullagh: "What is a Statistical Model?" In: *The Annals of Statistics*, Volume 30, Issue 5, 2002, pp. 1225–1310.

[MD95] Manish Mehta and David J. DeWitt: "Managing Intra-Operator Parallelism in Parallel Database Systems". In: *Proceedings of the International Conference on Very Large Data Bases (VLDB)*, 1995, pp. 382–394.

[MHH00] Renée J. Miller, Laura M. Haas and Mauricio A. Hernández: "Schema Mapping as Query Discovery". In: *Proceedings of the International Conference on Very Large Data Bases (VLDB)*, 2000, pp. 77–88.

[Mic15] Microsoft Developer Network: "ODBC Programmer's Reference". Technical report. Microsoft, 2015.

[MLI00] Bongki Moon, Inés Fernando Vega López and Vijaykumar Immanuel: "Scalable Algorithms for Large Temporal Aggregation". In: *Proceedings of the International Conference on Data Engineering (ICDE)*, 2000, pp. 145–154.

[MM09] Yuan Mei and Samuel Madden: "ZStream: A Cost-Based Query Processor for Adaptively Detecting Composite Events". In: *Proceedings of* *the International Conference on Management of Data (SIGMOD)*, 2009, pp. 193–206.

[Moe09] Guido Moerkotte: "Building Query Compilers (Draft)". Technical report. University of Mannheim, 2009.

[Mon13] Olivier Monnier: "A Smarter Grid with the Internet of Things". Technical report. Texas Instruments, 2013.

[Moo79] Ramon E. Moore: *Methods and Applications of Interval Analysis*. Society for Industrial and Applied Mathematics, 1979.

[MR10] Oded Maimon and Lior Rokach: *Data Mining and Knowledge Discovery Handbook*. 2nd Edition, Springer-Verlag, 2010.

[MS79] Clyde L. Monma and Jeffrey B. Sidney: "Sequencing with Series-Parallel Precedence Constraints". In: *Mathematics of Operations Research*, Volume 4, Issue 3, 1979, pp. 215–224.

[NB95] Bernhard Nebel and Hans-Jürgen Bürckert: "Reasoning About Temporal Relations: A Maximal Tractable Subclass of Allen's Interval Algebra". In: *Journal of the ACM*, Volume 42, Issue 1, 1995, pp. 43–66.

[OSV11] Martin Odersky, Lex Spoon and Bill Venners: *Programming in Scala: A Comprehensive Step-by-Step Guide*. 2nd Edition, Artima Incorporation, 2011.

[Piy13] Rajeev Piyare: "Internet of Things: Ubiquitous Home Control and Monitoring System using Android Based Smart Phone". In: *International Journal of Internet of Things*, Volume 2, Issue 1, 2013, pp. 5–11.

[PLW02] Leo L. Pipino, Yang W. Lee and Richard Y. Wang: "Data Quality Assessment". In: *Communications of the ACM*, Volume 45, Issue 4, 2002, pp. 211–218.

[Pos] PostgreSQL. `http://www.postgresql.org/`.

[PS10] Kostas Patroumpas and Timos K Sellis: "Multi-Granular Time-Based Sliding Windows over Data Streams". In: *Proceedings of the International Symposium on Temporal Representation and Reasoning (TIME)*, 2010, pp. 146–153.

[RB01] Erhard Rahm and Philip A. Bernstein: "A Survey of Approaches to Automatic Schema Matching". In: *The VLDB Journal*, Volume 10, Issue 4, 2001, pp. 334–350.

[Ric12] Jeffrey Richter: *CLR via C#*. 4th Edition. Microsoft Press, 2012.

[Rie08] Tobias Riemenschneider: “Optimierung kontinuierlicher Anfragen auf Basis statistischer Metadaten.” PhD thesis. University of Marburg, 2008.

[RJ86] L. Rabiner and B.H. Juang: “An Introduction to Hidden Markov Models”. In: *IEEE ASSP Magazine*, Volume 3, Issue 1, 1986, pp. 4–16.

[Sak10] Mahmoud Attia Sakr: “Spatiotemporal Pattern Queries”. In: *VLDB 2010 PhD Workshop*, 2010, pp. 72–77.

[SÇ05] Michael Stonebraker and Ugur Çetintemel: “One Size Fits All: An Idea Whose Time Has Come and Gone”. In: *Proceedings of the International Conference on Data Engineering (ICDE)*, 2005, pp. 2–11.

[Sco92] David W. Scott: *Multivariate Density Estimation: Theory, Practice, and Visualization*. Wiley, 1992.

[SÇZ05] Michael Stonebraker, Ugur Çetintemel and Stan Zdonik: “The 8 Requirements of Real-Time Stream Processing”. In: *SIGMOD Record*, Volume 34, Issue 4, 2005, pp. 42–47.

[Sel79] Patricia G. Selinger, Morton M. Astrahan, Donald D. Chamberlin, Raymond A. Lorie and Thomas G. Price: “Access Path Selection in a Relational Database Management System”. In: *Proceedings of the International Conference on Management of Data (SIGMOD)*, 1979, pp. 23–34.

[SG11] Mahmoud Attia Sakr and Ralf Hartmut Güting: “Spatiotemporal Pattern Queries”. In: *GeoInformatica*, Volume 15, Issue 3, 2011, pp. 497–540.

[She11] Kyumars Sheykh Esmaili, Tahmineh Sanamrad, Peter M. Fischer and Nesime Tatbul: “Changing Flights in Mid-Air: A Model for Safely Modifying Continuous Queries”. In: *Proceedings of the International Conference on Management of Data (SIGMOD)*, 2011, pp. 613–624.

[SHG13] Scott Schneider, Martin Hirzel and Buğra Gedik: “Tutorial: Stream Processing Optimizations”. In: *Proceedings of the International Conference on Distributed Event-Based Systems (DEBS)*, 2013, pp. 249–258.

[SJ11] Mohammad Sadoghi and Hans-Arno Jacobsen: “BE-Tree: An Index Structure to Efficiently Match Boolean Expressions over High-Dimensional Discrete Space”. In: *Proceedings of the*

International Conference on Management of Data (SIGMOD), 2011, pp. 637–648.

[SJ13] Mohammad Sadoghi and Hans-Arno Jacobsen: "Analysis and Optimization for Boolean Expression Indexing". In: *Transactions on Database Systems (TODS)*, Volume 38, Issue 2, 2013, 8:1–8:47.

[SJS01] Giedrius Slivinskas, Christian S. Jensen and Richard Thomas Snodgrass: "A Foundation for Conventional and Temporal Query Optimization Addressing Duplicates and Ordering". In: *Transactions on Knowledge and Data Engineering (TKDE)*, Volume 13, Issue 1, 2001, pp. 21–49.

[SL04] Thomas Strang and Claudia Linnhoff-Popien: "A Context Modeling Survey". In: *Proceedings of the International Conference on Ubiquitous Computing (UbiComp)*, 2004.

[SL90] Amit P. Sheth and James A. Larson: "Federated Database Systems for Managing Distributed, Heterogeneous, and Autonomous Databases". In: *ACM Computing Surveys*, Volume 22, Issue 3, 1990, pp. 183–236.

[SMP09] Nicholas Poul Schultz-Møller, Matteo Migliavacca and Peter Pietzuch: "Distributed Complex Event Processing with Query Rewriting". In: *Proceedings of the International Conference on Distributed Event-Based Systems (DEBS)*, 2009, 4:1–4:12.

[Sno] Snort. `https://www.snort.org`.

[Sno87] Richard T. Snodgrass: "The Temporal Query Language TQuel". In: *Transactions on Database Systems (TODS)*, Volume 12, Issue 2, 1987, pp. 247–298.

[Sno92] Richard T. Snodgrass: "Temporal Databases". In: *Proceedings of the International Conference GIS - From Space to Territory: Theories and Methods of Spatio-Temporal Reasoning on Theories and Methods of Spatio-Temporal Reasoning in Geographic Space*, Springer-Verlag, 1992, pp. 22–64.

[Sow10] Daby Sow, Alain Biem, Marion Blount, Maria Ebling and Olivier Verscheure: "Body Sensor Data Processing Using Stream Computing". In: *Proceedings of the International Conference on Multimedia Information Retrieval (MIR)*, 2010, pp. 449–458.

[SPG05] Yogesh L. Simmhan, Beth Plale and Dennis Gannon: "A Survey of Data Provenance in E-Science". In: *SIGMOD Record*, Volume 34, Issue 3, 2005, pp. 31–36.

[SQL] SQL:2003 Part 2 (SQL/Foundation): *ISO/IEC 9075-2:2003*.

[Sto07a] Michael Stonebraker, Chuck Bear, Ugur Çetintemel, Mitch Cherniack, Tingjian Ge, Nabil Hachem, Stavros Harizopoulos, John Lifter, Jennie Rogers and Stanley B. Zdonik: "One Size Fits All? Part 2: Benchmarking Studies". In: *Proceedings of the Biennial Conference on Innovative Data Systems Research (CIDR)*, 2007, pp. 173–184.

[Sto07b] Michael Stonebraker, Samuel Madden, Daniel J. Abadi, Stavros Harizopoulos, Nabil Hachem and Pat Helland: "The End of an Architectural Era: (It's Time for a Complete Rewrite)". In: *Proceedings of the International Conference on Very Large Data Bases (VLDB)*, 2007, pp. 1150–1160.

[SW04] Utkarsh Srivastava and Jennifer Widom: "Flexible Time Management in Data Stream Systems". In: *Proceedings of the Symposium on Principles of Database Systems (PODS)*, 2004, pp. 263–274.

[Tan15] Kanat Tangwongsan, Martin Hirzel, Scott Schneider and Kun-Lung Wu: "General Incremental Sliding-Window Aggregation". In: *Proceedings of the VLDB Endowment (PVLDB)*, Volume 8, Issue 7, 2015, pp. 702–713.

[Tat10] Nesime Tatbul: "Streaming Data Integration: Challenges and Opportunities". In: *Proceedings of the International Conference on Data Engineering Workshops (ICDEW)*, 2010, pp. 155–158.

[Ter92] Douglas Terry, David Goldberg, David Nichols and Brian Oki: "Continuous Queries over Append-Only Databases". In: *Proceedings of the International Conference on Management of Data (SIGMOD)*, 1992, pp. 321–330.

[The14] The LLVM Development Team: "LLVM Language Reference Manual (Version 3.6)". Technical report. LLVM Project, 2014.

[TIB11] TIBCO: "Optimizing the Supply Chain Ecosystem". Technical report. 2011.

[TM11] Jens Teubner and Rene Mueller: "How Soccer Players Would Do Stream Joins". In: *Proceedings of the International Conference on Management of Data (SIGMOD)*, 2011, pp. 625–636.

[Tom96] David Toman: "Point vs. Interval-Based Query Languages for Temporal Databases (Extended Abstract)". In: *Proceedings of the Symposium on Principles of Database Systems (PODS)*, 1996, pp. 58–67.

[Tos14] Ankit Toshniwal, Siddarth Taneja, Amit Shukla, Karthik Ramasamy, Jignesh M. Patel, Sanjeev Kulkarni, Jason Jackson, Krishna Gade, Maosong Fu, Jake Donham, Nikunj Bhagat, Sailesh Mittal and Dmitriy Ryaboy: "Storm@Twitter". In: *Proceedings of the International Conference on Management of Data (SIGMOD)*, 2014, pp. 147–156.

[Tuc03] Peter A. Tucker, David Maier, Tim Sheard and Leonidas Fegaras: "Exploiting Punctuation Semantics in Continuous Data Streams". In: *Transactions on Knowledge and Data Engineering (TKDE)*, Volume 15, Issue 3, 2003, pp. 555–568.

[Twi] Twitter. `https://twitter.com`.

[Vaj11] Andrs Vajda: *Programming Many-Core Chips*. 1st Edition, Springer Publishing Company, 2011.

[Ven03] Suresh Venkatasubramanian: "The Graphics Card as a Stream Computer". In: *SIGMOD-DIMACS MPDS Workshop*, 2003, pp. 101–102.

[VF13] Ovidiu Vermesan and Peter Friess: *Internet of Things: Converging Technologies for Smart Environments and Integrated Ecosystems*. River Publishers, 2013.

[WDR06] Eugene Wu, Yanlei Diao and Shariq Rizvi: "High-Performance Complex Event Processing over Streams". In: *Proceedings of the International Conference on Management of Data (SIGMOD)*, 2006, pp. 407–418.

[web] webMethods Business Events. `http://www.softwareag-gov.com/products/webmethods/cep/`.

[WFH11] Ian H. Witten, Eibe Frank and Mark A. Hall: *Data Mining: Practical Machine Learning Tools and Techniques*. 3rd Edition, Morgan Kaufmann Publishers, 2011.

[Wha09] Steven Euijong Whang, Hector Garcia-Molina, Chad Brower, Jayavel Shanmugasundaram, Sergei Vassilvitskii, Erik Vee and Ramana Yerneni: "Indexing Boolean Expressions". In: *Proceedings of the VLDB Endowment (PVLDB)*, Volume 2, Issue 1, 2009, pp. 37–48.

[WTA10] Louis Woods, Jens Teubner and Gustavo Alonso: "Complex Event Detection at Wire Speed with FPGAs". In: *Proceedings of the VLDB Endowment (PVLDB)*, Volume 3, Issue 1, 2010, pp. 660–669.

[WWW11] Peng Wang, Haixun Wang and Wei Wang: "Finding Semantics in Time Series". In: *Proceedings of the International Conference on Management of Data (SIGMOD)*, 2011, pp. 385–396.

[Yan07] Yin Yang, Jürgen Krämer, Ieee Computer Society, Dimitris Papadias, Bernhard Seeger and Ieee Computer Society: "HybMig: A Hybrid Approach to Dynamic Plan Migration for Continuous Queries". In: *Transactions on Knowledge and Data Engineering (TKDE)*, Volume 19, Issue 3, 2007, p. 2007.

[YG94] Tak W. Yan and Héctor García-Molina: "Index Structures for Selective Dissemination of Information Under the Boolean Model". In: *Transactions on Database Systems (TODS)*, Volume 19, Issue 2, 1994, pp. 332–364.

[YK97] Xinfeng Ye and John A. Keane: "Processing Temporal Aggregates in Parallel". In: *Proceedings of the International Conference on Systems, Man and Cybernetics (SMC)*, 1997, pp. 1373–1378.

[YW03] Jun Yang and Jennifer Widom: "Incremental Computation and Maintenance of Temporal Aggregates". In: *The VLDB Journal*, Volume 12, Issue 3, 2003, pp. 262–283.

[ZCT14] Dongxiang Zhang, Chee-Yong Chan and Kian-Lee Tan: "An Efficient Publish/Subscribe Index for E-Commerce Databases". In: *Proceedings of the VLDB Endowment (PVLDB)*, Volume 7, Issue 8, 2014, pp. 613–624.

[Zem07] Fred Zemke, Andrew Witkowski, Mitch Cherniak and Latha Colby: "Pattern Matching in Sequences of Rows". Technical report. ANSI Standard Proposal, 2007.

[ZM06] Justin Zobel and Alistair Moffat: "Inverted Files for Text Search Engines". In: *ACM Computing Surveys*, Volume 38, Issue 2, 2006.

[ZRH04] Yali Zhu, Elke A. Rundensteiner and George T. Heineman: "Dynamic Plan Migration for Continuous Queries over Data Streams". In: *Proceedings of the International Conference on Management of Data (SIGMOD)*, 2004, pp. 431–442.

[ZS02] Yunyue Zhu and Dennis Shasha: “StatStream: Statistical Monitoring of Thousands of Data Streams in Real Time”. In: *Proceedings of the International Conference on Very Large Data Bases (VLDB)*, 2002, pp. 358–369.

List of Acronyms

AIT	Active Instances Table
API	Application Programming Interface
ASI	Adjacent Sequence Interchange
BE	Boolean Expression
BT	Buffer Table
CEP	Complex Event Processing
CLR	Common Language Runtime
CPU	Central Processing Unit
CQ	Continuous Query
DBMS	Database Management System
DDoS	Distributed Denial-of-Service
DEP	Dynamic Event Processing
DNF	Disjunctive Normal Form
DPM	Dynamic Plan Migration
DSMS	Data Stream Management System
EP	Event Processing
EPA	Event Processing Agent
EPL	Event Processing Language
EPN	Event Processing Network
EPP	Event Processing Provider
FHSN	Filter Handle Set Node
FPGA	Field-Programmable Gate Array
GC	Garbage Collection
GESS	Generic External Space Sweep

GHz	Gigahertz
GiB	Gibibyte
GPS	Global Positioning System
GPU	Graphics Processing Unit
GSL	Grid Split Limit
HMM	Hidden Markov Model
HTTP	Hypertext Transfer Protocol
I/O	Input/Output
IDS	Intrusion Detection System
IoT	Internet of Things
IP	Internet Protocol
ISO	International Organization for Standardization
ISP	Internet Service Provider
IT	Information Technology
JDBC	Java Database Connectivity
JEPC	Java Event Processing Connectivity
JRE	Java Runtime Environment
JVM	Java Virtual Machine
KiB	Kibibyte
KPI	Key Performance Indicator
LLVM	Low Level Virtual Machine
M42	Milling Machine 42
MiB	Mebibyte
MRQ	Match-Recognize Queries
NFA	Nondeterministic Finite Automaton
ODBC	Open Database Connectivity
OGC	Open Geospatial Consortium
PAT	Potentially Applicable Transformation
PDF	Probability Density Function
PIPES	Public Infrastructure for Processing and Exploring Streams
PNA	Positive-Negative Approach
QoS	Quality of Service
RDF	Resource Description Framework
REST	Representational State Transfer
RNG	Random Number Generator
SFA	Simple Feature Access

SoI	Situation of Interest
SPE	Stream Processing Engine
SQL	Structured Query Language
STPQ	Spatiotemporal Pattern Queries
TC	Temporal Constraint
TCL	Temporal Constraint List
TCP	Transmission Control Protocol
TP	Temporal Pattern
TPL	Temporal Pattern List
UEPA	User-Defined EPA
VM	Virtual Machine
WKT	Well-Known Text
XML	eXtensible Markup Language
XXL	eXtensible and fleXible Library

List of Algorithms

List of Figures

List of Listings

List of Tables

Curriculum Vitae

Personal Information

Name	Bastian Hoßbach
Date of Birth	December 18, 1985
Place of Birth	Hessisch-Lichtenau (Germany)

Education

2006 – 2011	*Diploma in Computer Science* University of Marburg (Germany)
2003 – 2006	*Abitur (Higher Education Entrance Qualification)* Oberstufengymnasium Eschwege (Germany)

Work Experience

2012 – 2015	*Research and Teaching Assistant* BMBF-funded project "Anomaly Management in Computer Systems via Complex Event Processing Technology", University of Marburg (Germany)
2010 – 2012	*Student Assistant* Database Research Group, University of Marburg (Germany)
2010	*Intern* Software development, RTM Realtime Monitoring GmbH (Germany)
2005 – 2012	*IT Freelancer* Design and implementation of webpages and web apps, Various companies