State Management for Efficient Event Pattern Detection

Dissertation

zur Erlangung des akademischen Grades

doctor rerum naturalium (Dr. rer. nat.)
im Fach Informatik

eingereicht an der
Mathematisch-Naturwissenschaftlichen Fakultät
der Humboldt-Universität zu Berlin

von
Bo Zhao, M. Eng.

Präsident (komm.) der Humboldt-Universität zu Berlin:
Prof. Dr. Peter Frensch

Dekan der Mathematisch-Naturwissenschaftlichen Fakultät:
Prof. Dr. Elmar Kulke

Gutachter/innen:	1. Prof. Dr. Matthias Weidlich
	2. Prof. Dr. Prof. Dr. Kurt Rothermel
	3. Prof. Prof. Dr. Boris Koldehofe
Tag der mündlichen Prüfung:	1. März 2022

Abstract

Systems for event stream processing continuously evaluate queries over high-velocity event streams to detect user-specified patterns with low latency. Since the patterns become less important over time, it is crucial to detect them as quickly as possible. However, low-latency pattern detection is challenging, because query processing is stateful and the set of partial matches maintained by common algorithms for query evaluation grows exponentially in the size of the processed event data.

Handling this state during query evaluation is further complicated by the dynamicity of streams and the potential need to integrate remote data. First, heterogeneous event sources may yield streams with dynamic, unpredictable input rates and data distributions, and hence, query selectivities. During short peak times, exhaustive processing is no longer reasonable, or even infeasible, and systems shall resort to best-effort query evaluation: They shall strive for optimal result quality while staying within a latency bound. Second, some queries require access to remote data from external sources to determine whether a specific event is part of a pattern. Such dependencies are problematic, since waiting for remote data to be fetched interrupts the evaluation of queries over the streams. Yet, without event selection based on remote data, the growth of the number of partial matches maintained during query evaluation is amplified.

In this dissertation, we present strategies for optimised state management in event pattern detection. First, we enable best-effort query evaluation with load shedding that discards both input events and partial matches. We carefully select the partial matches and input events to drop in order to satisfy a latency bound while striving for a minimal loss in result quality. Second, to efficiently integrate remote data, we decouple the fetching of remote data from its use in query evaluation through a caching mechanism. Based thereon, we hide the transmission latency of remote data by prefetching data based on anticipated use and by lazy evaluation that postpones the event selection based on remote data to avoid interruptions. A cost model is proposed to determine when to fetch which remote data items and how long to keep them in the cache.

We evaluated our techniques for load shedding and the integration of remote data with queries over synthetic and real-world event data. We show that our load shedding technique significantly improves the recall of pattern detection over baseline approaches under different latency bounds, while our technique for remote data integration can drastically reduce the detection latency. Moreover, we report on a case study on smart grid management where event stream processing is employed to alleviate stress on the grid during peak demand hours. This is achieved by monitoring how consumers alter their consumption as requested by a utility, and by predicting potential non-compliance in real-time. Our simulation results show that load shedding and remote data integration enable the design of a system that scales up to 1.6 million residents.

ZUSAMMENFASSUNG

Systeme zur Event Stream Processing überwachen kontinuierliche Datenströme, um nutzerdefinierte Patterns zu detektieren. Diese Patterns sollen schnellstmöglich erkannt werden, um die Latenz einer Reaktion auf ein Pattern zu minimieren. Dies ist sehr herausfordernd, weil die Verarbeitung der entsprechenden Queries zustandsbasiert erfolgt und die Größe des Zustands durch partielle Patternübereinstimmungen exponentiell wächst.

Die Handhabung dieser Zustände während der Evaluation einer Query wird zusätzlich durch die möglicherweise existierende Notwendigkeit Remote-Daten für die Evaluation heranzuziehen verkompliziert. Als weiteres Hindernis sorgen heterogene Datenstrom-Quellen für sich dynamisch ändernde Datenströme mit unvorhersehbaren Inputraten und Datenverteilungen. Entsprechend variieren die Query Selektivitäten. Während kurzer Peaks ist eine vollumfassende Query Evaluation nicht praktikabel und teilweise unmöglich. Somit wird eine best-effort Query Evaluation notwendig, welche eine optimale Query Evaluation anstrebt, bei gleichzeitiger Einhaltung von Latenzbedingungen. Desweiteren erfordern einige Queries die Abfrage von Remote-Daten um die Relevanz einzelner Events für ein Pattern zu bestimmen. Diese Abhängigkeit ist problematisch, weil die Wartezeit, welche beim Zugriff auf Remote-Daten entsteht, die Evaluation der Queries verzögert. Dennoch ist eine solche Überprüfung notwendig, weil sonst die Anzahl der partiellen Patternübereinstimmungen weiter wachsen würde.

In der vorliegenden Disseration werden verschiedene Strategien für das optimierte Zustandsmanagement während der Event Pattern Detection vorgestellt. Als erstes wird ein Ansatz für die Best-effort Query Evaluation mittels Load Shedding vorgestellt, welcher sowohl partielle Patternübereinstimmungen, als auch Input Events selektiert. Diese Selektion erfolgt mit den Ziel der Einhaltung einer Latenzgrenze, bei gleichzeitiger Minimierung des Qualitätsverlustes bei der Query Evaluation. Außerdem stellen wir eine Strategie zur Integration von Remote-Daten vor, welche Abfragen jener von der Query-Evaluation durch einen Caching Mechanismus entkoppelt. So können Übertragungslatenzen bei der Abfrage von Remote-Daten mittels Prefetching und Lazy Evaluation Techniken vermieden werden. Mit Hilfe eines Kostenmodells wird bestimmt, welche Remote-Daten wann abgerufen werden sollen und wie lange sie zwischengespeichert werden müssen.

Die Evaluation unserer Ansätze für Load Shedding und die Integration von Remote-Daten erfolgt auf synthetischen und real-world Datensätzen. Dabei wird aufgezeigt, dass die in dieser Arbeit vorgestellten Ansätze eine signifikante Verbesserung gegenüber dem Stand der Technik ist. Beispielsweise kann der Ansatz für die Remote-Daten Integration die Detektionslatenz drastisch reduzieren. Außerdem wird eine Fallstudie aus dem Smart Grid Bereich vorgestellt, welche einen Anwendungsfall illustriert, bei dem mit Event Stream Processing Lastspitzen abgeschwächt werden. Im Rahmen der Fallstudie kann aufgezeigt werden, wie unsere Optimierungstrategien diesen Ansatz für ein System mit bis zu 1,6 Millionen Konsumenten skalieren.

Acknowledgements

First of all, I want to thank my supervisor, Prof. Matthias Weidlich, who supported me throughout my PhD journey, gave me the full freedom to explore research ideas and constantly sharpened my research skills such as analytical thinking. Prof. Weidlich is also a role model for me. His politeness, patience, friendliness, and the ability to balance the work and family commitments, taught me characters beyond academic research.

Thank you to my collaborators, Prof. Han van der Aa, Dr. Nguyen Quoc Viet Hung, and Dr. Nguyen Thanh Tam, with whom I am always enjoying the discussion about research ideas and casual chats, not to mention happy coffee and beer time. I also thank Dr. Gururaghav Raman and Prof. Jimmy Chih-Hsien Peng for helping me apply the research ideas on efficient event stream processing to smart grid management.

Thank you to my lovely colleagues and friends, Samira Akili, Stephan Fahrenkrog-Petersen, Galina Greil, Simon Heiden, Martin Kabierski, Yannic Noller, Robert Prüfer, Hermann Stolte, Jan Sürmeli, Marvin Triebel, and Kim Völlinger, with whom I enjoyed the lunchtime, the coffee break, the badminton play, birthday parties, the movie nights, and casual chats, among other events. This really helped me as an expatriate living in Germany to continue my PhD journey.

Thank you to my Chinese friends, Jinchun Chi, Wuqi Guo, Lei Sun, Chao Wang, Xifan Wang, Linyan Zhu, and Hui Zhu. We had some fruitful discussions and wonderful times. Hanging out together alleviates my occasional homesickness. A special thank you goes to Yanmeng Liu, who spent quite some time on the proofreading of this dissertation.

Last but not least, I would like to thank my family for their endless love, especially during the COVID-19 pandemic. They shared my ups and downs, supported my decisions, and gave me motivation and confidence to continue this path. Thank my parents for raising me up to more than I can be.

ERKLÄRUNG / DECLARATION

Hiermit erkläre ich, die Dissertation selbstständig und nur unter Verwendung der angegebenen Hilfen und Hilfsmittel angefertigt zu haben. Ich habe mich nicht anderwärts um einen Doktorgrad in dem Promotionsfach beworben und besitze keinen entsprechenden Doktorgrad. Die Promotionsordnung der Mathematisch-Naturwissenschaftlichen Fakultät, veröffentlicht im Amtlichen Mitteilungsblatt der Humboldt-Universität zu Berlin Nr. 42 am 11. Juli 2018, habe ich zur Kenntnis genommen.

I declare that I have completed the thesis independently using only the aids and tools specified. I have not applied for a doctor's degree in the doctoral subject elsewhere and do not hold a corresponding doctor's degree. I have taken due note of the Faculty of Mathematics and Natural Sciences PhD Regulations, published in the Official Gazette of Humboldt-Universität zu Berlin no. 42 on July 11, 2018.

Berlin, Germany, September 2021

SIGNATURE: .....

Table of Contents

List of Tables

List of Figures

CHAPTER 1

INTRODUCTION

Event stream processing (ESP) is a computing paradigm for processing continuous *streams* of *event* data to extract semantic insights [61]. An event is an instantaneous, unique, and atomic occurrence of interest at a point in time and an event stream is an unbounded sequence of events [70]. By analysing correlations and causal relations among events, ESP provides higher-level semantic abstractions compared to events themselves, thus supporting decisions and triggering actions in real-world applications. Unlike traditional database management systems (DBMS) and recent batch-based big data processing platforms that process static data sets, ESP enables *low-latency* analytics over event streams, which allows actions to be taken as soon as a situation of interest materialises.

This chapter discusses the motivation of this dissertation (Section 1.1). Based thereon, we present the research problem addressed in this dissertation (Section 1.2), the overview of our research method and the specific contributions (Section 1.3). Some of these contributions have been published as peer-reviewed scientific papers as summarised in Section 1.4. Lastly, we sketch the dissertation's outline in Section 1.5.

1.1 Motivation

Event stream processing (ESP) engines continuously evaluate user-defined queries over event stream data from different sources and applications, as shown in Figure 1.1. Here, events are represented by different geometric shapes. ESP queries define sequence patterns of multiple events that follow a strict temporal order. For instance, in Figure 1.1, query Q1 defines a sequence pattern of three events, such that a rectangle event must be followed by a triangle event, which is again followed by another rectangle event. On top of sequence patterns, ESP queries define

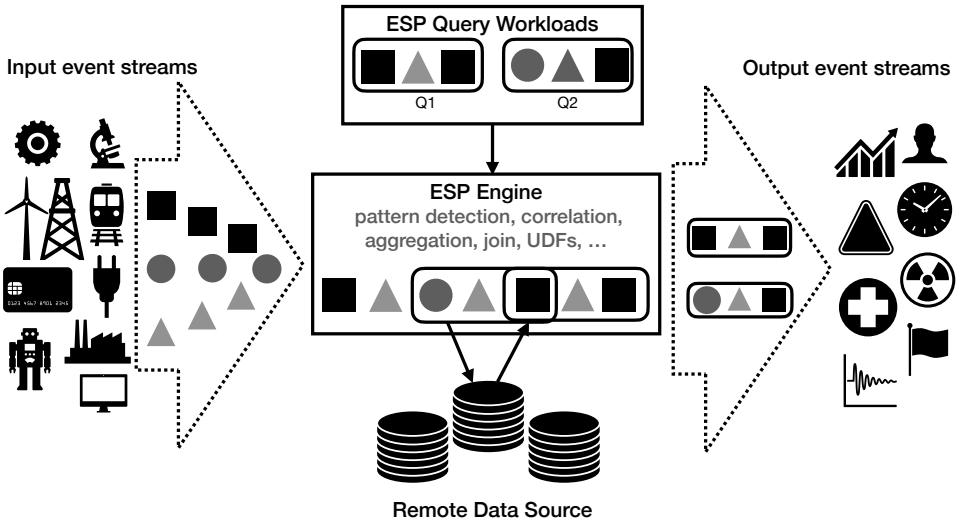


Figure 1.1: Event stream processing framework

value predicates of events, including aggregations and user-defined functions. In addition, value predicates may also correlate events with data that does not reside in the ESP engine due to privacy concerns or implementation considerations. As a result, ESP engines have to fetch the required data before evaluating related value predicates. We refer to such data as originating from remote data sources. Event streams are generally unbounded and common real-world applications focus on sequence patterns happening in a certain time duration. Therefore, an ESP query defines a *time window*: The sequence of events that satisfies the pattern and all value predicates must happen within a time interval specified by the time window.

Without loss of generality, we take an example of fraud detection in credit card transactions to illustrate aspects of ESP. Assuming that the triangles and rectangles in Figure 1.1 represent credit card transaction events, query Q1 detects fraudulent transactions as follows. Suppose that three time-ordered transactions (sequence pattern) of the same account (value predicate) happen in different locations (value predicate) within 20 minutes (time window). In this case, an ESP engine looks up data from remote data sources that store location and transportation information. If the above three locations cannot be reached within 20 minutes according to remote data sources (value predicates on remote data), the ESP engine reports a fraudulent transaction.

An ESP engine evaluates queries event by event in an online fashion. This means that the ESP engine must process an event at the moment when it arrives, instead of batching a set of events and then processing them later. An even kept longer than a time window is discarded permanently. Once a pattern has been detected, the ESP engine immediately materialises it and generates a corresponding *complex event* in one or more output streams, as shown in Figure 1.1. Each complex event consists of multiple input events and has higher-level semantics than an individual input event, enabling humans or automation systems to take further actions.

Query processing incurs *latency*—the time delay between the appearance of all input events needed to materialise a pattern in an ESP engine and the actual detection of that pattern. For a certain complex event, the query processing latency consists of the latency to evaluate the pattern (*pattern evaluation latency*) and the latency to look up remote data sources (*remote data fetching latency*). ESP engines aim at low-latency processing. For instance, in the context of fraudulent transaction detection, the sooner a fraud is detected, the smaller financial losses the credit card company bears.

Having explained the main aspects of ESP, we turn to a more detailed discussion of its essential properties.

ESP is becoming ubiquitous.

Due to the unique characteristics of ESP, researchers have focused on designing and implementing efficient ESP engines and optimisations to cope with the increasing growth of streaming data that needs to be analysed in science, engineering, and business scenarios, see the survey [64]. Commercial ESP software products are also offered by leading industry enterprises including Microsoft [43], Google [6], Amazon [8], IBM [82], SAP [198], and Oracle [188]. ESP has emerged from niche applications such as algorithmic trading to broader markets including health monitoring, smart grids, logistics, telecommunication, energy, manufacturing, supply chain management, and transportation. Nowadays, we are in the big data era with rapid deployments of 5G networks and internet of things (IoT) devices around the world in both industrial and consumer fields. High-speed data streams are constantly generated and need to be analysed in real time, which leads to various real-world applications of ESP:

- **Algorithmic trading.** Trading companies define complex patterns that trigger trading algorithms for high-frequency trading (HFT) in stock markets. HFT processes a large number of orders in fractions of a second. The quickest reactions to certain patterns are the most profitable.
- **Urban transportation.** Ride-hailing services may specify patterns that correlate user requests, nearby available vehicles, and drivers' responses to match costumers and drivers.
- **Security systems.** Sophisticated access control systems define patterns of CCTV camera footages and RFID badges in correlation with access permissions for employees and visitors to detect unauthorised intrusions.
- **Internet of Things (IoT)**. Fire alarm systems define patterns of measurements from smoke sensors, temperature sensors, and humidity sensors to detect fire alarms. Modern manufacturing companies define patterns of various sensor readings from Industry Internet of Things (IIoT) devices to improve automation efficiency and detect failures.

Despite the different semantics and functionalities, the above applications have in common

that they require low-latency processing. For instance, ride-hailing services aim at sub-second latency for smooth user experience [33]. Credit card fraud detection enforces a strict latency bound of 25ms [71]. HFT systems even require a latency below single-digit microseconds [103].

Common ESP queries for pattern detection are *stateful*.

ESP, in general, may involve operations ranging from filtering and aggregating event data to complex pattern detection and sophisticated mathematical algorithms including machine learning. However, pattern detection queries are commonly *stateful*, which means that an ESP engine processes each input event in a way that previous events influence how the current event is processed. To this end, an ESP engine has to maintain partially processed results of previous events (*e.g.*, partial matches of pattern detection): When an ESP engine receives an input event, it processes that event together with the maintained partial results and generates new partial results. These new partial results that satisfy the complete query (sequence pattern, value predicates and time window) denote the output of ESP, *i.e.*, the complex events. In this dissertation, we refer to partial results at a specific time as *state*. A single partial result is denoted as a *state element*. Therefore, the ESP engine, at runtime, can be viewed as an infinite sequence of evolving state.

State affects pattern evaluation latency.

State impacts on the efficiency of ESP engines, particularly in latency performance. It affects pattern evaluation latency—the computational overhead of processing state elements. The number of state elements can be significantly large due to expensive operators such as *Kleene closure* that detects arbitrarily long sequence patterns. In the worst case, the number of state elements grows exponentially in the number of processed events [196], which is unbounded. Moreover, state changes dynamically with input event streams [202]. Analysing the fast-growing state increases the processing latency and decreases the throughput of ESP engines.

Stateful pattern detection operations render parallelisation schemes non-trivial or even impossible, because of data dependencies between state elements and input events. In addition, it is difficult or even impractical for parallelisation schemes to predict how many computing resources to allocate beforehand. Elastically scaling on demand, on the other hand, interrupts stateful processing until additional computing resources have been successfully acquired. Besides, state migration incurs additional overhead.

When the computational overhead of processing state elements is beyond the computing capacity of an ESP engine, the ESP engine is in an *overload situation* with increasing processing latency. In the worst case, the ESP engine is not able to deliver any query results at all. This

makes low-latency processing challenging.

State affects remote data fetching latency.

State also impacts remote data fetching latency—the overhead for fetching remote data to ESP engines as part of query processing, due to both privacy concerns and implementation considerations:

- State of an ESP engine may be stored in persistent databases for failure recovery.
- Large number of state elements may not completely fit in the ESP engine. Therefore, part of them need to be temporarily moved to disks or external databases.
- Data regulations require sensitive data to be decoupled from ESP engines. For example, the general data protection regulation (GDPR) in the European Union (EU) enforces strict constraints about how organisations outside the EU can access its residents' personal data.

Nonetheless, fetching remote data results in data being transmitted along network connections and therefore, significantly increases the overall processing latency. This latency of data transmission is referred to as *remote data transmission latency* and it is part of the remote data fetching latency. For a certain pattern detection result, its remote data transmission latency is usually at least one order of magnitude higher than its pattern evaluation latency [203].

State elements and input events govern what remote data is required for query processing. However, the dynamics of input streams and state elements in ESP engines lead to unpredictable and irregular access patterns of looking up remote data sources. This makes prefetching and caching the required remote data challenging.

All the above issues deteriorate an ESP engine's performance and are caused by insufficient state management. Efficient event pattern detection requires new perspectives and optimisation techniques on state management.

1.2 Research Problem

This dissertation addresses the problem of low-latency pattern detection over event streams.

As discussed in Section 1.1, ESP query processing incurs latency that consists of pattern evaluation latency and remote data fetching latency. For real-world applications, the importance of detected patterns decreases with the increasing processing latency, as illustrated in Figure 1.2. Above a certain latency bound, the query results may become useless. In this dissertation, we aim at efficient pattern detection over event streams with low processing latency. Specifically, we solve the following research problems:

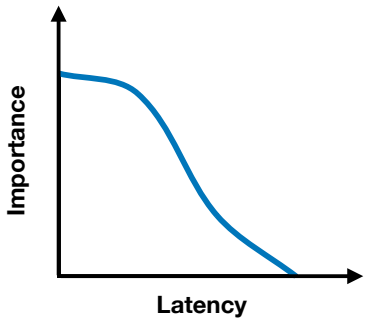


Figure 1.2: Importance vs latency.

1.3 Overview of the research contributions

This dissertation achieves low-latency event pattern detection by efficient state management.

We argue that the two research problems in Section 1.2 can be solved by efficient state management. As a foundation, we provide insights into the pivotal role of state management in efficient event pattern detection over event streams. Based thereon, we propose a hybrid load shedding technique that discards both input events and state elements during query processing to reduce pattern evaluation latency (**R1**). As to the problem of reducing remote data fetching latency (**R2**), we propose the EIRES framework that holistically combines prefetching, lazy evaluation, and caching techniques for remote data based on state management. In this dissertation, we make the following specific contributions:

- **Input-based and state-based load shedding techniques.** We present a hybrid load shedding approach for best-effort query processing in event pattern detection. It strives for optimal result quality while staying within a bound for the pattern evaluation latency. The hybrid load shedding combines input-based load shedding with a fundamentally new approach to shed state elements. To realise this idea, we first propose a cost model to quantify the utility of state elements (partial results), which is directly linked to the utility of input events. Based on this cost model, we develop efficient decision procedures for hybrid load shedding which automatically balance input-based and state-based load shedding. We show how the selection of shedding candidates can be formulated as a Knapsack problem and how shedding strategies are based on its solution. In overload situations, it is infeasible to run complex forecasting procedures for deciding and balancing shedding strategies. To overcome this issue, we report an efficient implementation strategy to tune the cost model

granularity, its estimation and adaptation, and approximation schemes for efficient online decision making.

- **Efficient integration with remote data in event stream processing.** We propose a framework called EIRES in order to reduce the remote data fetching latency for queries that depend on remote data. EIRES employs a cache to decouple fetching remote data from its actual use in pattern detection. Fetching may happen before the need for remote data materialises (*i.e.*, preparing data beforehand) and the evaluation of partial results based on it may be postponed after the remote data is available (*i.e.*, lazy evaluation). To this end, we present a cost model to assess the utility of remote data based on state elements. We show how the model is approximated efficiently in an online manner. Based on this cost model, prefetching, lazy evaluation, and cache management are integrated to hide remote data transmission latency. We propose mechanisms to decide when and which remote data elements to prefetch, and when to postpone the evaluation if the required remote data element is unavailable. We elaborate on strategies for cache management, based on simple yet widely used policies (*e.g.*, the least-recently-used (LRU) eviction algorithm) and based on the proposed cost model.

- **A comprehensive case study of adaptive demand response management in smart grids.** To illustrate the effectiveness and efficiency of the proposed hybrid load shedding technique and EIRES framework in real-world applications on a large scale, we investigate a comprehensive case study of a crucial operation problem in smart grid management—*demand response* (DR) management. DR refers to the actions taken by the consumers to alter their energy usage patterns as requested by the utility operator for financial incentives. Its goal is to alleviate stress on the power grid during peak demand hours, shifting the required energy load to non-peak hours. We show that distributed ESP enables the utility operator to monitor consumers' compliance with energy reduction requests in real time. By detecting predefined patterns over smart meter reading event streams, the utility operator predicts non-compliance before the initial DR program (peak hours) ends and adaptively schedules further DR requests. Therefore, the utility operator is able to fully exploit the flexibility of consumers' energy demand reductions. We show that distributed ESP with hybrid load shedding and the EIRES framework enables adaptive DR management at utility-grid scale.

1.4 Publications

The contributions of this dissertation have been published in peer-reviewed scientific outlets. The following list provides an overview of these publications.

- Bo Zhao, Han van der Aa, Nguyen Thanh Tam, Nguyen Quoc Viet Hung, Matthias Weidlich, EIRES: Efficient Integration of Remote Data in Event Stream Processing, in the 47th ACM SIGMOD International Conference on Management of Data (*SIGMOD*), Xi'an, China, June 20-25, 2021, ACM, pp. 2128-2141.

- Bo Zhao, Nguyen Quoc Viet Hung, Matthias Weidlich, Load Shedding for Complex Event Processing: Input-based and State-based Techniques, in the 36th IEEE International Conference on Data Engineering (*ICDE*), Dallas, TX, USA, April 20-24, 2020, IEEE Computer Society, pp. 1093–1104.

- Gururaghav Raman, Jimmy Chih-Hsien Peng, Bo Zhao, Matthias Weidlich, Dynamic Decision Making for Demand Response through Adaptive Event Stream Monitoring, in 2019 IEEE Power and Energy Society General Meeting (*PES-GM*), Atlanta, GA, USA, August 4-8, 2019, The IEEE Power and Energy Society, pp. 1–5.

- Bo Zhao, Complex Event Processing under Constrained Resources by State-Based Load Shedding, in the 34th IEEE International Conference on Data Engineering (*ICDE*), Paris, France, April 16-19, 2018, IEEE Computer Society, pp. 1699–1703.

1.5 Dissertation Outline

The reminder of this dissertation is organised in six chapters:

- *Chapter 2 : Foundations*. This chapter provides essential background information of event stream processing. It formally defines core concepts of the data model, query model, and performance model of ESP. Specifically, it defines the concept of *state* that is maintained by an ESP engine during query execution. These concepts lay the foundation for a systematic discussion of the problem of efficient pattern detection over event streams and elaborate the contributions made by this dissertation.

- *Chapter 3 : Literature Review*. This chapter investigates the existing approaches for efficient ESP. Before examining ESP, we first present the state of the art in a broader context—efficient general data stream processing. We then review the recent advancements in state management for various big data processing systems and their differences compared to ESP. Turning to the context of ESP, we examine lossy and lossless optimisation techniques for efficient pattern detection and optimisation techniques for efficient remote data integration. We discuss the limitations of the state of the art in pattern detection over event streams.

- *Chapter 4 : Hybrid Load Shedding*. This chapter addresses the research problem **R1** to reduce the pattern evaluation latency of ESP. To this end, we first discuss how the pattern evaluation latency deteriorates due to dynamic and rapidly growing state elements.

Based thereon, we present a lossy optimisation technique—hybrid load shedding that discards both input events and state elements. We show how its cost model is based on state management, and present its efficient implementation. This chapter concludes with evaluations of synthetic datasets and real-world use cases.

- *Chapter 5: Efficient Remote Data Integration*. This chapter addresses the research problem **R2** to reduce the remote data fetching latency of ESP. We first discuss how the irregular access pattern for remote data source affects remote data fetching latency and its root cause—the dynamics of input streams and state elements. Based thereon, we propose the EIRES framework that hides remote data transmission latency by employing caching, prefetching, and lazy evaluation techniques. We show how these three techniques are integrated by a cost model based on state management. This chapter concludes with evaluations of synthetic datasets and real-world use cases.

- *Chapter 6: Case Study: Demand Response Management in Smart Grids*. This chapter applies efficient distributed ESP with the proposed hybrid load shedding technique and EIRES framework in demand response (DR) management in a smart grid. We first discuss the context of DR management and the limitations of the state of the art. Based thereon, we show how the distributed ESP enables the adaptive DR management with improved efficacy. We then present how hybrid load shedding technique and EIRES framework help to realise efficient adaptive DR management on a utility scale. This chapter concludes with evaluations of a real-world use case.

- *Chapter 7: Conclusion*. This chapter concludes this dissertation. In this chapter, we summarise the main contributions and results presented throughout this dissertation. Further more, we reflect on future work and how the contributions in this dissertation can be applied to other research fields beyond ESP.

CHAPTER 2

FOUNDATIONS

This chapter discusses the foundations of event stream processing. Section 2.1 describes the data model of event stream processing and presents formal definitions of core concepts. Section 2.2 explains the query model including a language model and an execution model, which also covers a formal definition of the *state* maintained during query execution. Section 2.3 provides performance models to evaluate the properties of query execution, including the definition of query processing latency and throughput.

2.1 Data Model

2.1.1 Event and Event Stream

Event stream processing (ESP) engines continuously evaluate queries over high-velocity event streams from different sources and applications. Query processing generates output event streams with higher-level semantics than the input events, and thus triggers real-world actions. Formally, we define notions of an event schema, an event, and an event stream as follows.

Definition 2.1 (Event schema)**.** An event schema is a finite sequence of attributes $A = \langle A_1, ..., A_n \rangle$, each attribute A_i, $1 \leq i \leq n$, being of a primitive data type (*e.g.*, integer or floating-point format). An event schema must include a timestamp attribute.

The timestamp attribute defines an event's occurrence time or arrival time, depending on the application requirement. It is necessary to enforce a temporal order for sequence patterns over event streams.

Table 2.1: Notations for ESP data model.

Notation	Explanation
$A = \langle A_1, \ldots, A_n \rangle$	Event schema
A_i	Event attribute
$e = \langle a_1, \ldots, a_n \rangle$	Event
$e.t$	Event timestamp
$S = \langle e_1, e_2, \ldots \rangle$	Event stream
$S(..k)$	Event stream prefix at up to the k-th input event
$S(k)$	The k-th event in the input stream
$d = \langle a_1, a_2, .., a_n \rangle$	A remote data element
$D = \{d_1, \ldots, d_n\}$	Remote data elements

Example 2.1. *The event schema of a credit card transaction may be* `Transaction` = ⟨`timestamp`, `location`, `cardnumber`, `cardholder`, `amount`, `merchantID`⟩. *The attributes* `timestamp`, `location`, `cardnumber`, *and* `cardholder` *are string types. The attribute* `amount` *is a float type, whereas the attribute* `merchantID` *is an integer type.*

Definition 2.2 (Event)**.** An event is an instantaneous, unique, and atomic occurrence of interest at a point in time. An instance of the event schema $A = \langle A_1, ..., A_n \rangle$ is an event $e = \langle a_1, ..., a_n \rangle$ with a_i being the value of attribute A_i, $1 \leq i \leq n$. One of the attributes must be the timestamp and its value is denoted by $e.t$.

The event schema is usually referred to as an *event type* and the attribute values are referred to as *event payload*.

Example 2.2. *A credit card transaction event of the schema defined in Example 2.1 may look like* `t` = ⟨`10:00 − 11 − 11 − 2020`, `Berlin`, `4301 5005 6778 4004`, `John Smith`, `100.00 EUR`, `9527`⟩. *This event means that John Smith, a credit card holder, paid 100.00 Euro in a shop (id 9527) in Berlin at 10:00 on November 11th, 2020.*

Definition 2.3 (Event stream)**.** An event stream is an infinite sequence of events, $S = \langle e_1, e_2, \ldots \rangle$, that respects the order of event timestamps: For any two events e_i and e_j, $i < j$ implies $e_i.t \leq e_j.t$.

We further define the notion of a finite stream prefix, up to index k, as $S(..k) = \langle e_1, \ldots, e_k \rangle$. Table 2.1 summarises our notations.

Example 2.3. *A sequence of credit card transactions ordered by timestamps is an event stream. Table 2.2 illustrates a snippet of such an event stream. Each row shows an event and each column represents attribute values of the event payload defined in the event schema (Example 2.1). Note that the first column does not belong to the event payload and represents the stream index—the offset of an event compared to the beginning of the input stream.*

Table 2.2: A snippet of a credit-card-transaction event stream.

Attribute:	time stamp	location	card number	card holder	amount	merchant ID
Type:	`string`	`string`	`string`	`string`	`float`	`integer`
…	…	…	…	…	…	…
320	'10:00-11-11-2020'	'Berlin'	'4301 5005 6778 4004'	'John Smith'	100.00	9527
321	'10:18-11-11-2020'	'Zürich'	'4301 5005 6778 4004'	'John Smith'	432.00	1024
322	'10:19-11-11-2020'	'Paris'	'3475 4511 5085 6060'	'John Wick'	14,456.78	6256
323	'10:20-11-11-2020'	'London'	'4546 5600 6062 3619'	'Tony Stark'	294,687.00	1984
…	…	…	…	…	…	…

As explained in Section 1.1, an ESP engine takes input as event streams and processes pattern detection queries. The results of query processing are event streams of output events. An output event may have a different schema compared to its input counterparts. For instance, assume that an pattern detection query detects fraudulent credit card transactions based on locations, timestamps, and the possibility of travelling between different locations within the corresponding time interval. The schema for output events may look like `FraudTransactions` = ⟨$\texttt{timestamp}_1$, $\texttt{timestamp}_2$,$\texttt{location}_1$,$\texttt{location}_2$, `cardnumber`,`cardholder`⟩. It contains the timestamps and locations of two (potential) fraudulent transactions of the same credit card. Such an output event has higher-level semantics compared to an input event (fraud transactions versus general transactions).

Example 2.4. *For the fraudulent transaction detection over the input stream in Table 2.2, the output event consists of the payload of the 320th and the 321st input events :* ⟨10 : 00 − 11 − 11 − 2020, 10 : 18 − 11 − 11 − 2020, Berlin, Zürich, 4301 5005 6778 4004, John Smith⟩. *For the same credit card of John Smith, there were two transactions within 18 minutes in Berlin and Zürich, which may be considered implausible.*

2.1.2 External Data Enrichment

As explained in Section 1.1, when processing a query, an ESP engine may need to lookup remote data, which is not directly available at the ESP engine, due to both privacy concerns and implementation considerations. Remote data sources may be key–value databases or relational databases. Without loss of generality, we assume that remote data sources can be queried by look-up requests sent from an ESP engine. A remote look-up request contains a look-up-query string and parameters to query the remote data sources. A certain look-up query string may be sent once and indexed by a unique identifier in remote data sources for further requests with different parameters. In this dissertation, we abstract a look-up query string as an identifier and the parameters as data values. Based thereon, we define a remote look-up request as follows.

Definition 2.4 (Remote look-up request)**.** A remote look-up request Q_d is a tuple composed of a look-up query id qid, and parameters p_i. $Q_d = \langle qid, p_1, p_2, ..., p_n \rangle$. Here, qid is an an identifier and each p_i is a primitive data type value.

Example 2.5. *A remote look-up request to retrieve John Smith's top three locations for most credit card transactions might be* ⟨1027, 4301 5005 6778 4004⟩. 1027 *is the look-up query id. It is registered as "*`SELECT location FROM history WHERE cardNumber=%P1 ORDER BY transactionVolume LIMIT 3`*" and the place holder,* `%P1`*, is materialised by John Smith's credit card number* 4301 5005 67 78 4004.

After processing a remote look-up request, remote data sources return the look-up results as a remote data element to an ESP engine. The ESP engine uses a remote data element to correlate and select input events according to the predicates of a pattern detection query. Moreover, a remote data element can also be integrated as an attribute value of the output event payload, depending on the output event schema. We define the schema of a remote data element as follows.

Definition 2.5 (Remote data schema)**.** The schema of a remote data element is defined as a tuple of attributes $D = \langle A_1, A_2, .., A_n \rangle$, each A_i being a primitive data type.

Definition 2.6 (Remote data element)**.** A remote data element, $d = \langle a_1, a_2, .., a_n \rangle$, is an instance of a remote data schema $D = \langle A_1, A_2, .., A_n \rangle$, each a_i being a data value.

Note that the schema of a remote data element does not specify the number of attributes. The reason is that the size of the look-up results may only be known *after* they are materialised by querying remote data sources.

Example 2.6. *Assuming the look-up request in Example 2.5 is sent to the remote data source in Table 2.3, the remote data element of the look-up result is* {`Potsdam`, `Berlin`, `Paris`}*—the top three locations in terms of transaction volume.*

Table 2.3: A snippet of a remote data source.

Attribute:	card number	location	volume
Type:	`string`	`string`	`float`
	'4301 5005 6778 4004'	'Potsdam'	24,357.78
	'4301 5005 6778 4004'	'Berlin'	14,268.49
	'4301 5005 6778 4004'	'Paris'	4,018.00
	'4301 5005 6778 4004'	'London'	1,129.35
	'4301 5005 6778 4004'	'Brussels'	901.05
	...	...	...

The remote data element in Example 2.6 (top three locations of credit card transactions) can be used to enrich the query processing in Example 2.4 (fraudulent credit card transaction detection) as follows. The output event in Example 2.4 indicates that both transactions happened in Berlin and Zürich could be fraudulent. However, the remote data element in Example 2.6 provides evidence that Berlin is the second highest-transaction-volume location, whereas Zürich

is not among the top three locations. Therefore, the transaction in Zürich may be more likely to be fraudulent than the one in Berlin.

2.2 Query Model

A pattern detection query is expressed by a query language. An ESP engine parses the query and generates a query plan based on an execution model. The query results are generated by the materialised execution model.

2.2.1 Language Model

Queries can be expressed by SQL-like high-level query languages (*e.g.*, SASE [5, 189]), or by low-level application programming interface (API) (*e.g.*, Apache Flink [38]). Common query languages and APIs define sequence event patterns, value predicates, and time windows. In this dissertation, our research problems (Section 1.2) and contributions (Section 1.3) focus on the execution model and are independent of specific languages and APIs in terms of certain ESP engines, because a query language is eventually materialised by the execution model. However, for illustrative purpose, we use a language model that extends the SASE query language [5, 189]. Specifically, we integrate remote data look-up requests and support nested sequence patterns. Here, we omit the detailed formal definitions and provide a descriptive definition of our proposed query language. The detailed definitions will be formalised in the execution model in Section 2.2.2.

In a nutshell, a query begins with the keyword `PATTERN` that is followed by the sequence pattern using the keyword `SEQ` which enforces a strict temporal order of a sequence of events. Value predicates that correlate event payloads are specified by a `WHERE` clause and are connected by logical operators including conjunction (`AND`), disjunction (`OR`), and negation (`NEG`). For remote data look-up requests, we omit the specific look-up queries sent to remote data sources, but highlight which value predicates involve remote data look up by the keyword `REMOTE` and the look-up parameters within square brackets (*e.g.*, `REMOTE[`*`parameter`*`]`). A time window is specified by a `WITHIN` clause. In this dissertation, we employ the concept of sliding time widow that consists of a *window* and a *slide*: Events are grouped within a window (defined by a window size) that slides across an event stream by a specified time interval (slide size). Figure 2.1 illustrates a time window with the window size of six time units and the slide size of one time unit, which means that the pattern and value predicates are evaluated per time window (six time units) and after one time unit a new time window is constructed. Figure 2.1 shows three overlapping time windows W_1, W_2, and W_3. If not specified, we assume the default slide size to be one time unit.

We elaborate on the above query language model using a representative query given in Listing 2.1, to detect fraudulent credit card transactions. Specifically, it defines sequence patterns of three types of events, `Transaction`, `Deny`, and `Limit`, using the key words `PATTERN` and `SEQ`. The pattern is triggered by a transaction event `t1` (type `Transaction`) of high volume (`t1.volume>10`). A

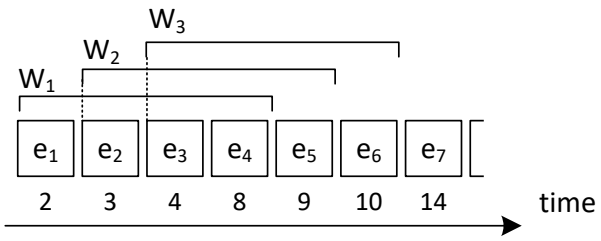


Figure 2.1: Sliding window (the window size is six time units and the slide size is one time unit).

suspicious pattern may emerge in two ways based on other events recorded for the same credit card (`SAME VALUE[creditCard]`): First, the event may be followed by a denied transaction `d` (type `Deny`) and another high-volume transaction `t2` that occurred at a location that differs from `t1` (`t1.location<>t2.location`) and is not in the set of known locations for the user, which is fetched from remote data sources (`t2.location NOT IN REMOTE[t1.user]`). Second, the initial event may be followed by a change in the spending limit (event `l` of type `Limit`), where the new limit is larger than the maximum limit of all credit cards within the same organisation (queried from remote data sources, `REMOTE[t1.organisation]`). Afterwards, another transaction is observed with a very high volume (`t3.volume>50k`), for which the beneficiary is not in the set of pre-authorised clients, as stored at a remote data source (`t3.beneficiary NOT IN REMOTE[t3.organisation]`). All this shall happen within a five-minute time window.

```
PATTERN SEQ(Transaction t1,
            (SEQ(Deny d, Transaction t2) OR SEQ(Limit l,  Transaction t3)))
WHERE SAME VALUE[creditCard] AND t1.volume> 10k
AND t2.volume > 10k
AND t1.location <> t2.location
AND t2.location NOT IN REMOTE[t1.user]
AND l.limit > REMOTE[t1.organisation]
AND t3.volume > 50k
AND t3.beneficiary NOT IN REMOTE[t3.organisation]
WITHIN 5min
```

Listing 2.1: Query to detect fraudulent credit card transactions.

2.2.2 Execution Model

To evaluate a pattern detection query, various formalisms have been proposed in the literature, most of them being based on automata, see [5, 32, 60], as well as operator tree structures [127]. We illustrate these two types of formalisms in Figure 2.2 for the example query in Listing 2.1.

Here, the automata-based model defines automaton states and transitions between them to describe how query results are step-wise constructed when processing an event stream. The query results (output events) are constructed at accepting automaton states (q_5 and q_6), whereas the

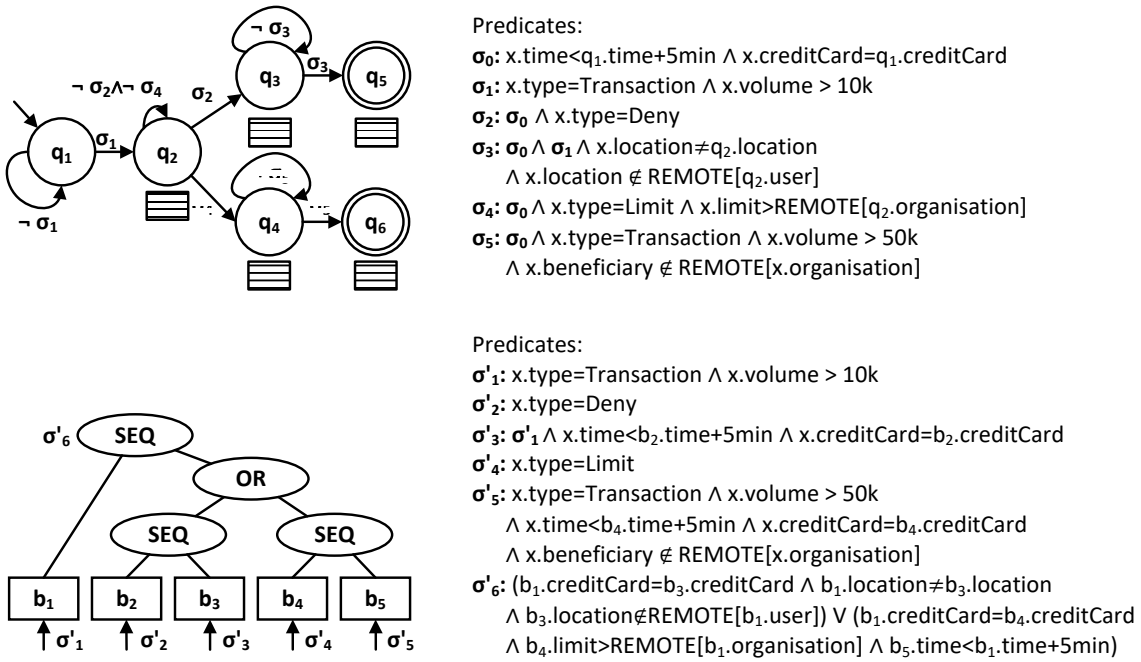


Figure 2.2: Execution model for pattern detection queries.

non-accepting ones (q_2, q_3, and q_4) construct intermediate results that are necessary to generate query results. These intermediate results are stored at buffers attached to the corresponding automaton states. Note that, the automata states are different from the conceptual *state* in the evaluation of a pattern detection query, which captures partial matches of the query. This conceptual state is independent of a specific execution model and refers to the intermediate results of query processing at a specific time.

The tree-based model defines a hierarchy of buffers. Inserting events into the leaf buffers, the tree is traversed from the leaves to the root, filling the operator buffers with intermediate results based on the child buffers. The query results are constructed at the root node.

In either evaluation model, event selection is governed by the value predicates. They correlate the current input event (denoted by x in Figure 2.2), events that are part of intermediate results (denoted by automaton state identifiers such as q_1, or tree buffer identifiers such as b_1), and data elements from remote sources (denoted by a parametrised variable such as `REMOTE`$[q_1.user]$). An example is the predicate σ_3 in Figure 2.2 that checks the time window, $x.time < q_1.time + 5min$ (*i.e.*, σ_0), a condition over the event's payload data, $x.type = Transaction \wedge x.volume > 10k$ (*i.e.*, σ_1), and a condition using remote data, $x.location \notin$ `REMOTE`$[q_1.user]$.

We formalise the essence of pattern detection query evaluation, under an automata-based or tree-based model, as follows.

Definition 2.7 (Query match)**.** Let Q be a query and τ_Q its time window. The results of evaluating

Q over a stream $S = \langle e_1, e_2, \ldots \rangle$ are matches. A query match is a finite sequence of events $\langle e'_1, \ldots, e'_m \rangle$ of the stream that is order-preserving, *i.e.*, for $e'_i = e_k$ and $e'_j = e_l$, $1 \leq k < l$, it holds that $i < j$ implies $k < l$, and respects the time window of the query, *i.e.*, $e'_m.t - e'_1.t \leq \tau_Q$.

If the sequence of events $\langle e'_1, \ldots, e'_m \rangle$ satisfies all the value predicates (*i.e.*, the sequence is constructed at an accepting state of an automaton or the root node of an operator tree), it is a **complete match**. Otherwise, it is a **partial match** (*i.e.*, the sequence is constructed at a non-accepting state of an automaton or a child node of an operator tree).

To derive complete matches, an ESP engine processes input streams incrementally, event by event. It maintains a set of partial matches. In the above execution models, partial matches correspond to partial runs of an automaton or the buffer content of non-root nodes in an operator tree. Partial matches in a certain automaton state or a tree node buffer are referred to as a *category* of partial matches. All the maintained partial matches of an ESP engine at a specific time are referred to as *state*. In any case, the number of partial matches is potentially exponential in the number of processed events [196], due to non-determinism in the automaton or the enumeration of all subsequences in a tree buffer. Given the current set of partial matches, an ESP engine checks how a new event of the stream changes the partial matches, *i.e.*, whether it leads to a partial match being discarded, extended, or split up. To this end, the query time window and the value predicates are evaluated, which potentially involves remote data elements.

Event selection policies also affect the consumption and selection of events during query execution [3]. An event selection policy defines how many events can be skipped from the input stream for sequence pattern detection. Specifically, we consider a *greedy* policy, under which an arbitrary number of input events can be skipped. This means that, based on the transition guards in the automaton or a tree structure, a partial match that can be extended with an input event from the input stream is always split up into at least two partial matches, one extended with the event and one left unchanged. The latter models the case that an input event may be skipped, to derive matches constructed from *any* set of events that satisfy the value predicates. This selection semantics is also known as *unconstraint* [40] or *skip-till-any-match* [5].

We also consider a *non-greedy* policy, also known as *continuous* or *skip-till-next-match* [5]. Here, a partial match would only be extended, *i.e.*, only events that do not satisfy the value predicates are skipped to always select the *next* event that satisfies these value predicates. We illustrate the semantics of greedy and non-greedy policies with a concrete example step by step in Example 2.7.

Example 2.7. *We consider an event stream consisting of event types A and B. Accordingly, an event is represented by a_i or b_j, with each subscript being the timestamp. For instance, a_1 means an event of type A at timestamp 1. To illustrate the event selection policies, we ignore the value predicates and only detect a sequence pattern* `PATTERN SEQ (A a, B b, A c) WITHIN 10 time units`

Table 2.4: Event selection policies.

Index	Input stream	Greedy selection	Non-greedy selection
1	a_1 $b_2 b_3 a_4 b_5 a_6$	$\langle a_1 \rangle$	$\langle a_1 \rangle$
2	$a_1 b_2$ $b_3 a_4 b_5 a_6$	$\langle a_1 \rangle$, $\langle a_1 b_2 \rangle$	$\langle a_1 b_2 \rangle$
3	$a_1 b_2 b_3$ $a_4 b_5 a_6$	$\langle a_1 \rangle$, $\langle a_1 b_2 \rangle$, $\langle a_1 b_3 \rangle$	$\langle a_1 b_2 \rangle$
4	$a_1 b_2 b_3 a_4$ $b_5 a_6$	$\langle a_1 \rangle$, $\langle a_1 b_2 \rangle$, $\langle a_1 b_3 \rangle$, $\langle a_1 b_2 a_4 \rangle$, $\langle a_1 b_3 a_4 \rangle$, $\langle a_4 \rangle$	$\langle a_1 b_2 a_4 \rangle$, $\langle a_4 \rangle$
5	$a_1 b_2 b_3 a_4 b_5$ a_6	$\langle a_1 \rangle$, $\langle a_1 b_2 \rangle$, $\langle a_1 b_3 \rangle$, $\langle a_1 b_5 \rangle$, $\langle a_4 \rangle$, $\langle a_4 b_5 \rangle$	$\langle a_4 b_5 \rangle$
6	$a_1 b_2 b_3 a_4 b_5 a_6$	$\langle a_1 \rangle$, $\langle a_1 b_2 \rangle$, $\langle a_1 b_3 \rangle$, $\langle a_1 b_5 \rangle$, $\langle a_1 b_2 a_6 \rangle$, $\langle a_1 b_3 a_6 \rangle$, $\langle a_1 b_5 a_6 \rangle$, $\langle a_4 \rangle$, $\langle a_4 b_5 \rangle$, $\langle a_4 b_5 a_6 \rangle$, $\langle a_6 \rangle$	$\langle a_4 b_5 a_6 \rangle$, $\langle a_6 \rangle$

over a six-event snippet of an input event stream: $a_1\ b_2\ b_3\ a_4\ b_5\ a_6$. *Table 2.4 shows the query execution event by event. Each row demonstrates the maintained partial matches and detected complete matches (coloured in orange) after processing a new input event (coloured in red), under different event selection policies. The processed stream prefix is within a box.*

We observe that the greedy selection policy generates more state elements (partial matches) and query results (complete matches) compared to the non-greedy one. This is because a partial match can only be extended under non-greedy policy, whereas it is also split up under greedy policy. For instance, upon the arrival of event b_2, the greedy policy splits up partial match $\langle a_1 \rangle$ into two partial matches. One is extended to $\langle a_1 b_2 \rangle$. The other is unchanged. However, the non-greedy policy only extends $\langle a_1 \rangle$ to $\langle a_1 b_2 \rangle$. As a result, under the greedy policy, $\langle a_1 \rangle$ contributes to three partial matches ($\langle a_1 b_2 \rangle$, $\langle a_1 b_3 \rangle$, $\langle a_1 b_5 \rangle$), and five complete matches ($\langle a_1 b_2 a_4 \rangle$, $\langle a_1 b_3 a_4 \rangle$, $\langle a_1 b_2 a_6 \rangle$, $\langle a_1 b_3 a_6 \rangle$, $\langle a_1 b_5 a_6 \rangle$). However, the same partial match only contributes to one partial match $\langle a_1 b_2 \rangle$ and one complete match ($\langle a_1 b_2 a_4 \rangle$) under the non-greedy policy.

In our formal execution model, we capture the core concepts of query execution that applies to both automata-based and tree-based execution models. To this end, we model the state of query execution at the stream index k as $P(k) = \{\langle e_1, \ldots, e_n \rangle, \ldots, \langle e'_1, \ldots, e'_m \rangle\}$. It is the set of partial matches maintained by the ESP engine for query Q after processing a stream prefix $S(..k)$. The next input event in the stream is $S(k+1)$. Processing it potentially requires remote data elements. We model the set of remote data elements needed by a partial match $p \in P(k)$ when processing event $S(k+1)$ by a set $D(p, k+1) \subseteq \mathscr{D}$. All remote data elements required by state $P(k)$ and input

Table 2.5: Notations for ESP execution model.

Notation	Explanation
$\mathscr{D} = \{d_1, \ldots, d_n\}$	Remote data elements
f_Q	The query processing function
p	A partial match
$P(k)$	Partial matches up to the k-th input event
$D(p,k), D(k)$	Remote data needed by partial match p, or all partial matches, to process the k-th input event
$C(k)$	Complete matches up to the k-th input event
R	Output stream of complete matches

Table 2.6: Query processing procedure.

k	$S(k+1)$	$P(k)$	$D(k+1)$	$P(k+1)$	$C(k+1)$
0	a_1	$\emptyset$	$\emptyset$	$\emptyset$	$\emptyset$
1	b_2	$\langle a_1 \rangle$	$\emptyset$	$\langle a_1 \rangle, \langle a_1 b_2 \rangle$	$\emptyset$
2	b_3	$\langle a_1 \rangle, \langle a_1 b_2 \rangle$	$\emptyset$	$\langle a_1 \rangle, \langle a_1 b_2 \rangle, \langle a_1 b_3 \rangle$	$\emptyset$
3	a_4	$\langle a_1 \rangle, \langle a_1 b_2 \rangle, \langle a_1 b_3 \rangle$	REMOTE[$b_2.v$] REMOTE[$b_3.v$]	$\langle a_1 \rangle, \langle a_1 b_2 \rangle, \langle a_1 b_3 \rangle,$ $\langle a_4 \rangle$	$\langle a_1 b_2 a_4 \rangle, \langle a_1 b_3 a_4 \rangle$
4	b_5	$\langle a_1 \rangle, \langle a_1 b_2 \rangle, \langle a_1 b_3 \rangle, \langle a_4 \rangle$	$\emptyset$	$\langle a_1 \rangle, \langle a_1 b_2 \rangle, \langle a_1 b_3 \rangle,$ $\langle a_1 b_5 \rangle, \langle a_4 \rangle, \langle a_4 b_5 \rangle$	$\emptyset$
5	a_6	$\langle a_1 \rangle, \langle a_1 b_2 \rangle, \langle a_1 b_3 \rangle, \langle a_4 \rangle$ $\langle a_1 b_5 \rangle, \langle a_4 b_5 \rangle$	REMOTE[$b_2.v$] REMOTE[$b_3.v$] REMOTE[$b_5.v$]	$\langle a_1 \rangle, \langle a_1 b_2 \rangle, \langle a_1 b_3 \rangle,$ $\langle a_1 b_5 \rangle, \langle a_4 \rangle, \langle a_4 b_5 \rangle$	$\langle a_1 b_2 a_6 \rangle, \langle a_1 b_3 a_6 \rangle$ $\langle a_1 b_5 a_6 \rangle, \langle a_4 b_5 a_6 \rangle$

event $S(k+1)$ are given by $D(k+1) = \bigcup_{p \in P(k)} D(p, k+1)$. Based thereon, the functionality of an ESP engine can be described as defined in Definition 2.8. Table 2.5 summarises our notations.

Definition 2.8 (Query processing)**.** Query processing is a function f_Q that takes the current input event $S(k+1)$, the current state $P(k)$, and the required remote data $D(k+1)$ as input and returns sets of new partial matches $P(k+1)$ and complete matches $C(k+1)$:

$$f_Q(S(k+1), P(k), D(k+1)) \mapsto P(k+1), C(k+1). \tag{2.1}$$

Applying this function repeatedly for an input stream constructs a stream of sets of complete matches, $R = \langle C(1), C(2), \ldots \rangle$. To achieve compositionality of the model, an ESP engine may order the matches per set and construct a single event per match. This way, R is transformed into another event stream again (*e.g.*, an input event stream for another query). We illustrate this formal execution model using a concrete example in Example 2.8.

Table 2.7: Notations for ESP performance model.

Notation	Explanation
c	A complete match
$\ell_{match}(c)$	Pattern evaluation latency of the match c
$F(c)$	Remote data fetch operation of the match c
$\ell_{fetch}(c)$	Remote data fetching latency of the match c
$\ell(c)$	Query processing latency of the match c
$\ell(k)$	Query processing latency at the k-th input event
$t(k)$	Query processing throughput at the k-th input event

Example 2.8. *We consider the same input event stream and query in Example 2.7: Detect the sequence pattern,* `PATTERN SEQ (A a, B b, A c) WITHIN` 10 `time units`*, over a snippet of an input event stream* $a_1\ b_2\ b_3\ a_4\ b_5\ a_6$ *under greedy selection policy. Here, we assume that an event of type B,* b_i*, requires a remote data element* `REMOTE`$[b_i.v]$ *based on its attribute value* $b_i.v$*. Table 2.6 shows the materialisation of query processing function* f_Q *with an increasing stream index* k*, which depicts the changes of partial matches* $P(k)$ *and* $P(k+1)$*, the required remote data elements* $D(k+1)$ *and the detected complete matches* $C(k+1)$*.*

For instance, at the stream index of 3 (the 4th row in Table 2.6), the current set of maintained partial matches, $P(3)$*, includes* $\langle a_1\rangle$*,* $\langle a_1 b_2\rangle$*, and* $\langle a_1 b_3\rangle$*. When processing the next input event* $S(4) = a_4$*, the query processing function* f_Q *needs remote data elements* $D(4) = \{$`REMOTE`$[b_1.v]$, `REMOTE`$[b_2.v]\}$*. After elevating* $f_Q(S(4), P(3), D(4)) \mapsto P(4)), C(4)$*, the ESP engine generates a new set of partial matches* $P(4) = \{\langle a_1\rangle, \langle a_1 b_2\rangle, \langle a_1 b_3\rangle, \langle a_4\rangle\}$ *and complete matches* $C(4) = \{\langle a_1 b_2 a_4\rangle, \langle a_1 b_3 a_4\rangle\}$*.*

2.3 Performance Model

The performance of an ESP engine is measured by throughput and latency. Throughput is the number of input events being processed within a unit of time (*e.g.*, events per second). Latency is the time between the arrival of the last event needed to detect a complete match at the ESP engine and the actual detection of that match. ESP query processing aims for both high-throughput and low-latency analysis. It should be noted that high throughput does not necessarily mean low latency. For instance, batch-based data analytic systems (*e.g.*, Apache Hadoop [92]) achieve high throughput, but also suffer from the problem of high latency.

In our query execution model, the query processing latency corresponds to the time needed to evaluate function f_Q in Equation 2.1. We denote the processing latency for a complete match c by $\ell(c)$. The query processing latency consists of two aspects. One is the pattern evaluation latency $\ell_{match}(c)$ that is inherent to the evaluation of partial matches when processing all events of the stream from the first to the last event that get selected as part of the creation of match c. The

other is the latency of fetching remote data elements that was required to process these events, denoted by $\ell_{fetch}(c)$. Table 2.7 summarises our notations. We define them formally as follows:

Definition 2.9 (Pattern evaluation latency)**.** Assuming the last event of the complete match c arrives at the ESP engine at time t_a and c is detected at time t_d, the pattern evaluation latency $\ell_{match}(c)$ is defined as

$$\ell_{match}(c) = t_d - t_a \tag{2.2}$$

A fetch operation of the remote data is defined by two timestamps (t_{nr}, t_{ar}), *i.e.*, the time $t_{nr} \in \mathbb{N}$ at which the *need* for remote data is detected during query evaluation and the time $t_{ar} \in \mathbb{N}$ at which the data is *available* and processing can continue.

Definition 2.10 (Remote data fetching latency)**.** Assuming the construction of the complete match c requires a set of fetch operations $F(c) \subseteq \mathbb{N} \times \mathbb{N}$, the remote fetching latency $\ell_{fetch}(c)$ is defined as

$$\ell_{fetch}(c) = \sum_{(t_{nr}, t_{ar}) \in F(c)} t_{ar} - t_{nr} \tag{2.3}$$

Note that the sum means that multiple remote data elements are required sequentially at *different* automaton states or operator tree nodes. If one single automata state or operator tree node needs multiple remote data elements, the corresponding fetching latency is the maximal latency, because these remote data elements can be transmitted in parallel.

Definition 2.11 (Latency)**.** The processing latency of the complete match c, $\ell(c)$, is defined as

$$\ell(c) = \ell_{match}(c) + \ell_{fetch}(c) = \ell_{match}(c) + \sum_{(t_{nr}, t_{ar}) \in F(c)} t_{ar} - t_{nr} \tag{2.4}$$

For a particular point in stream processing, *i.e.*, when a set of matches $C(k)$ have been generated at stream prefix $S(..k)$, the latency is derived through an aggregation function α (*e.g.*, median, mean, 95^{th}-percentile) from all these matches,

$$\ell(k) = \alpha\{\ell(c) \mid c \in C(k)\} \tag{2.5}$$

The throughput measures the number of processed events per time unit for an ESP engine. It is defined as follows:

Definition 2.12 (Throughput)**.** Assuming the arrival time of the $(k-1)$-th event, $S(k-1)$, and the k-th event, $S(k)$, are $t_s(k-1)$ and $t_s(k)$, $k \geq 1$, the throughout at $S(k)$, $\boldsymbol{t}(k)$, is defined as

$$\boldsymbol{t}(k) = \frac{1}{t_s(k) - t_s(k-1)} \tag{2.6}$$

CHAPTER 3

LITERATURE REVIEW

This chapter discusses the state of the art for efficient ESP. We first present the context of general data stream processing in Section 3.1 including relational data streams and XML (extensible markup language) data streams, as well as their major challenges. Section 3.2 discusses state management in data stream processing systems. Turning to the more specific topic of efficient ESP, we discuss optimisations for efficient pattern detection over event streams (Section 3.3). Lastly, in Section 3.4 we examine different optimisation techniques for efficient data enrichment from external sources in ESP.

Note that, we aim to provide a broad overview spectrum of data stream management systems and the challenges addressed by them. We present the background for related work on optimisations for state management and efficient pattern detection in ESP. A recent survey [77] comprehensively examines the evolution of stream processing systems.

3.1 General Data Stream Processing

Data stream management systems continuously evaluate queries over unbounded streams of data. Tapestry [179] is one of the first data stream management systems. It is an extension of relational database systems to evaluate continues queries over append-only databases. Henceforth, a lot of research on stream processing focused on relational data. In addition, another branch of research on stream processing explored data captured in an XML format. This section summarise these two branches of research.

3.1.1 Relational Data Stream Processing

Most of the academic prototypes and commercial products focus on relational data streams. This is because the fundamental concepts and ideas of data stream processing have originally been derived from the well-established database community. Representatives of those early prototypes include NiagaraCQ [47], TelegraphCQ [44], STREAM [12], Auroral/Borealis [1, 39], and Gigascope [59]. Even though they all share a similar relational data model, they differ in query semantics [13, 30]. These systems mainly address the challenges of efficient sliding window aggregation operations [14, 117], fault tolerance [17, 171], adaptive query processing [167], and queries for pattern detection [189]. Early systems usually employed a micro/mini-batch paradigm to realise stream processing [116].

Modern stream processing systems are influenced by the development of large-scale distributed data management systems [111], especially open-source systems like MapReduce [65]. The focus shifted towards scaling-out, data-parallel stream processing engines on shared-nothing architectures that run on commodity hardware, where the challenges of high robustness [115], fault tolerance [10, 72], and out-of-order stream processing [100, 118] need to be addressed. Some important examples include Millwheel [6], Apache Storm [183], Spark Streaming [63], Apache Flink [38], and Samza [138]. They organise streaming computations as dataflow graphs and transparently partition streams for execution on computer clusters to achieve data parallelism and high throughput [2]. Users can specify such dataflow graphs using SQL-like query language or low-level APIs [96]. A recent survey [124] discussed resource management and scheduling in distributed stream processing systems, showing that an SLA (service level agreement)-aware, cost-efficient, and self-adaptive resource management and scheduling framework is a promising future direction.

Another trend for modern stream processing systems is to exploit parallelism on heterogeneous hardware inducing multicore CPUs, GPUs and FPGAs (Field programmable gate arrays). Instead of scaling out, they focus on scaling up to achieve efficient computation. For instance, Trill [43], StreamBox [129, 130], BriskStream [197], and LightSaber [180] leverage parallelism on shared-memory multicore architectures. Turning to GPU acceleration, GStream [200] supports data streaming on GPU clusters, while SABER [110] allows hybrid processing on both, CPUs and GPUs, with dynamic scheduling. FPGAs are hardware chips that users can reprogram to achieve high performance. Hagiescu *et al.* [93] address challenges related to implementing stream processing on FPGAs. Sidler *et al.* [172] accelerate pattern matching queries in hybrid CPU-FPGA architectures. Compared to shared-nothing computer clusters, these systems have significantly higher interconnect bandwidth and, therefore, avoid the overhead of data transfer and migration between nodes in clusters. In addition to high throughput, they achieve low-latency processing. In this dissertation, we aim for low-latency pattern detection at microsecond level and build our prototype stream processing system for multicore processors with shared memory.

Turning to queries for pattern detection over event streams, the SASE system [91, 189] presents a query language and an automata-based execution model to support sequential pattern detection [5], Kleene closure operator [90], and imprecise timestamps [195]. The authors also analysed the computational complexity of expensive queries [196]. Cugola and Margara [61] surveyed the different data stream systems and complex event processing systems, and elaborated their relations. We cover more ESP systems and their optimisations in Section 3.3 and Section 3.4.

3.1.2 XML Data Stream Processing

While XML was originally designed as an extensible markup format for document management, it has been quickly extended to other areas because of the wide-availability of free XML parsers, its readability, and flexibility. Besides the use of document markup, the two key application scenarios of XML-based middleware/systems are the use for interoperable data interchange in loosely-coupled systems and for ad-hoc modelling of semi-structured data. SQL-like query languages such as XPath [55] and XQuery [27] have been among the first to express query semantics for general-purpose applications.

In the context of XML stream processing [107], the filter problem is the core technical challenge, which has been a subject of intense research [7, 41, 42, 66, 89]. There, a collection of user-defined XPath/XQuery queries are executed on streams of XML documents. The goal is to identify the elements that satisfy each query, as well as to route content to appropriate endpoints. Several XPath streaming engines have been proposed, including XSQ [145], SPEX [139], XAOS [22], and TwigM [51]. Turning to XQuery streaming engines, BEA [75] provides high performance for message-processing applications. FluXQuery [109] proposes optimisations based on rewriting of XQuery queries and buffer management in main memory. In the same vein, GCX [108] reduces main memory consumption and execution time by dynamically updating the maximum capacities of buffers.

Turning to more complex pattern detection queries, Mozafari *et al.* [132] present the XSeq language and a system that supports queries with sequential patterns and Kleene closure operators on XML streams. The execution model of the XSeq system is based on the visibly pushdown automata (VPA).

3.2 State Management

The concept of *state* and its applications vary widely across different big data processing systems. Roy and Haridi [166] define state to be "*a sequence of values in time that contain the intermediate results of a desired computation*." Recently, To *et al.* [182] surveyed state management in modern big data processing systems including those implementing a batching or streaming paradigm. They define the state as "*the intermediate value of a specific computation that will be used in subsequent operations during the processing of a data flow*."

In this dissertation, we focus on pattern detection over event streams. Hence, we define the *state* as partially matched patterns at a specific time during query evaluation. State management plays a pivotal role for stateful stream processing where state elements are heavily and frequently reused among different executions of a step function. Reusing state elements increases the expressiveness of streaming systems and enables complex analytical queries including streaming join, sequential pattern detection, common machine learning algorithms, and complex user-defined functions. Early distributed stream processing systems usually lack support for explicit state management, due to their focus on scaling-out via data parallelism. Several research prototypes have been proposed to overcome this issue. We discuss them in four dimensions.

Scalability. Scalable state management is the main optimisation direction for distributed stream processing systems, where state elements are typically partitioned by predefined keys and are carefully placed for different operators in a data flow graph. The primary goal is to scale out a system by exploring data parallelism. For example, Fernandez *et al.* [72] propose data structures, such as key-value pairs, to represent various state types (*e.g.*, processing state, buffer state, and routing state). They further propose stateful dataflow graphs (SDG) as a model for parallelisation [73]. With the help of explicit annotations (according to application semantics), SDGs separate data from mutable state elements and partition them for distributed computing. In the same vein, ChronoStream [191] divides computation state into a collection of fine-grained slice units and provides transparent elasticity support in cloud computing. Similar strategies have been applied in popular open-source systems, such as Apache Flink [37], and commercial industry projects, such as Samza [138]. However, the drawback of these distribution strategies is the large overhead to migrate state between distributed machines. This will significantly increase the overall processing latency. An alternative solution is to scale up state management in multicore CPUs, GPUs, and FPGAs, where state migration happens in main memory with significantly higher bandwidth than in the case of inter-machine network connections [197, 199].

Storage. States may be stored internally in main memory or in external storage such as disks and remote databases. In general, the number of state elements and the applications' latency bounds determine where state shall be stored. For a small number of state elements and low-latency requirements, researchers [135, 161, 194] proposed maintaining state in main memory, which can accelerate stream processing [194], but can also make failure recovery difficult: Replicating state to different machines will be needed, in order to recover from machine failures. In contrast, for large numbers of state elements and relaxed latency bounds, state elements are kept in persistent storage, such as disks [113, 122, 135]. However, this incurs a larger overhead for data movement. Nonetheless, deciding where to optimally store state is not always trivial and a hybrid approach is usually employed. For instance, Spark streaming [193] represents state as resilient distributed datasets (RDDs). RDDs are stored in disks and parts of them are cached in the main memory. Samza [138] can manage a large number of state elements (*e.g.*, several gigabytes in each partition) by preserving them in local storage and using Kafka [168] to maintain state

changes. Flink [37] keeps state in main memory (*i.e.*, holds state elements internally as objects on the Java heap), and periodically backs up state in a file system (*e.g.*, HDFS [92]), or persists state in RocksDB [164].

Programmability. Most data stream processing systems forbid users to manage state directly. The state is also referred to as internal *synopsis*, which is only visible to internal system components. Most systems allow users to define a state partition scheme before the SQL-like queries are compiled to data flow graphs. However, some systems provide APIs to query the internal state. For example, Flink introduces the concept of *queryable state* [37] that enables queries to access state's statistics. Yet, manipulation of internal state is typically not allowed for users, which guarantees the correctness of the query evaluation, but fails to leverage semantic optimisations gained from domain experts (*i.e.*, domain experts may be aware that some state elements will be filtered out eventually and therefore, prune them early during the runtime).

Fault tolerance. Fault tolerance refers to a system's ability to continue query processing despite of failures and to deliver the expected results as if no failures had happened. It is curial for data stream processing systems for two reasons. First, data stream processing systems perform stateful computations over unbounded data streams. Without fault tolerance, data stream processing systems would have to redo computations from scratch given that the state would be lost when a failure happens. However, re-computation may be infeasible because the processed segment of a data stream has been permanently discarded. Second, modern distributed stream processing systems deploy the data flow graph of queries on multiple commodity machines where failures are inevitable. Against this background, various strategies have been proposed to achieve fault tolerance in data stream processing systems.

Fault-tolerant data stream processing systems usually persist snapshots of state in reliable storage and update them periodically as *checkpoints* [17, 52, 72, 97, 98, 121, 153]. When failure occurs, data stream processing systems restore the state to another node, thereby recovering the computation from the last checkpoint. Fault tolerance, in general, requires redundancy, which can be achieved in several ways. According to Hwang *et al.* [98], there are three fault tolerance mechanisms: passive standby, active standby, and upstream backup. The passive standby approach only backs up the modified part of the state periodically. The active standby approach replicates the input data stream for another compute node and evaluates the queries in parallel. Regarding the upstream backup mechanism, each primary compute node retains the output, while the backup is still inactive. If a primary compute node fails, the backup restores its state by reprocessing data tuples stored at upstream backup nodes.

Note that there exit other challenges and optimisation opportunities in the context of state management. We discuss load balancing in Section 3.3 and state access patterns in Section 3.4. For a more comprehensive and detailed investigation, we refer to the recent survey by To *et al.* [182].

3.3 Efficient Pattern Detection in Event Stream Processing

The inherent complexity of evaluating pattern detection queries over event streams is widely acknowledged [15, 196]. To tackle this issue, various optimisations have been proposed. Nonetheless, they can be categorised as either lossless or lossy optimisations. Representative lossless optimisations include parallelisation schemes [18, 95, 126], pattern sharing [149, 159, 196], and semantic query rewriting [68, 187]. In contrast, lossy optimisations include load shedding [94, 173–175] and approximate query processing (AQP) [120]. We discuss these two categories separately.

3.3.1 Lossless Optimisations

Lossless optimisations do not discard any processing elements (input events or state) and guarantee accurate query results. Recall that the primary goal of event stream processing is low latency. When facing high-velocity input streams and a large number of state elements (partial matches), lossless optimisations achieve low-latency processing by increasing a system's computing capacity, by reusing already computed state, by rewriting queries to leverage semantic constrains (*i.e.*, reduce unnecessary state elements), and by encoding the state in a compact way.

Parallelisation. Recently, Röger and Mayer [165] comprehensively surveyed different parallelisation schemes in stream processing. The main idea of parallelisation is to partition the input stream or state elements into multiple subsets which are processed in parallel. Parallelisation can be realised by partitioning an input stream into multiple sub-streams or shards based on predefined keys [20, 81, 134, 169]. Each sub-stream or shard is handled independently by multiple instances of the event stream processing engine. Another parallelisation scheme is based on time windows [125, 126]. Specifically, a so-called splitter partitions the input event stream into subsequences, *i.e.*windows of events. It then assigns each window to an instance of the engine. However, processing each window in isolation may neglect opportunities for optimisation as results for overlapping sections of windows may be shared. Therefore, pane-based splitting has been proposed [19, 116]. It partitions the input stream into non-overlapping sub-sequences, called panes. Each pane belongs to one or more windows and is processed independently. In the same vein, Balkesen *et al.* [18] propose run-based intra-operator parallelisation (RIP), a window-based parallelisation scheme tailored for automata-based queries, where an input event stream is split into batches of events according to the windowing policy. Each batch is assigned to an instance of an automata-based ESP engine.

The major assumption of parallelisation schemes is the independence of different sub-streams, windows, panes, and runs. However, such an assumption may not always hold, especially for pattern detection queries over event streams, where new state elements are iteratively computed from previous ones. The other limitation of parallelisation schemes is that they have to be decided in advance, as well as the assumption of steady input event streams. Users shall allocate computing resources before compiling queries into a data flow graph via the ESP engine.

Over-provisioning of resources will result in wasted computing power. In contrast, insufficient resources increase processing latency. Even though approaches for elastic scale out may improve performance after additional computing resources have been acquired, ESP suffers from a performance bottleneck during the resource allocation phase and state migration. For instance, resharding a stream in Amazon Kinesis to double the throughput may take up to an hour if the input stream comprises around 100 shards already [8].

Pattern sharing. If multiple queries contain overlapping sub-patterns, the ESP engine shall share partial matches of these common sub-patterns in order to reuse the already computed state for high performance. E-Cube [123] exploits a hierarchy of query refinement from abstract patterns to more specific ones for multiple queries, thereby enabling the reuse of intermediate results along the refinement hierarchy. Cao *et al.* [36] achieve minimal utilisation of both compute and memory resources for processing a query workload in which queries share parts of patterns. They formulate the selection of shared patterns as a *skyline* problem [29] and present efficient strategies to solve this problem. Ray *et al.* [159] leverage time-based event correlations among queries and employ stream transactions that assure concurrent shared maintenance and reuse of sub-patterns across queries. Poppe *et al.* [152] propose the *sharon* optimiser which compactly encodes sharing candidates, their benefits and conflicts among candidates into a graph. Based thereon, they map the problem of finding an optimal sharing plan to an optimisation problem of the maximum weight independent set (MWIS).

The limitation of pattern sharing approaches is that they contribute little for query workloads with few common sub-patterns. For the query workloads that indeed have overlapping sub-patterns, the benefits largely depend on the size of shared sub-patterns and how many instances of them will be eventually materialised.

Query rewriting. When users specify complex patterns through SQL-like query languages, they are usually unaware of the actual query plan generation (data flow graphs) and optimal operator placement. In traditional database management systems, these optimisations are usually achieved by query rewriting. Similarly, ESP engines may rewrite the original query to generate more efficient query plans and carefully organise operators in a data flow graph or an operator tree. Schultz-Møller *et al.* [170] proposed a ESP engine called NEXT. It translates a high-level query language to an automata-based execution model. These automata are rewritten to enable more efficient processing. The automata are further distributed across a cluster of compute nodes for scalability. Li *et al.* [119] let users specify constraints about the input stream. Based thereon, the ESP engine can terminate corresponding pattern detection earlier and release unnecessary computing resources. Weidlich *et al.* [187] proposed a comprehensive method for pattern detection query optimisation using business process models. The method is based on the extraction of behavioural constraints used to rewrite patterns for event detection, and to select and transform execution plans. Ding *et al.* [68] utilise dynamic substream metadata at runtime to find

more efficient query plans than the one selected at compilation time. They instantiate multiple concurrent logical query plans, each processing different partially overlapping substreams and execute them in a single physical plan. Calbimonte *et al.* [34] show the possibility of leveraging optimisation techniques of ontology-based data access (OBDA) for relational databases to rewrite continuous queries over resource description framework (RDF) data streams.

Even though query rewriting can improve event stream processing performance in general, it lacks the ability to keep the steady performance when facing with an overload situation with highly dynamic input streams, where the number of state elements changes irregularly. The upper-bound of processing performance is determined once the query plan is generated. Even if the actual query plan can be dynamically adapted, state migration is inevitable and significantly increases the overall end-to-end processing latency.

Compact representation of state. Compact encoding of the state reduces both memory and CPU usage. Poppe *et al.* [149] define a graph to compactly encode all patterns matched by a query. Based on this graph, they define the spectrum of pattern detection algorithms from CPU-optimal to memory-optimal approaches. They further propose GRETA [150], a graph that dynamically propagates aggregations along its edges. Based on this graph, the final aggregation is incrementally updated and instantaneously returned at the end of each query window. In the same vein, they abstract the graph to minimise the number of aggregations – reducing both time and space complexity [151].

The limitation of optimisations based on the compact encoding of the state is that the performance gain is bound to a certain level – the most compact encoding. If the system load exceeds the computing capacity, the ESP engine becomes overloaded and low-latency processing is no longer possible.

3.3.2 Lossy Optimisations

Lossy optimisations improve the performance of event stream processing at the expense of result quality. This can be achieved by discarding some input events or state elements (*i.e.*, load shedding). Alternatively, an ESP engine may also deliver approximated query results (*i.e.*, approximate query processing). The actual usefulness of the lossy query results depends on the requirements of specific applications. Some applications enforce strict latency bounds (*e.g.*, 25ms for credit card fraud detection), where delivering some results in time is more important than late, yet accurate and complete query answers. Note that lossy optimisations are largely orthogonal to their lossless counterparts. For instance, an ESP engine may employ parallelisation schemes to enable scale-out when faced with increasing load, while during the resource allocation phase and state migration, load shedding techniques may be employed to keep the processing latency low.

Load shedding. Load shedding schemes improve stream processing performance by discarding stream elements without processing them. However, the side effect is that query processing

becomes lossy. A range of optimisation techniques have been proposed to improve performance as much as possible while minimising the loss of query results. Load shedding has received considerable attention in the broad field of general stream processing. To cope with bursty input rates, random shedding strategies have been implemented in widely-used streaming systems, *e.g.*, Apache Pulsar [11], Apache Heron [74, 79], and Apache Kafka [21]. Shedding may also be guided by the queueing latency [163], the concept drift detection [102], and the expected quality of service [148]. These techniques are operator independent and mainly focus on efficient distribution of query processing, so that there is a notable research gap related to load shedding for pattern detection queries.

Advanced shedding strategies have been presented for relational data stream processing. Aurora [1] and Borealis [39] are among the first systems to include load shedding functionality. In this context, Tatbul *et al.* [178] proposed semantic load shedding to discard input tuples based on their contribution to the query result, measured by a notion of utility. Similar approaches have been presented for relational range queries [83] and path selection queries over XML data [186]. All these approaches, however, assume that the utility can be assessed precisely per tuple *without* considering the state of query evaluation. This is the case for traditional selection and aggregation queries, but not for pattern detection queries, for which the utility of load shedding largely depends on the state.

Load shedding was also explored for join operations of data streams. For binary equi-join operations, Kang *et al.* [101] showed how to allocate computing resources across two input streams based on arrival rates to maximise the number of output tuples. Load shedding decisions may also be taken based on value distributions of the join attributes [62]. In the same vein, GrubJoin [85] realises load shedding for multi-way streaming joins based on value distributions of attributes. Still, these approaches define cost models for individual elements of a stream, instead of assessing the contribution and consumption of state elements. Gedik *et al.* [84] propose an adaptive load shedding approach for windowing stream joins, which allows stream tuples to be stored in the windows and shed excessive CPU load by performing the join operations. However, their major goal is to maximise the output rate of relational operators instead of the maximum result quality of detected sequence patterns within a strict latency bound. Also, their adaptiveness is based on input rates, time correlation between the streams, and join directions. They did not consider a cost model combining consumptions and contributions with complex correlation predicates.

A first formulation of the load shedding problem for pattern detection queries has been presented by He *et al.* [94]. They point out that shedding algorithms developed for traditional data stream processing are not applicable for pattern detection queries and analyse the theoretical complexity of shedding problems. The presented algorithms, however, are limited to input-based shedding and optimise shedding decisions for a set of queries based on pre-defined weights. As a

result, they cannot optimise the load shedding problem for a single query. In addition, their load shedding algorithms lack the fine-granular control of the lossy results: There is a strong bias towards the queries of higher weights and those queries with smaller weights may not generate any results at all.

Recently, the idea of state-based load shedding [201] for CEP has been sketched: For a CEP query, the importance of an event is highly dynamic and largely depends on the state (*i.e.*, partial matches). Therefore, instead of dropping input events, the state-based load shedding technique discards partial matches to realise best-effort processing. In the same vein, Slo *et al.* proposed pSPICE [173] to drop partial matches. To this end, they model the pattern detection as a Markov reward process and use Markov chain to predict the importance of partial matches to determine the ones to be dropped. The eSPICE [174] load shedding framework, on the other hand, leverages a probabilistic model to learn the importance of events to select ones to be discarded. It considers an event type and its position in a time window. The main idea is that the importance of events is influenced by other events in the same pattern. The hSPICE [175] load shedding framework extends eSPICE [174] by considering the importance of both events and partial matches. To drop input events, in addition to the event type and its position in a time window, hSPICE [175] also considers the current state of the partial match associated with the input events. Although pSPICE [173], eSPICE [174], and hSPICE [175] strive for minimum quality loss while keeping the processing latency below a certain bound, they focus on either input-based load shedding or its state-based counterpart, but not a combination of the two.

Approximate query processing (AQP). AQP aims to estimate the result of queries [45]. It may be based on sampling, exploiting preprocessing of the data or knowledge about the expected workload [4, 144]. Other approaches rely on online aggregation [143, 190] and continuously refine query answers. Moreover, *sketches* [56, 58] may be employed for efficient, but lossy data stream processing. These approaches focus on traditional aggregation queries, though, so that they are inapplicable for pattern detection queries.

Recently, AQP has been investigated for sequential pattern matching. Li and Ge [120] showed how to learn characteristics of subsequences of a pattern from historic data to prioritise input data for processing. Their focus, however, is on matches that deviate slightly from what is specified in the pattern detection query, such as a perturbed order of events, instead of reducing the query processing latency.

3.4 Efficient Remote Data Integration in Event Stream Processing

There has been little research on a holistic approach that integrates remote data in event stream processing for low-latency analysis. Nonetheless, related topics such as prefetching and caching

techniques have been incorporated in data management systems for decades. Below, we review some important related techniques in diverse contexts and application scenarios.

Prefetching in relational database systems. Two surveys [131, 184] review general mechanisms for prefetching. Many of them are based on data access patterns derived from static and dynamic program analysis. Early work prefetches data blocks based on sequential access patterns for operators like 'scan' to hide I/O latency [176] and memory-cache latency [185]. Moreover, Chen *et al.* [49, 50] employ software-based prefetching to improve the indexing performance with B^+-Trees. Furthermore, combined with a group fetching technique, they also improved the performance of hash joins [48]. *Asynchronous memory access chaining* (AMAC) [106] uses prefetching to hide the memory access latency for in-memory databases. Recently, Menon *et al.* [128] propose a compiler pass for automatically inserting prefetch instructions that can handle irregular, data-dependent memory accesses.

Query result prefetching. Scalpel [31] performs semantic prefetching for query patterns involving batches, nesting, and data structure correlations. Ramachandra and Sudarshan [155] propose a program-analysis-based approach to automatically detect opportunities to prefetch query results across call procedures by inserting prefetch instructions at the earliest possible points. ForeCache [23] learns and models user behaviour to predict future queries for exploratory visualisation of large datasets. It partitions data to tiles, ranks them, and fetches the most likely candidates. However, it assumes certain data access patterns and a restricted set of operations.

Query result caching. Kossmann [111] surveyed optimisation techniques for distributed query processing, including caching mechanisms to reduce communication costs. Here, the focus has been on the design of database proxies to generate distributed query plans that efficiently synchronise cached data with the original database. Scalable query caching mainly focuses on consistency management instead of latency reduction. DBProxy [9], DBCache [28], and MTCache [114] rely on dedicated database proxies to generate distributed query plans that can efficiently combine cached data with the original database. These systems need built-in tools of the database system for consistency management. Ferdinand [80] is a proxy-based cooperative query result cache with fully distributed consistency management achieved by a publish / subscribe system. Blanco *et al.* [26] investigate query caching in the context of incremental search indices by developing invalidation predictors and defining metrics to evaluate invalidation schemes.

Caching in analytical big data systems. Systems like Spark employ distributed data abstractions for in-memory computation [193], using the *least recently used* (LRU) eviction algorithm for managing the data hold in memory. Moreover, *least reference count* (LRC) [192] improves the cache hit ratio by recording the reference count between the current data block and the awaiting blocks. *Most reference distance* (MRD) [146] extends LRC by recording the reference distance among data blocks in a job.

Web Caching. CachePortal [35] is a dynamic content cache management system that detects invalidation-based changes of web pages by analysing corresponding SQL queries. Gessert *et al.* propose Orestes [86], a scheme for leveraging HTTP caching for database records. Based thereon, Quaestor [87] considers the setting of a full DBaaS API including arbitrary query workloads.

The prefetching and caching approaches proposed in the above areas face similar tradeoffs with ESP with remote data, *e.g.*, between dataset sizes and their usage frequency. Naturally, though, the actual cost models and problem formulations to address these tradeoffs are very different, since the remote data access pattern in ESP is highly dynamic and largely depends on the state during query execution.

CHAPTER 4

HYBRID LOAD SHEDDING

This chapter addresses the problem of efficient pattern detection over event streams in an overload situation. Specifically, it aims to reduce pattern match evaluation latency. ESP engines that evaluate queries over event streams may face with unpredictable input rates and query selectivities. During short peak times, exhaustive processing is no longer reasonable, or even infeasible, and ESP engines shall resort to best-effort query evaluation and strive for optimal result quality while staying within a latency bound. In traditional data stream management systems, this is achieved by load shedding that discards some stream elements without processing them based on their estimated utility for the query result. However, such input-based load shedding is not always suitable for pattern detection queries. It assumes that the utility of each individual element of a stream can be assessed in isolation. For pattern detection queries, however, this utility may be highly dynamic: Depending on the presence of partial matches, the impact of discarding a single event can vary drastically. In this chapter, we therefore complement input-based load shedding with a state-based technique that discards state elements—the partial matches. We introduce a hybrid shedding model that combines both input-based and state-based shedding to achieve high result quality under constrained resources.

This chapter is organised as follows: We first illustrate the problem of load shedding in event stream processing with examples in Section 4.1. Section 4.2 formally defines the load shedding problem and complement input-based load shedding with a new state-based technique that discards partial matches. The selection of shedding elements is guided by a cost model to access the importance of partial matches and input events. We show how the setting can be formulated as a *Knapsack* problem [104] and how shedding strategies are based on its solution. Section 4.3 discusses implementation considerations for hybrid load shedding related to the cost model granularity, its estimation and adaptation, and approximation schemes. Comprehensive

evolutions are detailed in Section 4.4, and Section 4.5 summarises this chapter.

4.1 Problem Illustration

Recall from Section 1.1 and Section 2.2 that pattern detection queries are stateful: The number of state elements may grow exponentially in the number of processed events and common evaluation algorithms show an exponential worst-case runtime complexity [196]. The inherent complexity of pattern detection queries imposes challenges especially in the presence of dynamic workloads. When input rates and query selectivities are volatile, hard to predict, and change by orders of magnitude during short peak times, preallocating sufficient computational resources is no longer reasonable. Permanent over-provisioning of resources to cope with peak demands incurs high costs or is even infeasible.

Moreover, ESP applications differ in the required guarantees for the *latency* and *results quality* of pattern detection. While ESP engines strive for low-latency processing, the precise requirements are application-specific. How the usefulness of query matches deteriorates over time varies greatly, and matches may become completely irrelevant after a certain latency threshold. Yet, depending on the application, it may be acceptable to miss a few matches if the latency for the detected matches is much lower. We illustrate the above properties with example applications:

Fraud detection. To detect fraudulent credit card transactions, ESP queries identify suspicious patterns (*e.g.*, a credit card is suddenly used at various locations). The event streams vary in their input rates and query selectivities, *e.g.*, due to data breaches being exploited. While such variations can hardly be anticipated, there are tight latency bounds for processing: In 25ms, a credit card transaction needs to be cleared or flagged as fraud [71]. Also, payment models of companies such as Fraugster [78] that penalise false positives make it impossible to simply deny all transactions in sudden overload situations. Hence, an ESP engine shall resort to best-effort processing, detecting as many fraudulent transactions as possible within the strict latency bound.

Urban transportation. Operational management may exploit movements of buses and shared cars or bikes, as well as travel requests and bookings by users [16]. Yet, the resulting streams are unsteady, as, *e.g.*, a large public happening leads to spikes in event rates and query selectivities (many ride requests to a single location). Also, the utility of pattern detection deteriorates quickly over time, whereas the result quality may be compromised for some queries. For instance, queries to correlate requests and offers for shared rides must be processed with sub-second latency to achieve a smooth user experience. Yet, in overload situations, detecting *some* matches *quickly* is more profitable than detecting *all* matches *too late* or investing in resource over-provisioning.

Example 4.1. *Consider CitiBike, a New-York-based bike sharing provider that published bicycle trip data of its 146k members [54]. Bikes are rented through a smartphone application, where users search bicycles at nearby docking stations, start rides, and finish trips, again at other docking*

stations. Since bikes quickly accumulate in certain areas, the operator moves more than six thousand bicycles per day among stations. Hence, the detection of 'hot paths' of bike trips promises to improve operational efficiency. 'Hot paths' consist of stations where bicycles are accumulated faster than at other stations. They indict the trend of movement for the bike fleet.

```
PATTERN SEQ(BikeTrip+ a[], BikeTrip b)
WHERE a[i+1].bike=a[i].bike AND b.end∈{7,8,9}
AND a[last].bike=b.bike AND a[i+1].start=a[i].end
WITHIN 1h
RETURN (a[1].start,a[i].end,b.end)
```

Listing 4.1: Query to detect 'hot paths' of stations.

Listing 4.1 shows a pattern detection query to detect such 'hot paths', using the SASE query language [5]: Within an hour time window, a bike is used in several subsequent trips, ending at particular stations, No. 7, No.8, and No.9 ({7,8,9}). Here, the Kleene closure operator detects arbitrary lengths of paths.

Evaluating the query over the citibike dataset [54] reveals a drastic spike in the number of partial matches maintained by the ESP engine, see Figure 4.1. While higher resource demands may eventually be addressed by scaling out the system, load shedding helps to keep the system operational in peak situations by detecting most of the matches with low latency.

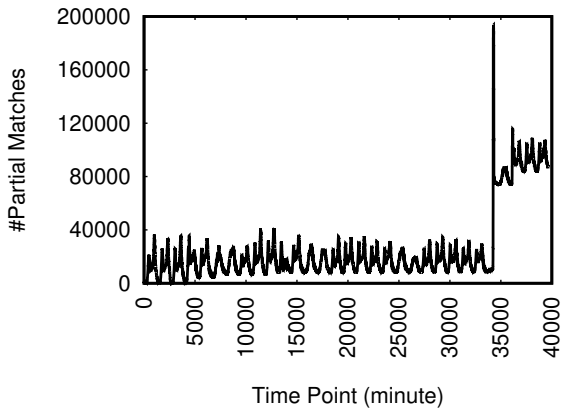


Figure 4.1: Number of partial matches over time for the evaluation of the pattern detection query in Listing 4.1.

Moreover, scaling-out of stream processing infrastructures provides only limited flexibility. For instance, resharding a stream in Amazon Kinesis to double the throughput may take up to an hour if the stream comprises around 100 shards already [8]. In addition, while scaling the system indeed improves performance when the number of partial matches is stable at a higher level load after the spike, it is not effective for dynamic and unpredictable loads. It is also not

economically efficient when most of the computing resources are idle, because overload situations may last for short time spans which are even shorter than state migration time during scaling.

Against this background, ESP engines shall employ *best-effort processing*, when resource demands peak dynamically [94]. Using resources effectively, an ESP engine shall maximise the result quality of pattern detection, while satisfying a latency bound. Best-effort stream processing may be achieved by *load shedding* [178] that discards some elements of the input stream. Simple strategies that shed events randomly have been implemented for many state-of-the-art streaming systems, *e.g.*, Heron [74], and Kafka [21]. More advanced strategies discard elements based on their estimated utility for traditional selection and aggregation queries [83, 148, 178, 186].

However, such input-based load shedding is not always suited for pattern detection queries [94]. The reasons become clear when reviewing three fundamental questions for the design of load shedding mechanisms, as brought forward in [178]:

(Q1) *When to shed?* An overload situation needs to be detected quickly for prompt load shedding.
(Q2) *What to shed?* One needs to decide which data shall be shed at which component in a system.
(Q3) *How much to shed?* One needs to decide on the amount of data to shed per overload situation.

Approaches for load shedding in traditional data stream processing systems answer Q1 by relating input rates of streams to processing rates of query operators [83, 178]. Following [94], however, such an approach cannot be realised for pattern detection queries. The reason is that the sequential pattern operator is very sensitive to a volatile query selectivity, which differs drastically per event depending on the query evaluation state with different maintained partial matches. As such, the processing rate of an ESP engine is also highly volatile.

Question Q2, the selection of data to discard, is commonly approached based on the selectivity of relational operators for data streams [83, 178, 186]. In essence, these approaches exploit that the impact of shedding can be determined rather accurately per stream element. For pattern detection queries, however, this is not the case. The utility of a single event, again, largely depends on the state of an ESP engine in terms of currently maintained partial matches. Under different sets of partial matches, an event may lead to none or a large number of complete matches.

The decision about the amount of data to shed, question Q3, is typically governed by the difference of the input rates of streams and the processing rates of query operators [83, 148, 178, 186]. The bigger this difference is, the more data is shed. Again, large fluctuations in query selectivity render such an approach unsuitable for ESP engines .

The above difficulties lead to the following observation: *"The CEP load shedding problem is significantly different and considerably harder than the problems previously studied in the context of general stream load shedding"* [94].

Example 4.2. *We illustrate the challenges of realising load shedding in event pattern detection in Figure 4.2, using the pattern detection query in Listing 4.1. Assume that we detect 'hot paths' that*

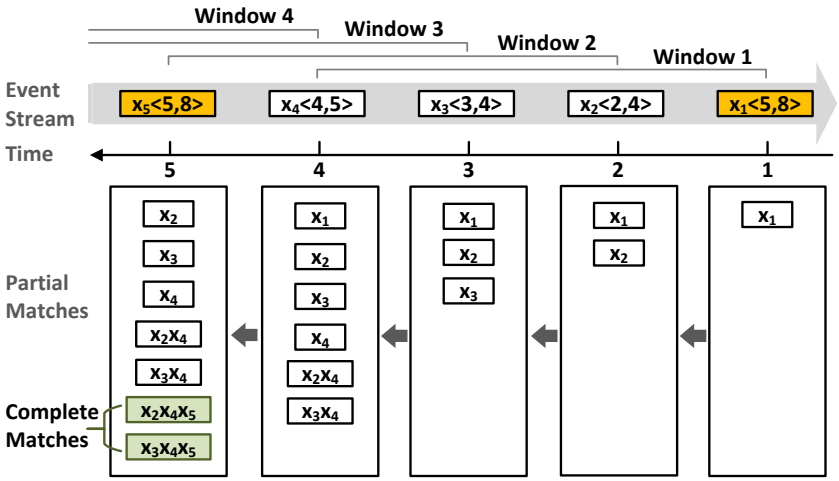


Figure 4.2: Example partial matches maintained by the pattern detection query in Listing 4.1.

are indicated by three trip events within a time window of four time units. The stream comprises event x_i of the schema ⟨start station,end station⟩, with i being the timestamp. Events x_1 and x_5 denote the same trip ⟨5,8⟩ that is the bike trip from docking station No.5 to station No.8. However, due to different sets of partial matches maintained at time points 1 and 5, respectively, only event x_5 contributes to complete matches. Hence, an event with payload ⟨5,8⟩ may be discarded at time 1, whereas discarding it at time 5 implies lost matches in three time windows, i.e. window 2, 3 and 4.

This problem illustration shows that previous input-based load shedding techniques and scaling systems are insufficient for pattern detection query processing in dynamic and unpredictable overload situations. To overcome this problem, this chapter argues for a fundamentally new perspective on load shedding for ESP. Since the state of query evaluation governs the complexity of query processing, we introduce the *state-based load shedding* that discards partial matches, thereby maximising the result quality of query processing in overload situations. Based thereon, we further propose a *hybrid* approach that combines state-based and input-based load shedding.

4.2 Foundations of Hybrid Load Shedding

Both input-based and state-based load shedding have their advantages and disadvantages. State-based shedding offers more fine-granular control of latency reduction and result quality, compared to shedding input events. Discarding a partial match only affects matches that could be derived from it, whereas dropping a single event may lose hundreds of matches. However, input-based shedding is generally more efficient: A discarded event is not processed at all, whereas a partial match already incurred some computational effort. When striving for a balance between result quality and efficiency for a given ESP application for pattern detection, we propose a hybrid approach that combines both state-based and input-based load shedding.

4.2.1 The Load Shedding Problem in Pattern Detection Queries

To realise load shedding in an ESP engine, we revisit the three questions raised in Section 4.1: *when* to conduct load shedding (Q1), and *what* (Q2) and *how much* (Q3) data to shed. We first define the overload situation of an ESP engine.

Definition 4.1 (Overload situation)**.** An ESP engine is in an overload situation if the query processing latency at stream prefix $S(..k)$, $\ell(k)$, exceeds a predefined latency bound θ.

The latency of query processing, $\ell(k)$, and the latency bound θ are subject to application-specific requirements (Section 4.1). We thus consider a model in which load shedding is triggered when the ESP engine is overloaded (the latency $\ell(k)$ exceeds a bound θ (Q1)). In practice, the effect of load shedding may materialise only with a minor delay, so that the bound θ shall be chosen slightly smaller than the bound that renders matches irrelevant in the application domain.

Solutions for the decisions of *what* and *how much* to shed (Q2 and Q3) have to consider the result quality of query evaluation. We assess this quality as the loss in complete matches induced by shedding. Let $R = \langle C(1),\ldots,C(k)\rangle$ be the results (sets of complete matches) obtained when processing a stream prefix $S(..k)$, and let $R' = \langle C'(1),\ldots,C'(k)\rangle$ be the results obtained when processing the same prefix, but with load shedding. The quality loss is defined as $\delta(k) = |R \setminus R'|$, which means the number of the lost complete matches due to load shedding.

Problem 4.1. *The problem of load shedding in ESP for pattern detection is to reduce the pattern evaluation latency $\ell_{match}(k)$ to ensure that when evaluating a pattern detection query for a stream prefix $S(..k)$, it holds that $\ell(k) \leq \theta$ for $k \geq 1$ and the quality loss $\delta(k)$ is minimal.*

In the remainder of this chapter, we focus on queries that are *monotonic* in the input streams and the partial matches. We define monotonic queries as follows:

Definition 4.2 (Monotonic queries)**.** Let $P(k)$ and $P(l)$ be the set of partial matches after evaluating query Q over stream prefix $S(..k)$ and $S(..l)$, $k < l$. A query is monotonic in the stream, if the partial matches $P(k')$ obtained when evaluating Q over an order preserving projection $S'(..k')$ derived by removing some events of $S(..k)$ is a subset of the original ones, $P(k') \subseteq P(k)$. A query is monotonic in the partial matches, if the complete matches $C'(l)$ obtained when evaluating Q over $S(..l)$ using only a subset $P'(k) \subseteq P(k)$ of the partial matches yields a subset of the original complete matches, $C'(l) \subseteq C(l)$.

Put differently, for monotonic queries, removing input events may only reduce the set of partial matches and removing partial matches may only reduce the set of complete matches.

Pattern detection queries that include conjunction, sequencing, Kleene closure, correlation conditions, and time windows are monotonic under a greedy event selection policy (*e.g.*, *skip-till-any-match* [5], see Section 2.2.2). The same holds true for common aggregations, such as a

query assessing whether the average of attribute values of a sequence of events is larger than a threshold. Because for a sequence of events that is not subject to this predicate, shedding some of its events or partial matches can only make the average value smaller than the threshold, but will not generate any extra matches. However, it is non-monotonic for the query to detect whether the average of attribute values of a sequence of events is smaller than a given threshold. Because for a sequence of events that is not subject to the predicate, shedding some of its events or partial matches reduces the average value, which may satisfy the predicate and generate extra matches.

Intuitively, a greedy event selection policy leads to all possible combinations of events and, hence, aggregate values being represented by partial matches derived from the original stream. Therefore, removing an input event or a partial match can only lead to missing complete matches, but will never create additional partial or complete matches. Counter-examples for the monotonicity are queries with more selective policies, *e.g.*, those that require *strict contiguity* [5] of events, and negation operators. Note that greedy event selection policies represent the most challenging scenario from a computational point of view, so that load shedding is particularly important.

For monotonic queries, it holds that $C'(i) \subseteq C(i)$, $1 \leq i \leq k$, so that the quality loss caused by load shedding, $\delta(k) = |R(k) \setminus R'(k)| = \sum_{1 \leq i \leq k} |C(i) \setminus C'(i)|$, is the *recall loss*. That is the total number of the lost complete matches due to shedding. Any shedding decision (*what* and *how much*) shall therefore aim at minimising this loss in the recall.

4.2.2 Hybrid Shedding Approach

To address the load shedding problem, we propose a new perspective on deciding *what* (Q2) and *how much* (Q3) data to shed. We introduce *hybrid* load shedding that discards both input events and partial matches. Taking up our formalisation of query evaluation as a function f_Q (see Equation 2.1) that is applied to the next event of the input stream and the current partial matches, we distinguish *input-based* shedding and *state-based* shedding, formalised by two functions ρ_I and ρ_S:

Definition 4.3 (Hybrid load shedding for pattern detection queries)**.**

$$\rho_I(e) \mapsto \begin{cases} e \\ \bot \end{cases} \quad \text{and} \quad \rho_S(P) \mapsto P', \, s.t. \; P' \subseteq P. \tag{4.1}$$

That is, ρ_I filters a single input event and potentially discards it (denoted by $\bot$), whereas ρ_S filters a set of partial matches (state elements at the time of processing), potentially discarding a subset of them. Based thereon, processing of a streaming event $S(k+1)$ is represented in our formal model as the application of the evaluation function f_Q to the results of load shedding,

$$f_Q(\rho_I(S(k+1)), \rho_S(P(k)), D(k)) \mapsto P(k+1), C(k+1). \tag{4.2}$$

Here, we assume that $f_Q(\perp, \rho_S(P(k)), D(k)) \mapsto \rho_S(P(k)), \emptyset$, *i.e.*, shedding an input event does not change the maintained partial matches, nor does it generate complete matches. Figure 4.3 links the two shedding strategies to the aforementioned automata-based and operator-tree-based computational models for ESP.

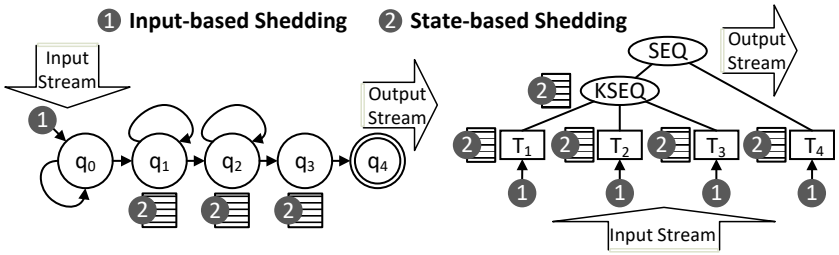


Figure 4.3: Input-based vs. state-based shedding for automaton and tree-based execution models.

4.2.3 Cost Model

The above approach for hybrid load shedding calls for an instantiation of the input-based and state-based shedding functions. This requires determining the amount of data to shed to ensure that the latency bound is satisfied as well as assessing the utility of input events and partial matches to minimise the loss in the recall. To this end, we introduce a cost model.

Input-based and state-based techniques for load shedding differ in the granularity with which the recall and computational effort of query evaluation are affected. Input-based shedding offers coarse-granular control, since a discarded event cannot be part of *any* partial or complete matches. It yields comparatively large savings of computational resources (preventing an exponential number of partial matches), while it may also have a large negative impact on the recall of query evaluation results (an exponential number of complete matches may be lost). State-based shedding, on the other hand, provides relatively fine-granular control, as the events of a discarded match may remain part of other partial matches. Consequently, the resulting computational savings and recall loss are also comparatively small.

The above difference in shedding granularity is important to handle different levels of variance in query selectivity. With small variance, the utility of an input event can be assessed precisely and input-based shedding is preferred: It avoids spending *any* computational resources for processing events with low utility. For a query with a large variance in selectivity, however, an assessment of the utility per event is inherently imprecise, so that resorting to state-based shedding promises a higher recall at the expense of smaller resource savings.

To reason on the impact of shedding strategies on the quality of query evaluation and the imposed computational effort, we define a cost model. Striving for a fine-granular assessment, this model is grounded in partial matches. However, it later also serves the selection of input events for input-based load shedding.

Consider the moment in time after a stream prefix $S(..k)$ has been processed. At this moment, we assess a partial match along the following dimensions:

Contribution. We assess the *contribution* of a partial match to the query result, *i.e.*, to the construction of complete matches. It is defined by the number of complete matches that are generated from it. With $C(k+1), C(k+2), \ldots$ as the complete matches derived in the future, the contribution of a partial match $p = \langle e_1, \ldots, e_n \rangle \in P(k)$ is defined as:

Definition 4.4 (Contribution)**.**

$$\Gamma^{+}(p) = \left| \left\{ \langle e'_1, ..., e'_m \rangle \in C(i) \mid i > k \wedge \forall 1 \leq j \leq n : e'_j = e_j \right\} \right|. \tag{4.3}$$

Consumption. We assess the *consumption* of computational resources induced by a partial match by considering all partial matches that are derived from it. Unlike for the contribution defined above, we capture the resource consumption explicitly instead of abstracting it by the count of derived matches. The rationale behind is that the resource consumption may vary greatly between partial matches. For instance, the number and complexity of predicates that need to be evaluated for a partial match per input event may differ drastically.

We capture the resource cost of a partial match p by a function $\Omega(p) \mapsto o$, where $o \in \mathbb{N}$. The exact value may be defined as the number of query predicates to evaluate for p (to capture runtime costs) or as its length (to capture the memory footprint). With $P(k+1), P(k+2), \ldots$ as the sets of partial matches constructed in the future, the consumption of a partial match $p = \langle e_1, \ldots, e_n \rangle \in P(k)$ is defined as:

Definition 4.5 (Consumption)**.**

$$\Gamma^{-}(p) = \sum_{\substack{\langle e'_1, \ldots, e'_m \rangle \in \bigcup_{i>k} P(i) \\ \forall\ 1 \leq j \leq n:\ e'_j = e_j}} \Omega(\langle e'_1, \ldots, e'_m \rangle). \tag{4.4}$$

Contribution and consumption are well-defined, since complete and partial matches obey the time window of a query (see Definition 2.7). This limits the number of matches that a single partial match can generate. However, the contribution and consumption of a partial match are measures in the future and can only be calculated in retrospect. We therefore later discuss how to construct effective and efficient estimators for these measures.

4.2.4 Shedding Set Selection

Once the contribution and consumption values are known or estimated for partial matches, load shedding is performed based on the following idea. The severity of the violation of the latency bound for query evaluation shall govern the severity of load shedding: The more the latency bound is violated, the higher the relative share of data that is shed. Specifically, we consider the

extent of latency violation as a lower bound for the extent of resource consumption that shall be saved by discarding partial matches.

Consider the situation that shedding has been triggered after a stream prefix $S(..k)$ had been processed. Then, the relative extent of the latency violation is given as $(\ell(k)-\theta)/\ell(k)$. Let $P(k)$ be the set of current partial matches. For a partial match p, we assess its relative amount of consumed computational resources compared to all partial matches, $\Delta^{-}(p,P(k))$ in Equation 4.5:

Definition 4.6 (Relative consumption)**.**

$$\Delta^{-}(p,P(k)) = |\Gamma^{-}(p)| / \sum_{p' \in P(k)} \left|\Gamma^{-}(p')\right|. \tag{4.5}$$

Taking this relative extent of latency violation as a lower bound for the relative amount of resource consumption to save, we control the amount of data to shed. That is, we select a subset of partial matches $\mathbb{D} \subseteq P(k)$, called a *shedding set*, such that:

$$\sum_{p \in \mathbb{D}} \Delta^{-}(p,P(k)) > \frac{\ell(k)-\theta}{\ell(k)}. \tag{4.6}$$

While the above formulation provides guidance on the partial matches to consider, shedding shall aim at minimising the loss in recall of query evaluation (see Problem 4.1). This loss is defined in terms of the missing complete matches due to shedding, which links it with our above notion of contribution of a partial match. We therefore assess the relative contribution of a partial match $p \in P(k)$ to avoid any loss in recall, as showed in Equation 4.7:

$$\Delta^{+}(p,P(k)) = \left|\Gamma^{+}(p)\right| / \sum_{p' \in P(k)} \left|\Gamma^{+}(p')\right|. \tag{4.7}$$

Based thereon, we phrase the selection of a shedding set from the set of partial matches as an optimisation problem (Equation 4.8) to guide the decisions on what and how much data to shed:

$$\begin{aligned} &\textbf{select } \mathbb{D} \subseteq P(k) \textbf{ that minimizes } \sum_{p \in \mathbb{D}} \Delta^{+}(p,P(k)) \\ &\textbf{subject to } \sum_{p \in \mathbb{D}} \Delta^{-}(p,P(k)) > \frac{\ell(k)-\theta}{\ell(k)}. \end{aligned} \tag{4.8}$$

The above problem is a application of the Knapsack problem [104]. Its capacity is defined by the extent of latency violation, which varies among different moments in which load shedding is triggered. Hence, the problem needs to be solved in an online manner. However, Knapsack problems are NP-hard [104]. To avoid the computational overhead of solving Equation 4.8, we later show how to obtain an approximated solution in Section 4.3.

4.2.5 Shedding Functions

When load shedding is triggered, a shedding set is computed as detailed above. It is then used to define different shedding strategies by instantiating the functions ρ_I and ρ_S introduced in Equation 4.1 for input-based and state-based shedding.

State-based shedding is achieved by not discarding input events and removing all partial matches of the shedding set $\mathbb{D}$ from the ESP engine. Then, ρ_I is the identity function, while ρ_S is defined as $\rho_S(P(k)) \mapsto P(k) \setminus \mathbb{D}$. For practical considerations, state-based shedding should not be triggered again immediately, *i.e.*, by the latency $\ell(k+1)$ being above the threshold, but only after some delay $j \in \mathbb{N}$, *i.e.*, by $\ell(k+j)$ the earliest. The intuition is that the effects of shedding first need to materialise, before it is assessed whether further shedding is still needed.

Input-based shedding is achieved by not discarding partial matches (ρ_S is the identity function), but deriving the filter ρ_I for input events from the partial matches in the shedding set $\mathbb{D}$. Intuitively, the partial matches that are most suitable for load shedding are exploited to derive the conditions based on which input events shall be discarded. Recall that events have a schema, $A = \langle A_1, \ldots, A_n \rangle$, so that each event is an instance $e = \langle a_1, \ldots, a_n \rangle$ of this schema (see Section 2.1.1). Given the set of events that are part of matches in the shedding set, defined as $E_{\mathbb{D}} = \{e \mid \exists \langle e'_1, \ldots, e'_m \rangle \in \mathbb{D}, 1 \leq i \leq m : e'_i = e\}$, the input-based shedding function is defined as:

$$\rho_I(e) \mapsto \begin{cases} e & \text{if } \psi(e) \notin E_{\mathbb{D}}, \\ \bot & \text{otherwise.} \end{cases} \tag{4.9}$$

Here, ψ is the mapping of value predicates that correlate input event e and partial matches. Put differently, an event filter can be derived based on the shedding set $\mathbb{D}$ and the value predicates in the query. For instance, if the value predicate regarding to input event type `e` is `t.v+e.v=10` (`t.v, e.v` $\in \mathbb{N}$), and event type `t`'s attribute `v`'s value ranges from 1 to 5 and is incorporated in the shedding set $\mathbb{D}$, we can infer that events of type `e` with attribute `v` that is less than 5 can not satisfy predicate `t.v+e.v=10`. Therefore, the event filter $\rho_I(e)$ for type `e` can be materialised as `e.v<5`.

Input-based shedding by ρ_I applies to the single input event $S(k+1)$, the $(k+1)$-th event in stream S, that is to be handled next, after processing the stream prefix $S(..k)$. Hence, unlike for state-based shedding, to have any effect, input-based shedding needs to be applied for a certain interval. The length of this interval is determined by the latencies $\ell(k+1), \ell(k+2), \ldots$ observed after load shedding was triggered. Once the latency bound is satisfied, $\ell(k+j) \leq \theta$ for some $j \in \mathbb{N}$, input-based shedding is stopped.

Hybrid shedding combines the above two strategies. The shedding set $\mathbb{D}$ is defines function ρ_S to remove partial matches and also serves as the basis for function ρ_I for input-based shedding. Again, the latter function is applied for some interval based on the observed latencies. A major advantage of hybrid shedding is that it does *not* require explicit balancing of input-based and

state-based shedding, *e.g.*, by a fixed weighting scheme. Since both strategies are grounded in the same cost model, balancing is achieved directly by the unified assessment of the consumption and contribution of partial matches and, thus, input events.

4.3 Implementations of Hybrid Load Shedding

When implementing our cost model for hybrid load shedding, several practical aspects have to be considered. In this section, we first elaborate on the granularity of the cost model (Section 4.3.1), before introducing strategies for its efficient estimation and adaptation (Section 4.3.2). Finally, we discuss how to approximate shedding sets (Section 4.3.3) and how to manage state (partial matches) efficiently (Section 4.3.4).

4.3.1 Granularity of the Cost Model

Our cost model for partial matches (Section 4.2.3) is very fine-granular to enable precise shedding decisions. Yet, considering *each* partial match at *any* point in time leads to very large computational overhead: The selection of shedding sets (Section 4.2.4) is then based on a Knapsack problem with a large quantity of items, while input-based shedding (Section 4.2.5) also becomes costly, due to a potentially complex derivation of input event filters. We therefore tune the granularity of the cost model through temporal and data abstractions, striving for a balance between the precision of the cost estimation and the computational overhead.

Temporal abstractions. Even though contribution and consumption of matches may change when a single event is processed, there are typically only a few important change points over the entire lifespan of a partial match. Since exact measurements are not needed for shedding decisions, a *partial order* of the cost of partial matches is sufficient. Therefore, we employ the temporal abstraction of *time slices*. The query time window, which determines the maximal time-to-live of a partial match, is split into a fixed number of intervals. Each interval is a time slice. The cost model is then instantiated per time slice, rather than per time point. Specifically, given m time slices and a partial match $p = \langle e_1, \ldots, e_n \rangle \in P(k)$, p's contribution and consumption values are estimated by values at time point $\lceil \frac{t_n - t_1}{\lceil \tau/m \rceil} \rceil$. That is, with τ as the query time window, the cost models are materialised at every $\lceil \tau/m \rceil$ time interval.

Data abstractions. Partial matches that overlap in their events and the events' attribute values are likely to show similar contribution and consumption values. We therefore lift the cost model to *classes* of partial matches. Each class is characterised by a predicate over the attribute values of the respective events. For instance, in Example 4.1, partial matches for which the last event denotes a trip ending at one of stations {3,4,5,6} may have similar consumption and contribution values. Assessing the cost model per class, shedding sets (Section 4.2.4) and shedding functions

(Section 4.2.5) are also realised per class. If a class is part of the selected shedding set, the function for input-based shedding uses the predicate of this class to filter related input events.

4.3.2 Estimating the Cost Model

Recall that the value of contribution and consumption can only be computed in retrospect (see Section 4.2.3). Therefore, to enable decision-making based on our cost model, we need to estimate the contribution and consumption of partial matches. Such an estimation is first performed offline, using historic data, while the estimates are then adapted in an online manner.

Offline estimation. We evaluate a query over historic data and record partial matches and complete matches to derive the contribution and consumption of each match. For each partial match, its contribution value is computed by checking how many times its payload was incorporated into complete matches, in the relevant time slice of a time window. Its consumption value is computed in a similar way, by checking against both partial matches and complete matches.

For each *category* of partial matches of the execution model (defined by an NFA state or a buffer in an operator tree, see Section 2.2.2), the partial matches are then clustered based on their contribution and consumption values per time slice. Here, clustering algorithms that work with a fixed number of clusters (*e.g.*, K-means) enable direct control of the granularity of the employed data abstraction: Each cluster induces one class for the definition of the cost model. We employ the *gap statistic* technique [181] to estimate an optimal number of clusters. The contribution and consumption per class (cluster) are computed as the 90^{th} percentiles of the values among all the partial matches in that class. We keep a lookup table that maps the class labels to these values.

To use the class estimates in online processing, we need an efficient mechanism to classify a partial match immediately after its creation. Put differently, during pattern detection query processing, we need to maintain the newly created partial matches into their corresponding classes learned from the offline estimation. We therefore train a classifier for the partial matches of the classes obtained for each of the categories of the execution model, *i.e.*, one classifier per category. The classifier uses the attributes of partial matches that appear in the query predicates as features for training. The choice of the classification algorithm is of minor importance, assuming that the classifier can be evaluated efficiently. In this dissertation, we employ balanced decision trees, setting the maximal depths to the numbers of classes for the respective categories.

Online adaptation. An instantiation of the cost model based on online clustering and classification is infeasible. Because the computational overhead will thwart any benefit of load shedding in the first place. However, the estimates per class and time slice may be monitored and adapted. Initially, we start with the classifiers obtained through offline estimation and the mapping of cluster labels to contribution and consumption values. Once a partial match is generated, it is classified using the classifier of the respective category. As a consequence, partial matches are maintained in different classes. However, the contribution and consumption values may change

as more events are processed because of the dynamics of stream and hence, the drift of the contribution and consumption values. Therefore, we monitor updates to these values by streaming counts: We maintain the contribution and consumption per class via a lookup table for each category. Upon the creation of a match, the counts for the class and time slice of the originating partial matches are incremented in the lookup table (consumption values). If the new match is a complete match, the counts for contribution values are also incremented. At the end of each time slice, the new contribution value of a class is calculated as $\Gamma^{+}_{new} = (1-w)\Gamma^{+}_{old} + w\Gamma^{+}_{incremented}$. Here, w is the weight of incremented contribution and large values increase the pace of value updating (we set $w = 0.5$). Consumption values are updated following the same procedure. This way, adaptation is based on sketches [57] for efficient streaming counts.

4.3.3 Approximated Shedding Sets

Selecting a shedding set requires solving the Knapsack problem (see Section 4.2.4), which is NP-hard [104]. To solve it efficiently online, we prune the search space by coarsening the cost model granularity (see Section 4.3.1). We found that by lifting the cost model from millions of individual partial matches to tens of classes of them, computation of shedding sets using dynamic programming [147] takes a few nanoseconds on modern CPUs, which is feasible for online processing in overload situations.

If the number of classes is large due to results quality requirements, the computational overhead is high. Therefore, dynamic programming approaches become infeasible and approximations shall be applied. One approximation is to reuse previous shedding decisions. Specifically, the size of the shedding set (the capacity of the Knapsack problem) is determined by the extent of latency violation (see Section 4.2.4). For similar latency violations, we may employ the same shedding set. To this end, we maintain the mapping of latency violations and shedding sets (indexes of classes of partial matches). For repeated load shedding, shedding sets may be reused, assuming stable contribution and consumption values per class and time slice for a short time duration. The other approximation is about the Knapsack problem itself. In this dissertation, we aim at approximated solutions, see [46], rather than exact solutions using dynamic programming. A simple greedy strategy is to select classes of partial matches in the order of their contribution and consumption ratios, until the capacity bound of the Knapsack problem is reached.

4.3.4 Managing Partial Matches Efficiently

In order to access and maintain partial matches efficiently in main memory, we employ indexing and early curbing techniques for partial matches.

Indexing. Common ESP engines build indexes over the attribute values of events based on the query predicates to evaluate. To enable efficient access of matches in the construction of shedding sets, we further define indexes based on the predicates that define the classes of the cost model.

We maintain the partial matches in the main memory in a column-store manner. Specifically, each attribute is a column. The materialised value of each attribute of partial matches is stored in consecutive memory blocks for efficient random access. Indexes are built for related attributes and are sorted. We maintain the mapping of partial matches' classes and their starting memory address. The actual memory address of an attribute value is computed by the starting address of the memory block and its offset.

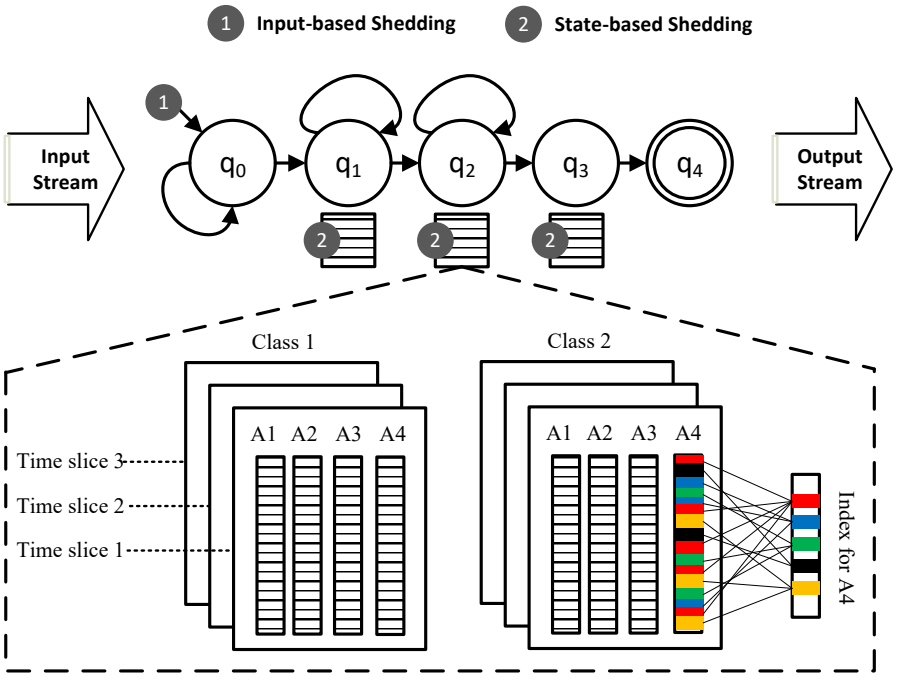


Figure 4.4: Indexing partial matches.

Figure 4.4 illustrates the organisation of partial matches of category q_2 in the automata-based ESP engine shown in Figure 4.3. Here, the partial matches are split into two classes and three time slices. Each partial match consists of four attributes (A_1,A_2,A_3 and A_4). The values of each attribute are stored in consecutive main memory blocks. Attribute A_4 is indexed. Different colours represent different values. Same values are indexed by a single pointer for fast lookup. *i.e.*, if a red data item is not selected by value predicates, all partial matches whose A_4 values are red shall be ignored directly without evaluating query predicates on the payload.

Early curbing. Once a shedding set is computed, the respective partial matches are discarded. However, processing the current input event, new partial matches that would be part of the shedding set may be generated. Put differently, these partial matches are created first but will be discarded later, which wastes computing resources. To avoid this effect, we curb the creation of these partial matches in the first place during shedding, based on the predicates that define the classes of partial matches in the shedding set.

Table 4.1: Synthetic datasets for load shedding.

	Attribute	Value Distribution
DS1	Type	$\mathcal{U}(\{A,B,C,D\})$
	ID	$\mathcal{U}(1,10)$
	V	$\mathcal{U}(1,10)$ (or controlled)
DS2	Type	$\mathcal{U}(\{A,B,C,D\})$
	ID	$\mathcal{U}(1,10)$
	A.x, A.y, B.x, B.y	$\mathbb{P}(0<X\leq 2)=33\%$, $\mathbb{P}(2<X\leq 4)=67\%$
	B.v	$\mathbb{P}(X=2)=33\%$, $\mathbb{P}(X=5)=67\%$
	C.v	$\mathbb{P}(X=3)=33\%$, $\mathbb{P}(X=5)=67\%$
	D.v	$\mathbb{P}(X=5)=33\%$, $\mathbb{P}(X=2)=67\%$

4.4 Evaluations

This section presents the evaluation of our approach for hybrid load shedding. We first give details on the experimental setup (Section 4.4.1), before turning to the overall effectiveness and efficiency (Section 4.4.2) of the proposed hybrid load shedding technique. Section 4.4.3 discusses the sensitivity analysis of a range of parameters. Finally, we apply hybrid load shedding to two real-world use cases (Section 4.4.4).

4.4.1 Experimental Setup

Datasets and queries. For controlled experiments, we generated two synthetic datasets as detailed in Table 4.1. The first dataset, DS1, comprises events with a three-valued, uniformly distributed payload: A categorical type, a numeric event ID, and a numeric attribute V. This dataset enables us to evaluate queries of a common structure: Queries test for sequences of events of particular types that are correlated by an ID, whereas further conditions may be defined for attribute V. To explore the impact of diverse resource costs of matches (see Section 4.2.3), we generated a second dataset, DS2. As shown in Table 4.1, the events' payload includes multiple numeric attributes with the values being drawn from partially overlapping ranges.

We execute queries Q1, Q2, and Q4 of Listing 4.2 over dataset DS1, and query Q3 over dataset DS2. Note that Q1-Q3 are monotonic, whereas Q4 is not. The queries will be explained further in the respective subsections.

We further use the real-world dataset of *citibike* [54], introduced already in Example 4.1. Here, we use a pattern detection query to detect 'hot paths' with low latency. As a second real-world dataset, we use the Google Cluster-Usage Traces [160] for cluster monitoring. We employ a pattern detection query to detect abnormal scheduling behaviours of a 12.5k-machine cluster with low latency. The details of these two use cases are presented in Section 4.4.4.

```
Q1: PATTERN SEQ(A a, B b, C c)
    WHERE a.ID=b.ID AND a.ID=c.ID AND a.V+b.V=c.V
    WITHIN 8ms
Q2: PATTERN SEQ(A a, A+ b[], B c, C d)
    WHERE a.ID=b[i].ID AND a.ID=c.ID AND a.ID=c.ID AND b[i].V=a.V
    AND a.V+c.V=d.V
    WITHIN 1ms
Q3: PATTERN SEQ(A a, B b, C c, D d)
    WHERE a.ID=b.ID AND a.x≥ b.v/2 AND a.x≤b.v AND a.y≥ b.v/2
    AND a.y≤b.v AND b.ID=C.ID AND c.ID=d.ID AND b.v=d.v
    AND AVG(sqrt((a.x)^2+(a.y)^2)+sqrt((b.x)^2+(b.y)^2))<c.v
    WITHIN 5ms
Q4: PATTERN SEQ(A a, NEG B b, C c)
    WHERE a.ID=b.ID AND a.ID=C.ID
    WITHIN 4ms
```

Listing 4.2: Queries for experiments with synthetic data.

Shedding strategies. We compare against several baseline shedding strategies:

1. *Random input shedding (RI)* discards input events randomly (*e.g.*, Apache Kafka [21]).
2. *Selectivity-based input shedding (SI)* discards input events by assessing the query selectivity per event type, which corresponds to semantic load shedding as developed for traditional data stream processing with Borealis [39].
3. *Random state shedding (RS)* discards partial matches randomly.
4. *Selectivity-based state shedding (SS)* discards partial matches based on the query selectivity for the incorporated events, which is inspired by techniques for approximate ESP [120].

For our approach, we test three instantiations of the shedding functions (see Section 4.2.5): *Input-based shedding (HyI)*, *state-based shedding (HyS)*, and *hybrid shedding (Hybrid)*.

We estimate the cost models for 4 time slices and 10 clusters (using K-Means) for each category of queries Q1, Q2, and Q4, while 4 clusters per state are derived for query Q3. The number of clusters was used as the maximum depth for the decision tree classifier learned for each category.

For the citibike scenario, we considered 3 time slices, while the number of clusters in K-Means and the maximal tree depth of the classifiers were set to 15. Turning to the cluster monitoring application, we also relied on 3 time slices. The number of clusters and the maximal tree depth were set to 30.

Measures. We measure the performance of query evaluation under a strict latency bound. In our experiments, this bound is typically defined as a percentage of the latency observed without load shedding. Enforcing the latency bound, we measure the effect of shedding on the result quality and the throughput of the ESP engine. Result quality is mostly assessed in terms of recall, *i.e.*, the ratio of complete matches obtained with shedding within the latency bound, and all

complete matches, derived without shedding. For monotonic queries, false positives will not occur, so that precision is not compromised. For the non-monotonic query Q4, we also measure precision, though. Throughput is measured in the number of processed events per second (events/s).

Implementation and environment. We developed a stand-alone ESP engine in C++ for automata-based execution model of pattern detection queries.[1] The offline estimation of the cost model was parallelised for different sets of partial matches. To reduce the overhead during online adaptation, we further derived lookup tables from the learned classifiers. Most experiments ran on a workstation with an i7-4790 CPU, 32GB RAM, with operating system Ubuntu 16.04. Cost model estimation took between 0.75 and 4.5 seconds on this machine, which we consider feasible for offline bootstrapping. The results of real-world case studies were obtained on a NUMA node with 4 Intel Xeon E7-4880 CPU (60 cores, 120 threads) and 1TB RAM, running the operating system openSUSE 15.0.

4.4.2 Overall Effectiveness and Efficiency

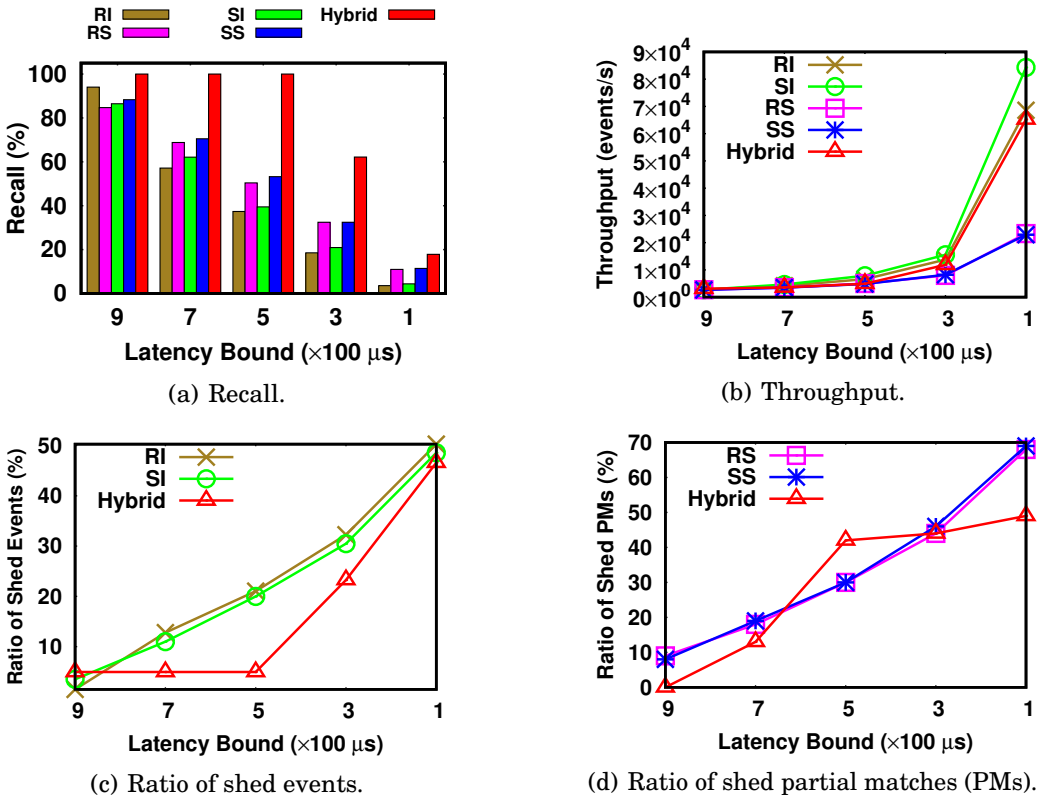
We test the general performance of hybrid load shedding with query Q1 over dataset DS1, drawing the values of attribute V for events of type C from a uniform distribution $\mathcal{U}(2,10)$. The attribute values of event A and B obey $\mathcal{U}(1,10)$. Hence, all events of types A and B may, in general, be part of complete matches. However, partial matches (consist of events A and B) whose payloads satisfy $a.V + b.V > 10$ will never lead to a complete match (recall the value predicate `a.V+b.V=c.V`) and can thus be discarded without compromising the recall.

We test the baseline approaches (RI, SI, RS, SS) against our hybrid strategy when varying the latency bound. Without load shedding, the average and 95$^{\text{th}}$ percentile latencies are 1,033μs and 1,673μs, respectively, so that we consider latency bounds ranging from 100μs to 900μs. The recall and throughput performance of the query processing under different latency bounds is illustrated in Figure 4.5 (average latency) and Figure 4.6 (95$^{\text{th}}$ percentile latency).

Figure 4.5(a) and Figure 4.6(a) show that hybrid load shedding yields the highest recall. With tighter latency bounds, the recall quickly degrades with the baseline strategies, whereas our hybrid approach keeps 100% recall for an average latency bound between 900μs and 500μs (Figure 4.5(a)). Turning to the stricter 95$^{\text{th}}$ percentile latency bound (Figure 4.6(a)), 100% recall is still achieved for the bound of 900μs. This highlights that our approach is able to assess the utility of partial matches and input events. Therefore, it is able to discard the least promising data. In contrast, random and selectivity-based strategies are unaware of such utilities and tend to discard data that induces a loss of many complete matches, leading to much lower recall value.

Specifically, in this setting, events of type A with $a.v_1 = 10$ and events of type B with $a.v_1 = 10$ have zero contribution to complete matches. Partial matches that consist of events instances of type A and B with $a.v_1 + b.v_1 > 10$ have zero contribution. The hybrid approach is able to

[1] The code and datasets are publicly available at https://github.com/zbjob/AthenaCEP



(a) Recall.

(b) Throughput.

(c) Ratio of shed events.

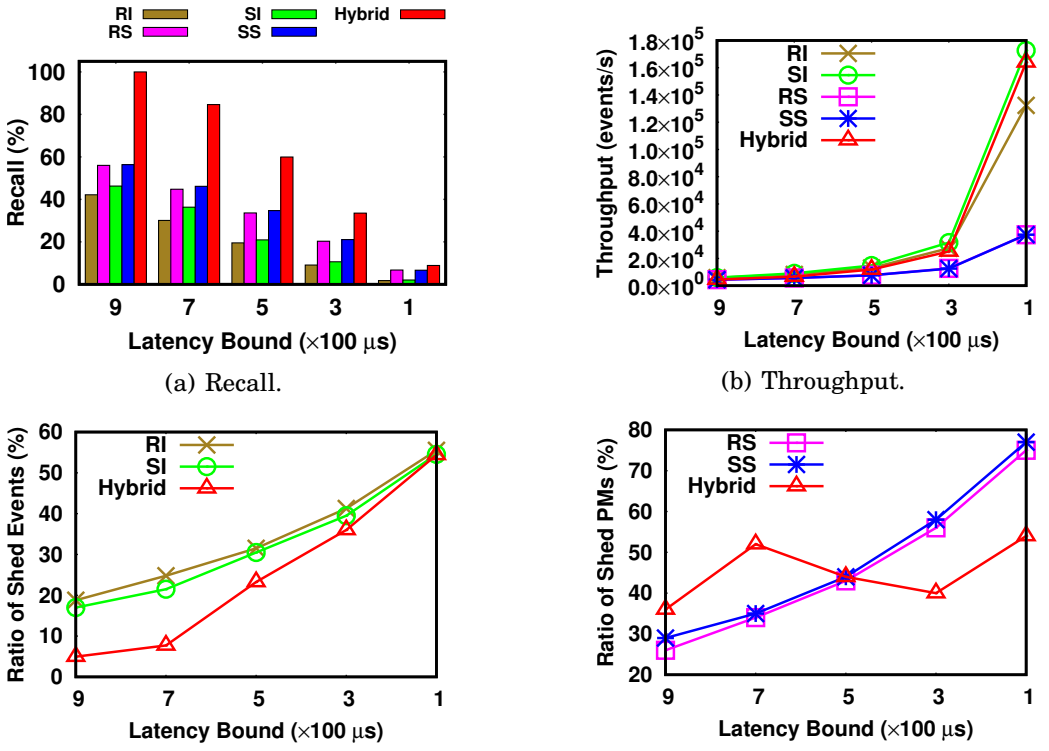
(d) Ratio of shed partial matches (PMs).

Figure 4.5: Experiments when varying the bound enforced for the average latency.

capture these differences and discard input events and partial matches with zero contribution. In contrast, RI and RS randomly drop input events and partial matches without considering their contributions. SI only distinguishes an order of selectivity at event type level: Events of type D have the lowest selectivity (zero) and type A and B have lower selectivity than type C. But it is unable to distinguish the importance of different event instances that are incorporated in partial matches – not considering the state. Similarly, SS only distinguishes a partial order between partial matches at the type-level, instead of partial match at an instance-level.

Moreover, state-based strategies yield better recall than their input-based counterparts, because they make shedding decisions based on cost models at a more fine-granular level. However, input-based techniques yield higher throughput, see Figure 4.5(b) and Figure 4.6(b). The reason is that RI and SI immediately discard the input events, while shedding with RS and SS still consumes computational resources to construct the partial matches that will be shed eventually. Our hybrid approach also turns out to be efficient, showing nearly the same throughput performance as the input-based strategies. This is remarkable, given the aforementioned high recall results.

The reason for the above result becomes clear when exploring the ratios of shed events and partial matches among different approaches. According to Figure 4.5(c), our hybrid strategy discards fewer input events compared to RI and SI. Up to the average bound of 500μs, our



(a) Recall.

(b) Throughput.

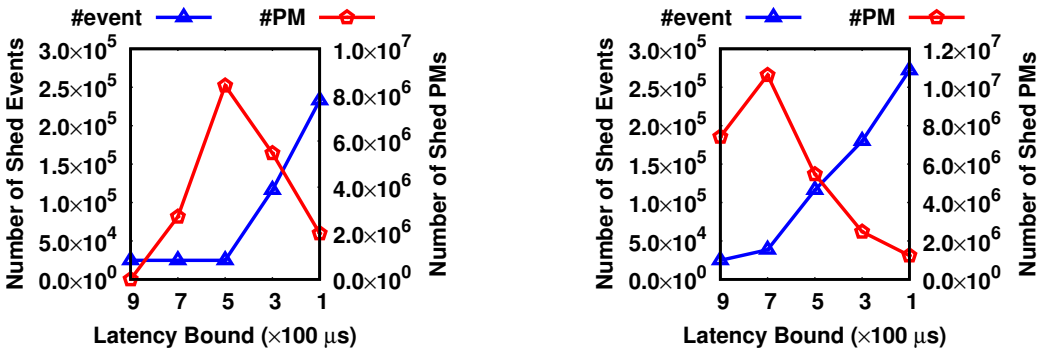
(c) Ratio of shed events.

(d) Ratio of shed partial matches (PMs).

Figure 4.6: Experiments when varying the bound enforced for the 95$^{\text{th}}$ percentile latency.

strategy discards a steady ratio of input events, but an increasing number of partial matches (Figure 4.5(d)). As such, the required reduction of latency is achieved by an increasing ratio of shed partial matches, which does not compromise the recall value, see Figure 4.5(a). Once more input events need to be shed to satisfy the latency bound, however, the ratio of discarded partial matches flattens, see Figure 4.5(d) for bounds from 500μs to 300μs. Shedding more input events also decreases the number of generated partial matches, thereby reducing the pressure to shed partial matches. With tighter latency bounds, the shedding ratio of partial matches increases again, since the bound cannot be met by mainly shedding input events. The above effects are mirrored in the results obtained for the 95$^{\text{th}}$ percentile latency in Figure 4.6(c) and Figure 4.6(d).

The above results illustrate that state-based shedding, in general, leads to higher recall. However, throughput is increased more rapidly through input-based shedding, since it completely avoids to spend computational effort on the creation of (potentially irrelevant) partial matches. Hybrid load shedding strives for both, high recall and high throughput, by balancing input-based and state-based shedding. Figure 4.7(a) shows a turning point at the aforementioned latency bound of 500μs, at which the number of shed partial matches decreases and the number of shed input events increases. This behaviour is explained as follows. For tighter bounds, the filter function for input-based shedding (Section 4.2.5) derived from the shedding set (Section 4.2.4)



(a) Average latency bound.

(b) 95th percentile latency bound.

Figure 4.7: Details on workings of hybrid load shedding.

contains more heterogeneous partial matches (*i.e.*, from a larger number of classes and time slices), which increases the selectivity of the filter function, while filtering is also applied for longer intervals (until the latency drops below the threshold). Since more input events are filtered, fewer partial matches are created in the first place, so that the absolute number of shed partial matches also decreases. The result is mirrored for the 95th percentile latency in Figure 4.7(b), with a turning point at the bound of 700μs

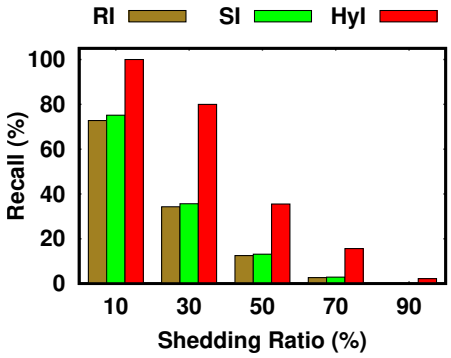
4.4.3 Sensitivity Analysis

We investigate how different query and data characteristics and settings of parameters affect the effectiveness and efficiency of load shedding approaches, as listed below.

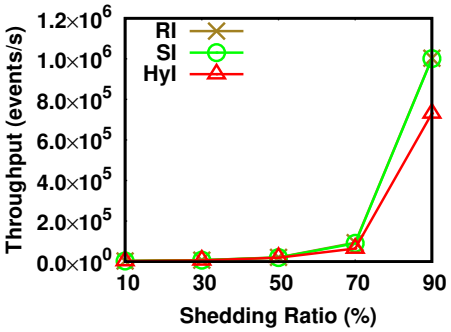
- How good is the selection of data to shed (Section 4.4.3.1)?
- How sensitive is the approach to query properties, such as its selectivity, time-window size, and pattern length (Section 4.4.3.2)?
- What is the impact of cost model properties (Section 4.4.3.3)?
- What is the benefit of indexing and early curbing (Section 4.4.3.4)?
- Does the model adapt to changes in the stream (Section 4.4.3.5)?

4.4.3.1 Selection of Data to Shed

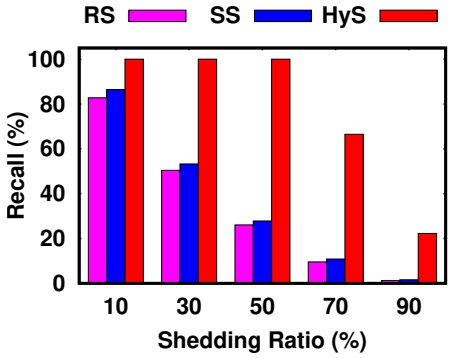
Using the same dataset DS1 and query Q1, we explore how well the different load shedding strategies select input events or partial matches that do not incur a loss in the recall. To this end, we fix the ratio of shed events and partial matches. Figure 4.8(a) and Figure 4.8(b) illustrate that input-based shedding using our cost model (HyI) yields significantly better recall with slightly worse throughput compared to random-based (RI) and selectivity-based (SI) input shedding. We conclude that our cost model enables a precise assessment of the utility of partial matches based on which we characterise the input events to shed.



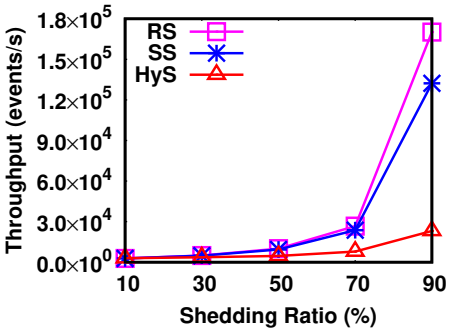
(a) Recall (input-based).



(b) Throughput (input-based).



(c) Recall (state-based).



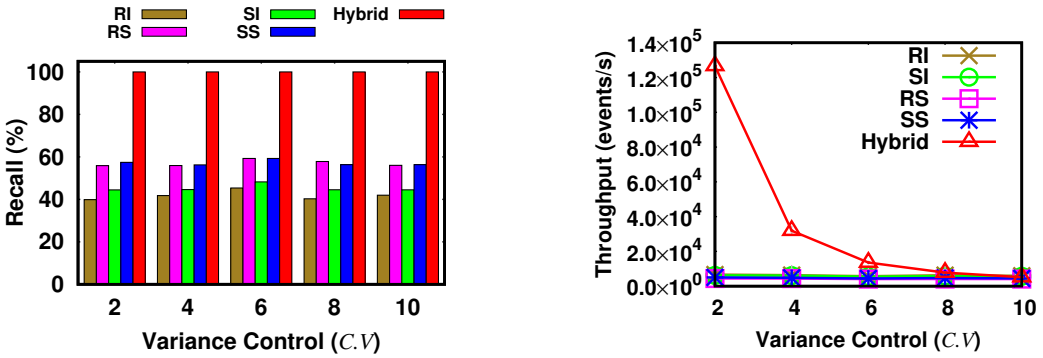
(d) Throughput (state-based).

Figure 4.8: Evaluation of the effectiveness of the selection of data to shed.

Figure 4.8(c) shows the recall for state-based strategies. Our approach (HyS) shows better recall than random (RS) or selectivity-based strategies (SS). The margin becomes larger with more data being shed. When discarding 50% of the partial matches, our approach keeps 100% recall, whereas the baseline strategies drop to 30%. Interestingly, at that point, all approaches show similar throughput, see Figure 4.8(d). With high shedding ratios, our approach is less efficient than the baselines. Yet, the comparatively higher throughput is of little practical value, given the very low recall.

4.4.3.2 Sensitivity to Query Properties

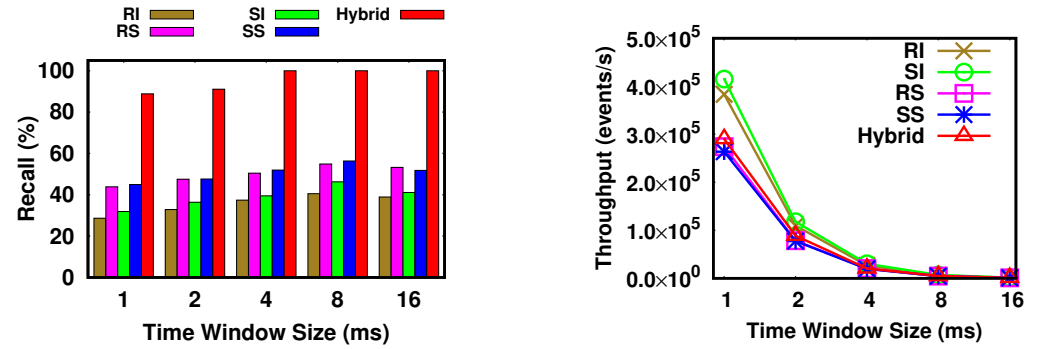
Variance of query selectivity. We test the impact of the variance of query selectivity with query Q1 over dataset DS1. Specifically, we change the distribution of attribute V for type C events in the range $[2,x]$ with $x \in [2,11]$. This way, we control the overlap of the distributions of attribute V for type A and B events that lead to complete matches. Setting a bound for the 95^{th} percentile latency as 800μs (50% non-shedding latency), Figure 4.9(a) shows that, as expected, the recall is not affected. Also, our hybrid load shedding always achieves the highest recall.



(a) Recall. (b) Throughput.

Figure 4.9: Impact of variance of query selectivity.

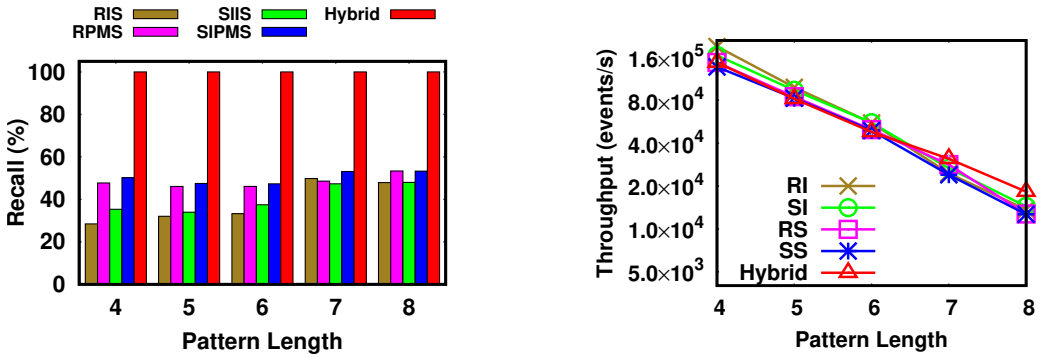
Figure 4.9(b), in turn, shows a major impact on throughput. If selectivity shows low variance ($x = 2$), our hybrid approach is able to precisely assess the utility of input events and discard irrelevant ones. Hence, the throughput is 120× higher than the baseline approaches. If the variance of the selectivity is high, the utility of input events cannot be assessed precisely, so that shedding happens at the more fine-granular level of partial matches. This leads to the throughput becoming similar to one of the baseline strategies.



(a) Recall. (b) Throughput.

Figure 4.10: Impact of time window size.

Time window size. Under a steady input rate, the size of a query time window affects the growth of partial matches. Because it defines the life span of partial matches. A large time window leads to more partial matches which grow rapidly. We evaluate this effect by varying the time window size of query Q1 over dataset DS1 from 1ms to 16ms, with an 800μs bound on the 95$^{\text{th}}$ percentile latency (50% non-shedding latency). The input rate is 1 million events per second. Figure 4.10(a) shows that with increasing window size, recall improves for all approaches, while our strategy yields the best results. According to Figure 4.10(b), input-based baseline strategies achieve the best throughput. Our hybrid approach has comparable performance to the state-based



(a) Recall.

(b) Throughput.

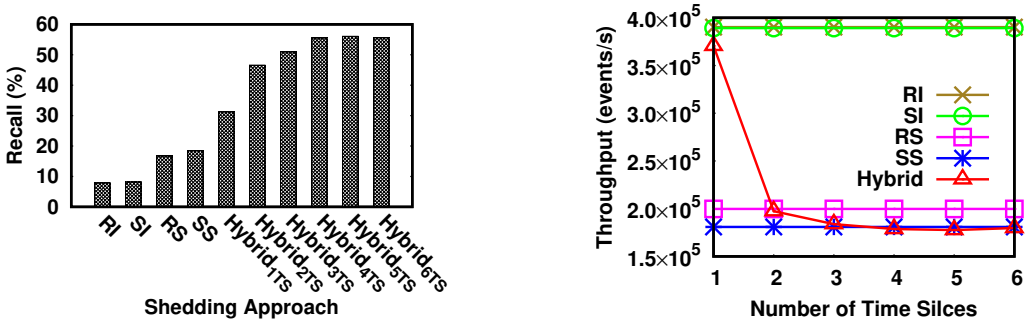
Figure 4.11: Impact of queried pattern length.

strategies. With an increasing time window size, the differences become marginal due to the exponentially growing number of partial matches and their increased lifespans.

Pattern length. We consider the length of the sequential pattern to detect and the number of correlation predicates. Using query Q2 over dataset DS1 and a bound for the 95$^{\text{th}}$ percentile latency (800μs), we vary the limit of the Kleene closure operator to obtain patterns of length from four to eight. As shown in Figure 4.11(a) and Figure 4.11(b), the recall remains stable with increasing pattern length, whereas throughput decreases drastically. Because longer patterns and more correlation predicates increase the computational overhead and thus reduce throughput performance. Interestingly, our hybrid approach shows a less severe reduction than the other strategies, achieving the highest throughput for pattern lengths above six. The reason is that for more complex quires, consumption of partial matches grows rapidly and reduce processing throughput. However, this is exactly what our cost model is able to exploit. Compared to other baselines, it can target the most computationally expensive partial matches with small contributions. Hence, more complex queries may particularly benefit from our approach.

4.4.3.3 Impact of Cost Model Properties

Temporal granularity. To assess the effect of temporal abstractions in our cost model (Section 4.3.1), we use query Q1 (time window 2ms) over dataset DS1 with a bound 350μs on the 95$^{\text{th}}$ percentile latency (20% non-shedding latency), varying the number of time slices. Figure 4.12(a) depicts the observed recall, where our hybrid approach is annotated with the number of time slices (*e.g.*, 3TS means three time slices). While our approach outperforms all baseline strategies in the recall, we see evidence for the benefit of using time slices as the maximum recall is obtained with four or more slices. Increasing the number of time slices leads to decreased throughput, see Figure 4.12(b), due to the implied overhead. The throughput observed for hybrid shedding is between the input-based (RI and SI) and state-based strategies (RS and SS). Yet,



(a) Recall. (b) Throughput.

Figure 4.12: Impact of temporal granularity.

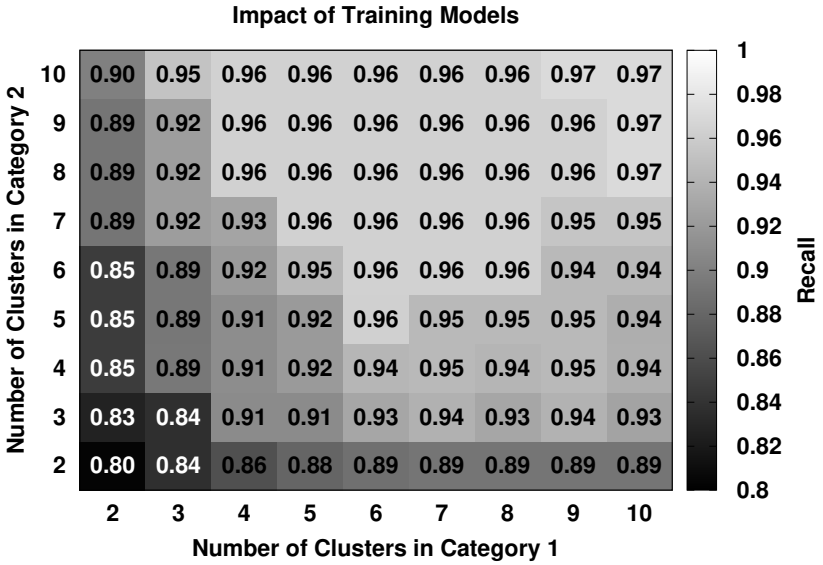
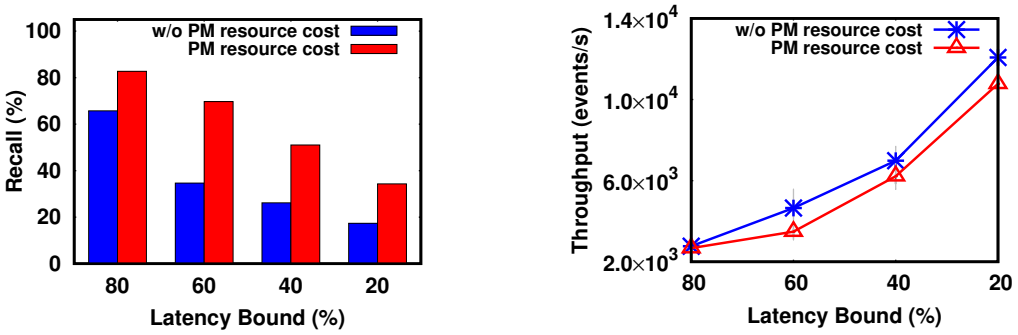


Figure 4.13: Cost model estimation.

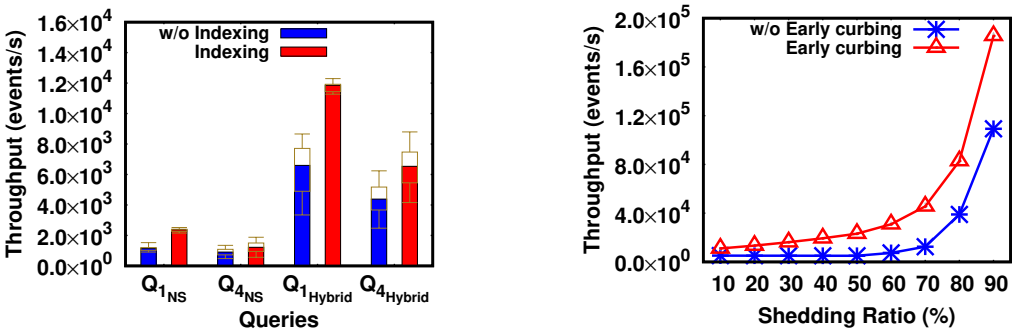
when throughput is comparable to RI and SI, the recall of our hybrid approach is 3.8× higher. Similar trends are observed with respect to the state-based baseline load shedding strategies.

Data granularity. We assess the impact of data abstraction of the cost model (Section 4.3.1) by evaluating query Q1 over dataset DS1. Q1 has two intermediate categories of state (partial matches that consist of type A events and partial matches that consist of type A,B events). We vary the number of clusters from 2 to 10 for each state for clustering (K-means) and set the maximum depth of decision tree classifiers as 10. Under a 500μs average latency bound, we measure the recall as illustrated in Figure 4.13. Overall, the observed recall is not very sensitive to the number of clusters. More clusters lead to higher recall scores, but only until reaching a certain number (*e.g.*, 8), after which the gain becomes marginal.



(a) Recall. (b) Throughput.

Figure 4.14: Impact of resource costs of partial matches.



(a) Indexing. (b) Early curbing.

Figure 4.15: Impact of indexing and early curbing.

Resource costs of partial matches. For some queries, the consumption of resources may differ greatly among partial matches. We explore this aspect with query Q3 over dataset DS2. The query computes the average Euclidean distance to pairs of numeric values of type A and B events, checking whether the result is larger than a numeric attribute of type C event. We established empirically that handling partial matches comprising an A and a B event requires 5× more runtime than handling matches of a single A event. We compare hybrid shedding with and without incorporating an explicit resource cost for the consumption of partial matches (Section 5.3). Applying a bound on the average latency (certain ratio of the non-shedding latency), Figure 4.14(a) shows that our comprehensive cost model leads to higher recall. This improvement comes at a small expense, as there is only a minor reduction in throughput, as seen in Figure 4.14(b).

4.4.3.4 Impact of Indexing and Early Curbing

Indexing. Turning to efficient management of matches (Section 4.3.4), we evaluate two queries, Q1 and Q4, that differ in pattern lengths and the variance of selectivity, over dataset DS1. We apply our hybrid strategy and an 800μs bound on the 95$^{\text{th}}$ percentile latency. We measure the throughput performance of the baseline pattern detection without any shedding (NS) and hybrid

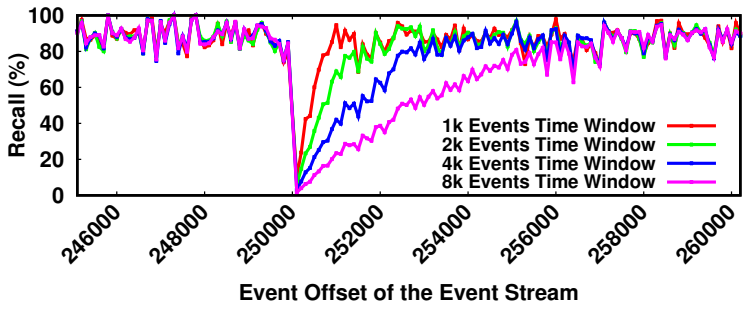


Figure 4.16: Adaptivity of the cost model.

shedding approach (Hybrid) , as well as with and without indexing. Figure 4.15(a) illustrates the importance of indexing partial matches which increases throughput by up to 2×.

Early curbing. We further evaluate hybrid shedding using query Q4 over dataset DS1, with and without early curbing, under different shedding ratios. Figure 4.15(b) shows that early curbing leads to a significant increase of throughput. This is expected since early curbing avoids the computational overhead of creating partial matches that are discarded again at later time.

4.4.3.5 Adaptivity to Changes in the Stream

We consider the adaptivity of our cost model, when the value distribution of the input events' payload data changes. For dataset DS1, we change the value distribution of attribute V for C events at a fixed point (the 250k$^{\text{th}}$ event in the stream) from $\mathscr{U}(2,10)$ to $\mathscr{U}(12,20)$. This is the worst-case scenario where the costs of partial matches totally flip: The previous shedding sets before the change point become the most promising matches and events to retain. Setting a bound on the average latency (400μs), we run query Q1 with four different sizes of time windows (1k, 2k, 4k, and 8k events). Figure 4.16 shows how our approach (Section 4.3.2) adapts the estimates for the contribution and consumption of partial matches: At the change point, the recall drops to zero as outdated cost model estimates lead to shedding of all relevant partial matches. However, the change is quickly detected and incorporated, so that the recall converges to the previous level. Convergence is quicker for smaller window sizes, because partial matches obtained by outdated shedding decisions have a shorter lifespan and therefore, higher updating rate for the cost model.

4.4.3.6 Non-Monotonic Queries

To test the impact of query monotonicity, we evaluate query Q4 over dataset DS1. As discussed in Section 5.2.1, shedding may produce false positives for non-monotonic queries' results, so that we measure both precision and recall for these queries. To this end, we control the extent of non-monotonicity by varying the occurrence probability of the negated event type B from 5% to 50%. The other types are evenly distributed. Figure 4.17 shows the results when shedding 10% of partial matches. The recall is stable, as our approach discards only the least important partial

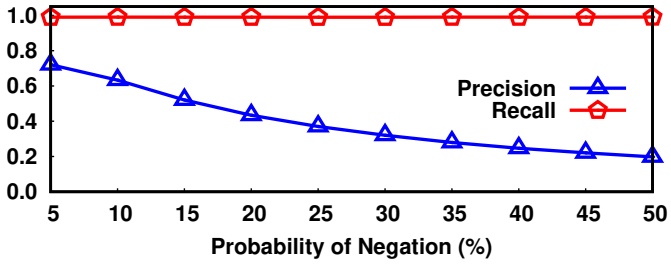


Figure 4.17: Impact of monotonicity violation.

matches and all complete matches are detected (recall value is 1.0). Yet, precision decreases when increasing the probability of the negation (type B events), *i.e.*, the number of false positives becomes larger. Whether this effect is acceptable, depends on the selectivity of the sub-query that violate the monotonicity property and the specific requirements of certain applications.

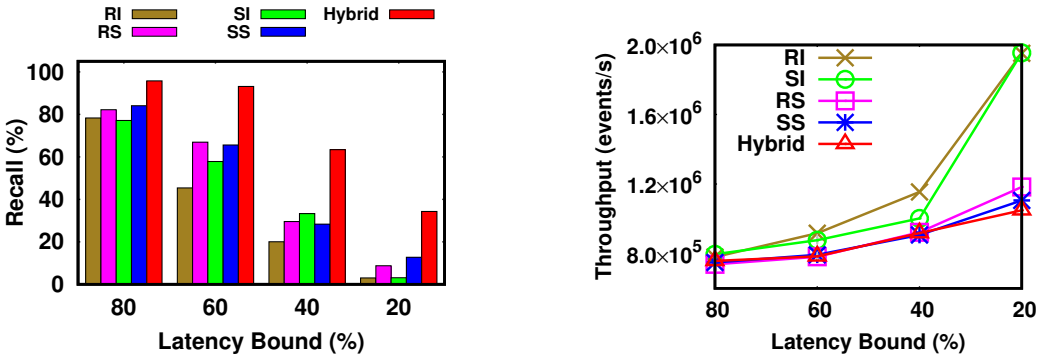
4.4.4 Case Studies

We test the proposed hybrid load shedding techniques on two real-world scenarios: *citibike* [54], a bike sharing provider in New York City and Google cluster-usage traces [160].

4.4.4.1 Bike Sharing

To assess real-world feasibility, we use the aforementioned dataset of *citibike* [54], introduced already in Example 4.1. Here, the event stream comprises information about trips by individual users of the service. For the trip data of October 2018, we test the query given in Listing 4.1 that checks for 'hot paths'. We configure the query such that a path has to contain at least five stations, *i.e.*, five is the minimal length of the Kleene closure operator in the query. We test our hybrid strategy against the baseline approaches with four latency bounds, 80%, 60%, 40% and 20% of 99^{th} percentile latency in non-shedding cases. Here, the selectivity-based approaches (SI, SS) exploit the user type (annual member users, 3-day pass users, and 24-hour pass users).

Our hybrid approach consistently yields the best recall, see Figure 4.18(a), with the margin becoming larger for tighter latency bounds. At a 20% bound, the recall of our approach reaches 11.4×, 11×, 3.9×, 2.7× the recall of RI, SI, RS, SS, respectively. Figure 4.18(b) shows that the throughput of our hybrid approach is comparable to the state-based strategies (RS and SS), but lower than the input-based strategies (RI and SI). We found the reason to be that, for this dataset, our approach tends to shed more partial matches than input events. While this leads to high recall, shedding is less efficient.



(a) Recall comparison.

(b) Throughput comparison.

Figure 4.18: Case study: Bike sharing.

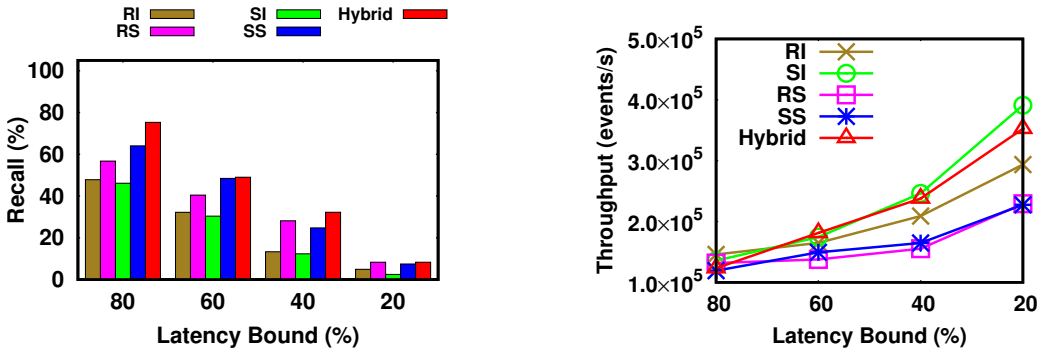
4.4.4.2 Cluster Monitoring

As a second real-world dataset, we use the Google Cluster-Usage Traces [160] that have been obtained from a 12.5k-machine cluster over about a month-long period in May 2011. The dataset contains events that indicate the lifecycle (e.g., submit, schedule, evict, and fail) of tasks running in the cluster. We evaluate the query given in Listing 4.3, which detects the following pattern: A task is submitted, scheduled, and evicted on one machine; later it is rescheduled and evicted again on another machine; and finally it is rescheduled on a third machine, but fails execution eventually; all within one hour.

```
PATTERN SEQ(Submit a, Schedule b, Evict c, Schedule d, Evict e, Schedule f,
    Fail g)
WHERE [task_id] AND b.machine=c.machine
AND b.machine!=d.machine AND d.machine=e.machine
AND d.machine!=f.machine AND f.machine=g.machine
WITHIN 1h
```

Listing 4.3: Query for Google cluster dataset.

Again, we tested our hybrid approach and the RI, SI, RS, and SS baseline strategies with different latency bounds on the average latency. Figure 4.19(a) illustrates that hybrid shedding yields the best recall, up to 4× better than with input-based shedding (RI, SI) and 1.5× better than with state-based shedding (RS, SS). Figure 4.19(b) shows the observed throughput, hinting at the general trade-off of input-based and state-based shedding. The former tend to achieve higher throughput at the expense of lower recall. Our hybrid approach achieves similar throughput to the best performing baseline strategy (SI), being slightly slower only for the 20% latency bound. However, hybrid shedding achieves much higher recall, thereby confirming the observations obtained in the controlled experiments.



(a) Recall comparison.

(b) Throughput comparison.

Figure 4.19: Case study: Cluster monitoring.

4.5 Summary

In this chapter, we proposed hybrid load shedding for pattern detection queries over event streams. It enables best-effort query evaluation, striving for maximal result quality while staying below a latency bound. Since the utility of an event in a stream may be highly dynamic, we complemented traditional input-based shedding with a novel perspective of state-based approach—shedding partial matches. We presented a cost model to balance various shedding strategies and decide on what and how much data to shed. To realise these ideas, we built a prototype system and evaluated it on both synthetic and real-world datasets. Our experiments highlight the effectiveness and efficiency of our proposed hybrid shedding approach.

CHAPTER 5

EFFICIENT REMOTE DATA INTEGRATION

This chapter addresses the problem of efficient integration of remote data in event stream processing. Specifically, it aims to reduce the remote data fetching latency, $\ell_{fetch}(c)$. Many ESP applications require the combination of the events' payload with data from remote sources to determine whether a partial match shall be processed further. However, such dependencies are problematic, since waiting for remote data to be fetched interrupts the processing of the stream. Yet, without event selection based on remote data, the query state to maintain may grow exponentially. In either case, the performance of the ESP engine degrades drastically. To tackle these issues, this chapter presents EIRES, a framework for efficient integration of data from remote sources in the context of event stream processing. It employs a cache to decouple fetching of remote data and its use in query evaluation. We present a cost-model to determine when to fetch certain remote data elements and how long to keep them in a cache for future use. EIRES combines strategies for (i) prefetching technique that queries remote data based on the anticipated use and (ii) lazy evaluation that postpones the event selection based on remote data without interrupting the stream processing.

This chapter is organised as follows: We first illustrate the problem of integrating remote data in event stream processing with examples in Section 5.1. In Section 5.2, we formally define the remote data integration problem and propose the EIRES framework that consists of three core components, utility modelling (Section 5.3), remote data fetching (Section 5.4) and cache management (Section 5.5). The utility modelling component provides a unified cost model to guide remote data prefetching, lazy evaluation, and cache management. Comprehensive evaluation results are detailed in Section 5.6 to illustrate the effectiveness and efficiency of EIRES. Section 5.7 closes this chapter with a summary.

5.1 Problem Illustration

Many ESP applications require the combination of the events' payload with data from remote sources to determine whether a partial match shall be processed further. For instance, in financial fraud detection [112], a decision about suspicious transactions relies on contextual information on a user's spending history, the transactional volume at a specific location, or the behaviour of other users. Similarly, the control of logistic processes is often based on patterns of events sensed from RFID- or barcode-tagged packages. Decision-making is then based on a combination of these events with static data [204], *e.g.*, on a package's content or customs clearance, that is queried from remote databases as part of the EPCglobal infrastructure [76]. Moreover, virtually all ESP engines, from enterprise offerings such as Oracle CEP [140] to open-source systems such as Esper [69] enable the integration of static data in the specification of pattern detection queries, through SQL interfaces or user-defined functions. Below, we review two exemplary use cases to illustrate the need for efficient integration of remote data in event stream processing.

Fraud detection. The detection of fraud in credit card usage is a common use case for event pattern detection. Here, events denote financial transactions or indicate status changes, such as a transaction denied by one of the involved operators or a change in the spending limit. Based on these events, pattern detection queries identify patterns of suspicious usage and potentially block the respective transactions. Yet, many of these patterns involve contextual information, *e.g.*, related to a user's spending history or the behaviour of related accounts. Recall that the previous example query in Listing 2.1 illustrates this setting. Its semantics has been explained in Section 2.2. In a nutshell, it detects suspicious patterns that query remote data including historical frequent locations (`REMOTE[t1.user]`), maximum spending limit enforced by costumers' organisations (`REMOTE[t3.user]`) and pre-authorised clients (`REMOTE[t3.org]`). Note that the same remote data queried for a partial match can also be useful when evaluating other partial matches, *e.g.*, the organisation-wide spending limits and pre-authorised clients that apply to all corporate credit cards of a single organisation (business unit). The query in Listing 2.1 is evaluated under tight latency bounds. Clearance of a credit card transaction needs to happen within 25ms [71], which includes data ingestion and forwarding of the clearance decision. Yet, ignoring event selection based on remote data leads to non-determinism (*e.g.*, both possible outcomes of `l.limit > REMOTE[t1.org]` are considered) and false positives and false negatives may lead to significant financial losses [71].

Logistics processes. Tracking packages in logistics processes based on RFID or barcode tags is an integral part of applications for the supply chain management. It enables the detection of events that indicate progress, milestones, or exceptional situations of business processes. Based thereon, service level agreements (SLAs) are monitored [25] and fine-granular operational decision making is facilitated [204]. As an example, consider a conveyor belt system at a warehouse intake, which may route packages based on their content or customs clearance. To make the

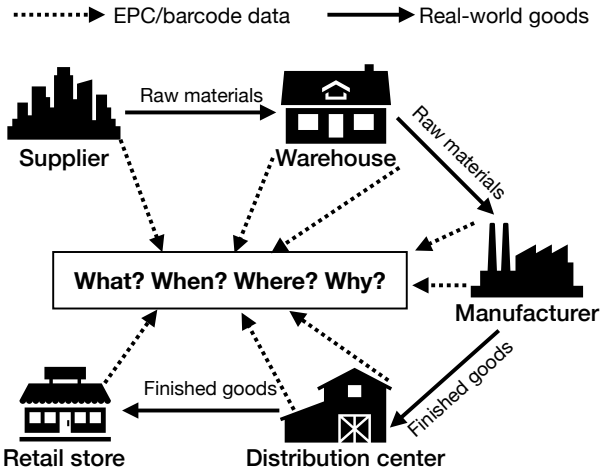


Figure 5.1: Illustration of EPICIS in supply chain management.

routing decision, a pattern of events emitted by tag readers is detected. This involves a lookup of a package's content and clearance information at a remote database. In supply chain management, such a lookup is facilitated by the EPCglobal infrastructure as depicted in Figure 5.1, which provides the so-called EPC Information Services (EPCIS) to query for an object identifier along the supply chain [162]. The evaluation of such queries with low latency is crucial to enable effective routing and warehouse management. However, since EPCIS are shared among globally distributed enterprises, fetching remote data incurs a significant overhead in event stream processing that aims at microsecond-latency [204]. We further observe that bundling and unbundling operations, which are common in logistics processes, impact the access to remote data in event stream processing. That is, the detected event patterns may refer to individual packages, while some of the information fetched from remote databases is actually aggregated per container or complete shipment (*e.g.*, customs details).

For applications like the above examples, a naive integration of remote data reduces the performance of an ESP engine drastically. Fetching data once it is needed to proceed with query evaluation, see Figure 5.2 (top), interrupts the processing of the stream and temporarily increases the latency by orders of magnitude. Even a small latency of dozens of milliseconds to look up remote data is problematic, given that contemporary ESP engines achieve microsecond latency and many applications enforce tight latency bounds. While the assessment of remote data is mandatory to maintain result correctness, only incorporating it in a post-processing step, while ignoring it as part of event selection during query evaluation, is also not a viable option: The resulting non-determinism in query evaluation would add further to the exponential growth of the number of partial matches, thereby increasing the overall query processing latency.

To address the above issues, this chapter presents the framework for efficient integration of remote data in event stream processing (EIRES). As illustrated in Figure 5.2, our idea is to decouple (i) fetching of remote data from (ii) its use in query evaluation. While under a

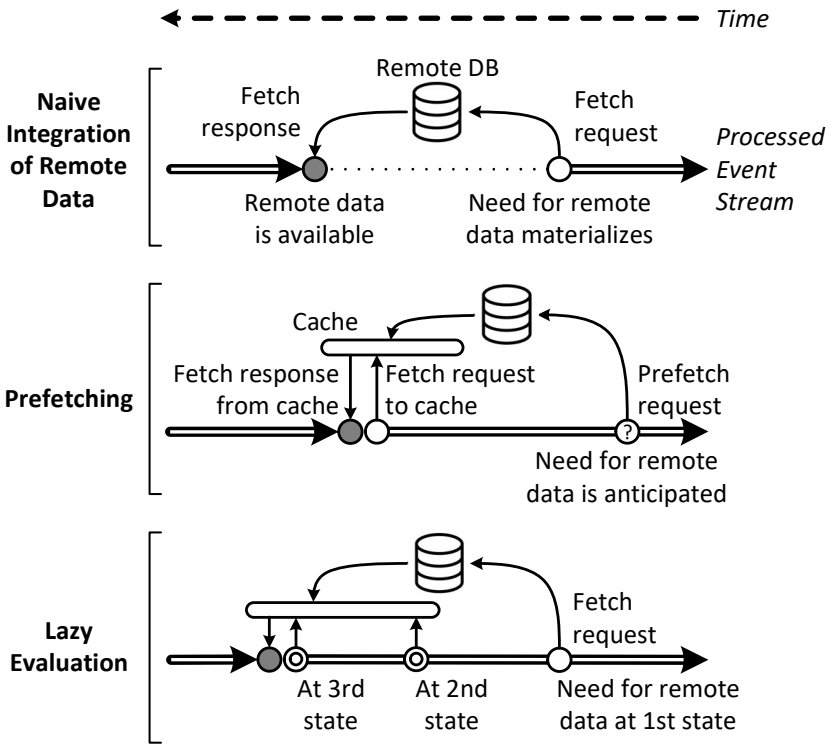


Figure 5.2: Strategies to integrate remote data in event stream processing: Naive integration; prefetching based on anticipated use; lazy evaluation once data is available.

naive model, data is fetched once it is needed and then used immediately, EIRES employs a local cache to decouple both operations. Fetching may happen *before* the need for remote data materialises (*i.e.*, *prefetching* in Figure 5.2 (middle)) and the evaluation of partial matches based on it may be postponed until *after* the remote data is available (*i.e.*, *lazy evaluation* in Figure 5.2 (bottom)). While both strategies hide the data transmission latency, they also come with side effects. Prefetching may fill the cache with superfluous data because of inaccurate prediction of remote data usage, while lazy evaluation may suffer from the growth of the number of partial matches. To mitigate these issues, we carefully balance the strategies' application based on their expected benefits and costs in a given situation.

5.2 Foundations of Remote Data Integration

This section first refines the formal data model for remote data integration in ESP (Section 5.2.1), before introducing the latency minimisation problem for this setting (Section 5.2.2), which is addressed in the remainder.

5.2.1 A Closer Look at Remote Data

Recall that the remote data is formally defined in Section 2.1.2. Here, we focus on *what* data elements are fetched from remote data sources, rather than *how* they are retrieved. That is, we abstract from the specific look-up queries executed at remote sources and use data elements as the basis for our model. A data element may be thought of as a key-value pair or a relational tuple. Formally, we use a set $\mathscr{D} = \{d_1, \ldots, d_n\}$ to capture the remote data elements. Moreover, we write $|d| \in \mathbb{N}$ for the size of a data element. Since fetching data elements from different sources can result in different transmission latencies. Fetching data from a cloud infrastructure may incur higher network delays compared to fetching from local clusters. Here, we assume that latency is monitored per data element, and denoted as $\ell_{remote}(d)$.

Notably, data models are hierarchical in many applications (*e.g.*, logistic containers or organisation-hold credit card accounts), which means that there exists a containment relation between remote data elements. Therefore, we also consider a function $\rho : \mathscr{D} \nrightarrow \mathscr{D}$ that maps an element to another one if the former is contained in the latter. The size of the containing element is then given as the sum of the contained elements, *i.e.*, $|d| = \sum_{d' \in \mathscr{D}, \rho(d')=d} |d'|$. In the above fraud detection example, remote data would capture the known locations of credit card usage per client, the limits of card accounts, and the set of pre-authorised clients for a specific organisation. The set of pre-authorised clients may also be organised hierarchically, *i.e.*, it can be fetched per credit card, per user, or for the organisation as a whole.

5.2.2 Problem Statement

Recall that query evaluation incurs latency — the time between the arrival of the last event of a match at the ESP engine and the actual detection of the match (see Section 2.3). Based thereon, we formulate the problem of efficiently evaluating a pattern detection query with remote data as the minimisation of the respective latency. This chapter realises the minimal latency by reducing the remote data fetching latency.

Problem 5.1. *Let Q be a pattern detection query and $S(..k)$ be a stream prefix. The problem of efficient remote data integration is to compute all matches $R = \langle C(1), \ldots, C(k) \rangle$ defined by Q, while minimising the remote data fetching latency $\ell_{fetch}(k)$, with $k \to \infty$.*

5.2.3 The EIRES Framework

To address Problem 5.1, we propose the EIRES framework. We summarise its intuition in Section 5.2.3.1 and introduce its components in Section 5.2.3.2.

5.2.3.1 Framework Intuition

We first reflect on the factors that constitute the latency $\ell(c)$ of a match c, and hence, the aggregated latency $\ell(k)$ (see Section 2.3). The overall processing latency $\ell(c)$ consists of pattern evaluation latency, $\ell_{match}(c)$, and the remote data fetching latency, $\ell_{fetch}(c)$. Many optimisations for event pattern detection , such as those based on state sharing [159, 196], semantic rewriting [67, 187], or load shedding [94, 202], will ultimately reduce the inherent latency of query evaluation, *i.e.*, $\ell_{match}(c)$.

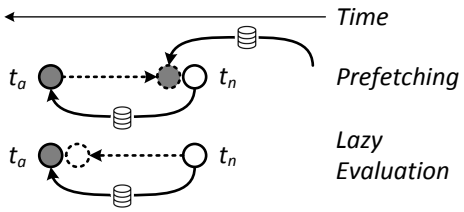


Figure 5.3: Intuition of the proposed strategies.

By contrast, this chapter focuses on the latency induced by fetching remote data, $\ell_{fetch}(c)$. Acknowledging that this latency corresponds to the time interval between the need (t_n) and availability of remote data (t_a), Figure 5.3 illustrates that it may be reduced from either end. That is, $\ell_{fetch}(c) = t_a - t_n$ (see Definition 2.10): The closer t_a and t_n are, the smaller $\ell_{fetch}(c)$ is. Here, we assume that the latency of the actual transmission of a data element d, *i.e.*, $\ell_{remote}(d)$, as induced by the requests and responses sent over the network, is monitored. This property, for example, applies to the use cases outlined in Section 5.1, where remote data resides at databases that external parties operate. As such, without any optimisation, $t_a - t_n = \ell_{remote}(d)$ holds. However, the latency $\ell_{remote}(d)$ may solely be *hidden* by either moving t_a closer to t_n, *i.e.*, by fetching the data earlier than it is actually needed, or by moving t_n closer to t_a, *i.e.*, by postponing the evaluation of the predicates that are based on the remote data. The more $\ell_{remote}(d)$ is hidden, the more $\ell_{fetch}(c)$ is reduced. We realise both ideas with a strategy for prefetching, coined **PFetch**, and a strategy for lazy evaluation, referred to as **LzEval**.

PFetch fetches remote data before it is actually needed, thereby hiding the data transmission latency. Data is then kept in a local cache at the ESP engine, from which it may be retrieved with negligible latency (in comparison to remote data transmission latency). Specifically, PFetch needs to realise the following operations:

(P1) Decide *when* prefetching a data element may be beneficial.
(P2) At a specific time, select *which* data elements to prefetch.
(P3) Prefetch the set of selected data elements.

Here, operation (P3) is trivial and performs data transmissions. In contrast, operations (P1) and (P2) induce optimisation problems. Determining the most beneficial time to prefetch a single

element, *i.e.*, (P1), involves conflicting goals. Fetching an element later has a positive effect on the prediction accuracy (the later, the more accurate) and efficient cache usage (the later, the better the cache utilisation). Yet, fetching should not occur too late, since the fraction of the transmission latency that is hidden decreases after a certain time point (the later, the smaller this fraction). As such, it is desirable to prefetch as late as possible while ensuring that the data still arrives on time, *i.e.*, just before it is needed.

Similarly, the selection of elements to actually prefetch, (P2), requires optimisation across a set of data elements. Intuitively, the elements that have the largest positive impact on the performance of query evaluation shall be selected. Yet, this impact depends on the currently maintained partial matches as well as on future partial matches to be generated while the data is fetched and kept in the cache.

LzEval postpones the evaluation of predicates based on remote data until the data is available in the cache. While fetching is triggered once the data is needed, pattern detection query evaluation of other non-remote data predicates continues in parallel to this fetching operation without interruption. The respective predicates are evaluated only at later stages, so that fetching of remote data is no longer a blocking operation. LzEval includes the following operations:

(L1) Initiate the fetching of remote data if it is needed during query evaluation and not available in the cache.
(L2) Decide on the partial matches for which lazy evaluation is applied, *i.e.*, for which the evaluation of predicates is postponed.
(L3) Adapt the evaluation procedure for the partial matches with lazy evaluation. Verify the predicates based on remote data once the remote data is available in the cache.

Here, operation (L1) is trivial, whereas operations (L2) and (L3) jointly induce a trade-off. Postponing the evaluation of predicates leads to additional partial matches being created, because event selection becomes less strict (relaxing remote-data-related predicates). As mentioned previously, the set of partial matches may grow exponentially in the number of processed events and the inherent evaluation latency $\ell_{match}(c)$ directly depends on this number. Consequently, postponing the evaluation of the respective predicates may help to hide the transmission latency of remote data, $\ell_{remote}(d)$, while at the same time increasing the inherent evaluation latency $\ell_{match}(c)$. Therefore, lazy evaluation of predicates shall be applied only when its benefits outweigh the increase in the inherent evaluation latency.

Both PFetch and LzEval strategies have in common that they require **cache management**, which corresponds to the following operation:

(C1) Maintain a set of data elements in the cache based on its current content and newly fetched elements.

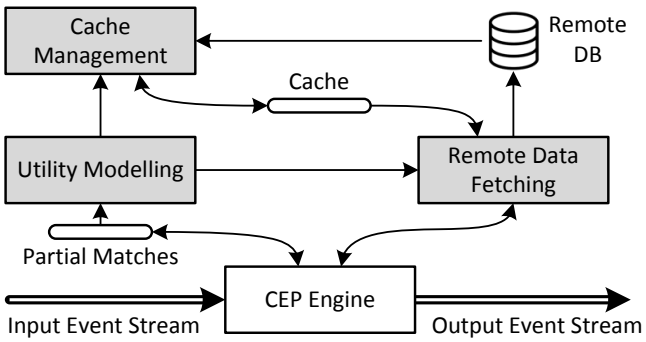


Figure 5.4: Components of the EIRES framework.

Cache management calls for a mechanism to use the available storage optimally, which again requires to assess the impact that data elements have on the performance of query evaluation.

5.2.3.2 Framework Components

To realise the above ideas, the EIRES framework comprises three main components. As illustrated in Figure 5.4, these components extend the functionality of a traditional ESP engine.

Utility modelling. This component includes a cost model to assess the (expected) utility of remote data elements for pattern detection query evaluation. As such, it provides the basis to select data elements for prefetching (P2); to decide on the partial matches for which lazy evaluation is applied (L2); and to govern the cache management (C1). Since realisations of all three operations interact with each other, the respective cost model shall provide a unified view on the utility of remote data elements. This component is discussed in Section 5.3.

Remote data fetching. This component realises the PFetch and LzEval strategies. Using the cost model for the utility of remote data elements, it manages the trade-offs induced by operations (P1) and (P2) for PFetch, and by (L2) and (L3) for LzEval. It is important to note that these trade-offs are managed in an *online* manner, *i.e.*, based on the information available at a specific point in time. The component does not create an optimal plan that combines PFetch and LzEval for some future state, because such an optimal plan requires the stable utility of data elements. Yet, because utility modelling is based on the maintained partial matches, the utility of an element may be subject to change. In particular, it may differ at the time a (pre)fetching decision is taken and at the time the data arrives at the ESP engine, *i.e.*, the utility changes during the remote data transmission. Therefore, this component realises the operations for prefetching and lazy evaluation, whereas the cache management is handled separately. We introduce the details in Section 5.4.

Cache management. This component realises operation (C1), *i.e.*, retains the data elements in the cache that are most beneficial. To this end, it exploits the aforementioned cost model. Following

Algorithm 1: EIRES workflow.

Input: Input event $S(k+1)$;
Query Q with time window τ_Q;
Partial matches $P(k),\ldots,P(k-\tau_Q)$;
ESP engine f_Q;
Cache $\mathbb{C}$.

Output: Matches $C(k+1)$;
Partial matches $P(k+1)$.

1 $U, \#P(k) \leftarrow$ `utilityEstimation` $(Q, P(k),\ldots,P(k-\tau_Q))$; `// Alg. 2`

2 $T, O \leftarrow$ **prefetchTiming**(); `// P1 in PFetch, Alg. 3`

3 $P', C' \leftarrow f_Q(S(k+1), P(k), D(k+1) \cap \mathbb{C})$;

4 $D' \leftarrow \emptyset$;

`// Determine data elements for prefetching by lookahead timing`

5 **foreach** $p \in P' \setminus P(k)$ **do** `// For each new partial match`

`// For each` d `for which` p `is in its prefetch category`

6 $\forall\, d \in D(p, k+1)$: **if** p *is in category* $T(d)$ **then** $D' \leftarrow D' \cup \{d\}$;

`// Determine data elements for prefetching by estimated arrival timing`

7 **foreach** $p \in \bigcup_{k-\tau_Q \le i \le k} P(i)$ **do** `// For each partial match`

`// For each` d `for which the time offset to fetch has passed for` p

8 $\forall\, d \in D(p, k+1)$: **if** p *is older than* $O(d)$ **then** $D' \leftarrow D' \cup \{d\}$;

`// P2 in PFetch: Prefetch elements not in cache that have high utility`

9 **foreach** $d \in D' \setminus \mathbb{C}$ **do**

10 **if** $U(d, k, k+\tau_Q) > \min_{d' \in \mathbb{C}} U(d', k, k+\tau_Q)$ **then** `Prefetch` d;

11 $P'', C'' \leftarrow$ `LzEval`$(S(k+1), P', D(k+1), \mathbb{C}, f_Q, \#P(k))$; `// Alg. 4`

12 $C(k+1) \leftarrow C' \cup C''$;

13 $P(k+1) \leftarrow P' \cup P''$;

14 **return** $P(k+1), C(K+1)$;

the above argument on the infeasibility of an optimal plan of prefetching and lazy evaluation for some future state, cache management is also conducted *online*, using the information currently available. As such, cache management decisions are taken independently of the component for remote data fetching, even though both are potentially linked through the unified assessment of the utility of data elements. We elaborate on cache management policies in Section 5.5.

EIRES workflow. All above components must cooperate for the desired efficacy. As shown in Alg. 1, when processing an input event, EIRES first estimates utility of remote data (line 1), which is further detailed in Alg. 2. It then computes remote data prefetching time (operation (P1) in PFetch, line 2), based on Alg. 3. After retrieving and evaluating required data elements that are cached locally (line 3), EIRES performs operation (P2) in PFetch, prefetching remote data according to the computed prefetch timing (line 5-10). It prepares remote data to process current

and future input events. If required data elements are not available from the cache, EIRES performs lazy evaluation (line 11), which is further explained in Alg. 4. Finally, new matches and partial matches are derived (line 12-13).

5.3 Utility Modelling

We first present a cost model to assess the utility of remote data elements (section 5.3.1). Because the utility is in part determined by future system state which can only be computed in retrospect, we also present a method for its efficient estimation (section 5.3.2).

5.3.1 Utility Definition

During query evaluation, the role of a remote data element is to enable the evaluation of some query predicate that determines whether a partial match shall be discarded, extended, or split up. Therefore, the utility of a data element is primarily based on the number of partial matches for which the element is required in their evaluation. However, since a fetched data element is cached, it may be used to evaluate respective predicates for upcoming partial matches as well. Against this background, our utility assessment considers both, the currently maintained partial matches, which induce the *urgent utility*, and future partial matches potentially derived from them, which induce the *future utility*.

Consider a point in time when a stream prefix $S(..k)$ has been processed, so that $P(k)$ is the set of current partial matches. The ESP engine needs to handle event $S(k+1)$. Recall that, for a partial match $p \in P(k)$, $D(p,k+1)$ is the set of remote data elements required for processing, see Section 2.2.2. For a data element $d \in \mathscr{D}$, at this point in time, the *urgent utility* is defined as the number of partial matches that require d or one of its constituents (with ρ^* as the reflexive transitive closure of the part-of-relation ρ), weighted by its transmission latency $\ell_{remote}(d)$:

$$UU(d,k) = \ell_{remote}(d) \cdot \left|\{p \in P(k) \mid \exists\, d' \in D(p,k+1) : d \in \rho^*(d')\}\right|. \tag{5.1}$$

In the same vein, the number of partial matches that require d, or its constituents, can be considered for some future state of query evaluation. Given perfect information about the future stream up to a stream index $k' > k$, the *future utility* of d is given by the urgent utilities of future partial matches in $k < i \leq k'$:

$$FU(d,k,k') = \sum_{k<i\leq k'} UU(d,i) \tag{5.2}$$

However, because information about future state is inherently uncertain, we actually need to compute an estimate for the future utility up to a time point k' (see Section 5.3.2), which we

denote as $\hat{FU}(d,k,k')$. As such, in the remainder we rely on an *overall utility* that is defined as the weighted sum of the above measures, defined, with $\omega \in [0,1]$, as:

$$U(d,k,k') = \omega \cdot UU(d,k) + (1-\omega) \cdot \hat{FU}(d,k,k') \tag{5.3}$$

Here, the rational is that the weighting enables tuning of the respective importance of the known utility UU and the estimated utility $\hat{FU}$. Moreover, different weighting schemes are applied when incorporating the utility assessment for the realisation of PFetch, LzEval, and cache management. Specifically, the strategies for fetching remote data, PFetch and LzEval, assign higher weights to the urgent utility than the cache management. The latter is more effective when catering for the requirements of partial matches in terms of remote data over a longer time span.

Note that the greediness of event selection (Section 2.2.2) is implicitly incorporated in the above utility model, because its semantics is reflected in the number of partial matches, which is directly captured by the urgent utility. While the estimation of future utility does not directly capture greediness, it is indirectly taken into account: The estimation implicitly incorporates dependencies between counts of partial matches at different categories of state (Section 2.2.2), at least on an aggregated level (aggregation from stream index k to k' in Equation 5.2).

Furthermore, our utility model is able to cope with multiple queries in a straightforward manner: The utility of a data element is assessed based on its related current and future partial matches, regardless of the query for which these partial matches have been created. Sharing of data elements among queries is thereby captured directly in our cost model. If queries are assigned priorities, these need to be used as weights in the utility definition in Equation 5.1.

5.3.2 Utility Estimation

To estimate the future utility $\hat{FU}$ of a data element, we determine the expected number of partial matches for which the element is relevant to its evaluation. Since the utility of data elements needs to be materialised in an online manner, estimating this should not incur significant overhead. Therefore, we estimate the number of relevant partial matches by considering two aspects: (i) how many partial matches of a particular category are expected at a time point and (ii) to what fraction of these partial matches an element is relevant. Algorithm 2 demonstrates the estimation procedure. Note that the urgent utility, UU, is directly monitored (line 2 and 6).

Number of partial matches. Recall that we partition partial matches into *categories*, where the partitioning is determined by the adopted computational model (Section 2.2.2). That is, in an automata-based model, each state of the automaton denotes a category, while in a tree-based model, the partitioning stems from the operator buffers. Intuitively, the category of a partial match, through a set of query predicates, induces a category of data elements that may be needed for the evaluation of its predicates. *e.g.*, in Figure 2.2, all partial matches of state q_3 of the automaton require the evaluation of predicate σ_3 when processing an event. The evaluation of σ_3

Algorithm 2: Utility estimation.

Input: Query Q with time window τ_Q;
Partial matches $P(k), \ldots, P(k-\tau_Q)$, partitioned in categories $1 \leq j \leq n$, *i.e.*, $P^1(k)$, $P^2(k)$, ..., $P^n(k-\tau_Q)$;
System configuration parameter: Weighting factor ω.

State: Maintained transition counts per remote reference key *tranKey* and per category of partial matches *tranCategory*.

Output: Utility function U;
Estimated number of partial matches $\#P(k)$ partitioned in categories $\#P^i(k), 1 \leq i \leq n$.

1 **foreach** $d \in \mathscr{D}$ *required by a partial match* $p \in P(k) \setminus P(k-1)$ *of category* j **do**
  // Increase urgent utility because of new partial match
2 $UU(d,k) \leftarrow UU(d,k) + 1$;
  // Update auxiliary counts for future utility estimation
3 $tranKey(d,j,k) \leftarrow tranKey(d,j,k) + 1$;
4 $tranCategory(j,k) \leftarrow tranCategory(j,k) + 1$;
5 **foreach** $d \in \mathscr{D}$ *required by a partial match* $p \in P(k-1) \setminus P(k)$ **do**
  // Decrease urgent utility due to timed out partial matches
6 $UU(d,k) \leftarrow UU(d,k) - 1$;
7 **foreach** $d \in \mathscr{D}$ *required by partial matches of category* j **do**
  // Estimate future utility
8 $Pr(j,d,k) \leftarrow \sum_{i=k-\tau_Q}^{k} tranKey(d,j,i) \;/\; \sum_{i=k-\tau_Q}^{k} tranCategor(j,i)$;
9 $\#P^j(k) \leftarrow \mathbf{AVG}\left(|P^j(k-\tau_Q)|, |P^j(k-\tau_Q+1)|, \ldots, |P^j(k)|\right)$;
10 $\hat{FU}(d,k,k+\tau_Q) \leftarrow \sum_{\substack{1 \leq i \leq n \\ d' \in \mathscr{D}, d \in \rho^*(d')}} \#P^i(k) \cdot Pr(i,d',k)$;
  // Compute overall utility
11 $U(d,k,k+\tau_Q) \leftarrow \omega \cdot UU(d,k) + (1-\omega) \cdot \hat{FU}(d,k,k+\tau_Q)$;
12 **return** U, $\#P(k)$;

refers to the remote data element $r[q_1.user]$, *i.e.*, the set of known locations associated with a particular `user`. Therefore, a data element d that refers to such a set of locations is potentially relevant to any such partial match.

Let $\{1, \ldots, n\}$ be the identifiers of the categories of partial matches (*i.e.*, automaton states or tree buffers). Then, we denote the expected number of matches of a category j at time point k as $\#P^j(k)$. To efficiently estimate $\#P^j(k)$, we compute it as the average number of partial matches of category j over a time window of fixed size, as shown in line 9, Alg. 2.

Partial match relevance. However, a specific element is only truly relevant to a subset of partial matches of a particular category, which is determined based on the payload of a partial match's events. *e.g.*, continuing on the above example, a data element d, capturing the locations associated with a particular `user`, is only relevant to those partial matches of which the first

event relates to that same `user`. To estimate the fraction of partial matches of a category to which a particular data element is relevant, we incorporate a stochastic model. Given a category of partial matches j and a data element $d \in \mathscr{D}$, we use $Pr(j,d,k)$ to capture, at time point k, the probability that element d is required to evaluate the predicates of partial matches of category j when processing an input event. We assume that such a probability is relatively stable in the short term. The probability distribution may be derived from the value distribution of the events' attributes that serve as a reference for the selection of data elements. Here, the value distribution and, hence, the probability distribution, again, is computed adopting a sliding window. In Alg. 2, line 3, 4, and 8 show how to monitor and maintain it online.

Future utility. Based on the above, we estimate the future utility at the time the stream prefix $S(..k)$ has been processed up to some future stream index $k' > k$ for a data element $d \in \mathscr{D}$, as follows:

$$\hat{FU}(d,k,k') = \quad (k'-k) \sum_{\substack{1 \le j \le n \\ d' \in \mathscr{D}, d \in \rho^*(d')}} \#P^j(k) \cdot Pr(j,d',k) \tag{5.4}$$

Both, the average number of partial matches per category and the distribution of attribute values (as the basis for the probability distribution of the data access per category), are based on simple counts, which is why they can be maintained efficiently for a sliding window (line 3-4 and 8-10 in Alg. 2). Nevertheless, their computation still involves computational effort. Instead of recomputing all utility values every time a single event is processed, we lift the utility model through a temporal abstraction. Specifically, we consider the utility values as being static for a time slice, *i.e.*, a number of subsequent stream indices. Recall that τ_Q denotes the time window of a query Q, which determines the maximal time-to-live of a partial match. Then, using m as the number of slices in which a window is split, the length of a time slice, and thus the frequency with which utility values are recomputed, is given by τ_Q/m. Here, parameter m allows one to trade off the precision of the utility model versus the incurred computational overhead.

5.4 Remote Data Fetching

The EIRES framework defines PFetch and LzEval as two strategies to fetch remote data, which we describe in Section 5.4.1 and Section 5.4.2, respectively. As illustrated in line 5-10, Alg. 1, EIRES always performs prefetching. If prefetching failed to prepare required remote data on time, EIRES performs lazy evaluation (line 11).

5.4.1 Prefetching

As explained in Section 5.2.3, three operations need to be instantiated for PFetch: deciding when prefetching is beneficial for a data element (P1), selecting the elements to prefetch (P2) and

Algorithm 3: Prefetch timing (operation (P1) in **PFetch**)

State: Monitored event input rates $\lambda[n]$;
Latest prefetch cache hit history $\mathscr{H}$.

Output: Prefetch timing function T;
Offset timing function O.

```
1  T ← ∅; O ← ∅;
2  foreach d ∈ 𝒟 required by partial matches of category m do
3      j ← m's directly preceding category;
4      while j ≺ m do
           // Check if lookahead timing can be applied
5          if partial matches of category j require d ∧ j > 1 then
6              if ℋ(m, j, d) = true then                    // Prior cache hit successful
7                  O(d) ← 0, T(d) ← j ;                     // Set time offset to be 0
8                  break;
9              else j ← j's directly preceding category ;   // Try next category
10         else                                             // Use estimated-arrival timing
11             O(d) ← 1/λ[j] − ℓ_remote(d), T(d) ← j ;      // Set time offset
12             break;
13 return T, O;
```

performing the actual fetch operation (P3). Here, operation (P3) is straightforward. Hence, we explain operations (P1) and (P2) in detail.

(P1): Prefetch timing. Intuitively, the best time to prefetch a data element d is such that it arrives *right before* it is needed. As discussed in Section 5.2.3.1, let t_n be the time when a data element is needed for query evaluation. Then, with $\ell_{remote}(d)$ as the transmission latency, and under a negligible latency for accessing the cache, the *perfect* time to trigger prefetching for the data element is $t_p = t_n - \ell_{remote}(d)$.

Now, let t_i be the time at which a fetch request for a data element is actually initiated. Prefetching early ($t_i > t_p$) hides the transmission latency completely. Yet, early fetching may be based on an inaccurate estimation of the (future) utility of d. Hence, in the worst case, d is prefetched and consumes space in the cache, but is never used in query evaluation before it is evicted due to limited cache capacity. On the contrary, late prefetching ($t_i < t_p$) generally benefits from a more accurate utility estimation, but hides the transmission latency only partially. To estimate the appropriate time to prefetch in light of this trade-off, we propose complementary techniques: lookahead timing and estimated-arrival timing, with the workflow shown in Alg. 3.

Lookahead timing. First, we strive to identify partial match categories that provide useful indicators of when to prefetch a data element related to another match category, *i.e.*, a *lookahead* category. To do this, we exploit the partial order of the categories of partial matches as they are derived from the query predicates. This order corresponds to an order of states of an automaton

or buffers in a tree-based model, which captures that partial matches of one category may become those of another category during query evaluation.

Let $\{1,\ldots,n\}$ be the identifiers of categories of partial matches with m being a category of matches that require a data element d (line 2 in Alg. 3). Then, we determine a lookahead category j of matches that is a predecessor of m in the partial order of categories, denoted $j \prec m$, meaning that partial matches of category j potentially develop into those of category m. From the set of all preceding categories $\{1 \leq j \leq n \mid j \prec m\}$, we choose j such that (i) its partial matches contain events of which the payload serves as a reference to identify d, and (ii) it is closest to m in the partial order while still allowing for timely and accurate prefetching. This process is detailed in line 3-9 in Alg. 3. Here, without loss of generality, we illustrate the situation that category m has one directly preceding category, though in practice it may have multiple such categories. If multiple categories contain references to the *same* data elements, multiple prefetches could be merged into a single fetch through semantic query rewriting. If they contain references to *different* data elements, a traverse order could be enforced over all these directly preceding categories, *i.e.*, ordered by monitored transmission latencies. EIRES incorporates both of these approaches.

To achieve this, we determine the lookahead category j in a dynamic manner, based on the recent cache hit history from cache management (input $\mathscr{H}$ in Alg. 3). Particularly, for a given data element d, $\mathscr{H}$ maintains cache hit/miss information, where $\mathscr{H}(i,i',d)$ returns *false* if d was prefetched upon the construction of a partial match of category i, but was not available when required during the evaluation of a partial match of category i'. Such a cache miss can have two causes: First, it may hold that $t_{i \rightarrow i'} < \ell_{remote}(d)$, *i.e.*, prefetching happens too late because the time for a partial match to develop from category i to i' is shorter than the transmission latency. Second, when constructing a partial match of category i, the utility estimation may have been inaccurate, so that the wrong data elements have been fetched. Either way, such a recent cache miss indicates that the respective category is not suited to trigger prefetching.

In sum, for a partial match of category m that requires a data element d, we select the preceding category j closest to m for which $\mathscr{H}(j,m,d) = true$ (line 6 in Alg. 3). Then, the point in time to prefetch data element d for partial matches of category m is defined as the moment that a partial match for category j is constructed. For example, if $\mathscr{H}(j_1,m,d)$ and $\mathscr{H}(j_2,m,d)$ are both *true*, and $j_1 \prec j_2$, we select j_2 as the lookahead category for m, avoiding the unnecessarily early fetching that using j_1 would yield. If partial matches of multiple categories m reference d, the smallest index j, in terms of the partial order $\prec$, is chosen. Intuitively, the recent cache hit history $\mathscr{H}$ can be implemented as a matrix, as shown in Figure 5.5, where a cell with value one means a cache hit and value zero means a cache miss. A cell with a box means the selected lookahead category.

In practice, we deal with event stream fluctuations, as follows. The recent cache history $\mathscr{H}$ contains counts of cache misses, so that a threshold determines what to interpret as negative

evidence. Also, count values are reset to zero after a fixed time period after their last increment.

	...	q_3(t1.user)	...
q_1	...	1	...
q_2	...	**1**	...
q_3	...	0	...
...	...	...	...

(a) Cache hit matrix at $S(k)$.

	...	q_3(t1.user)	...
q_1	...	**1**	...
q_2	...	0	...
q_3	...	0	...
...	...	...	...

(b) Cache hit matrix at $S(k'), k' > k$.

Figure 5.5: Recent cache hit history $\mathscr{H}$ implemented as a cache hit matrix.

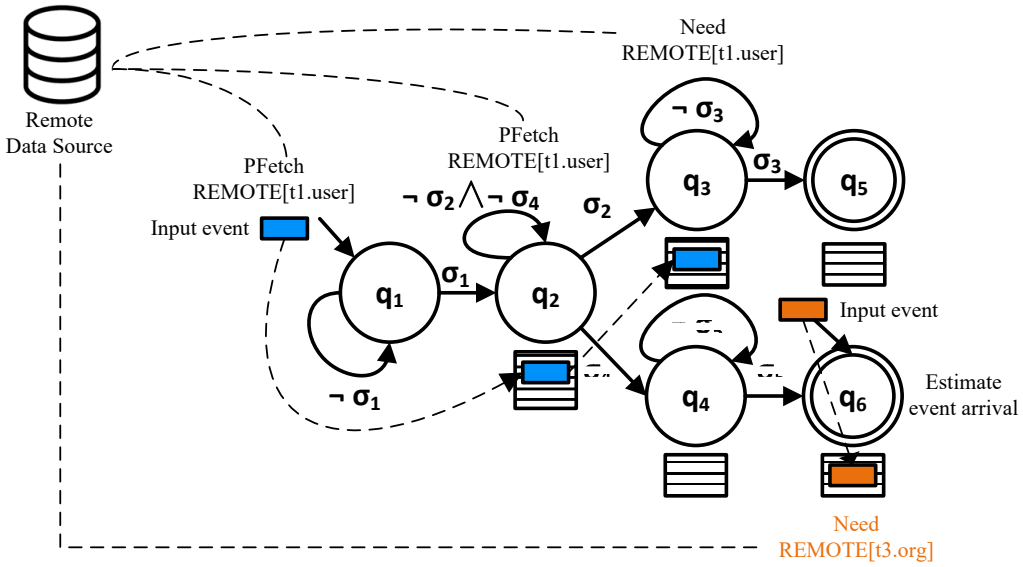


Figure 5.6: Automata-based execution model with the partial order of partial match evolvement.

To illustrate the lookahead timing mechanism, we revisit the automaton of Figure 2.2 by adding the partial order $\prec$ of different categories of partial matches, as shown in Figure 5.6. Here, categories of predicates are given by the automaton states and it holds, *e.g.*, that $q_1 \prec q_2 \prec q_3$. Consider partial matches of category q_3 that require the set of known locations of a credit card for evaluation (in predicate σ_3). The attribute value used to identify this remote data (`r[q1.user]`), is part of partial matches of categories q_1 and q_2. Hence, fetching may be triggered when constructing a partial match of either of the categories q_1, q_2, and q_3. If the recent cache hit history (matrix) at the k-th event $S(k)$ is given as in Figure 5.5(a), we observe that triggering the fetching of `r[q1.user]` only upon the creation of a partial match of category q_3 is not sufficient to hide the transmission latency ℓ_{remote}, because $\mathscr{H}(q_3, q_3, \texttt{t1.user}) = 0$. However, prefetching when creating a partial match of category q_1 or q_2 is shown to hide ℓ_{remote} ($\mathscr{H}(q_1, q_2, \texttt{t1.user}) = 1$, $\mathscr{H}(q_1, q_3, \texttt{t1.user}) = 1$). Because q_2 is closer to q_3 than q_1, it becomes the lookahead category, avoiding the unnecessarily early fetching that q_1 would yield. At a later time (the k'-th event $S(k')$, $k' > k$), due to the dynamics of the stream and the variation of the network connection, the recent cache hit history $\mathscr{H}$ is changed as illustrated in Figure 5.5(b). This time, prefetching upon

the creation of a partial match of category q_2 is not sufficient and the category of q_1 is selected as the lookahead category.

Note that there may be partial match categories for which lookahead timing is unsuitable, *i.e.*, cache misses occur since there is no lookahead category enabling timely and accurate prefetching. This occurs, *e.g.*, when the reference required to identify d is part of the payload of the input event that leads to the creation of the partial match of category m. An example for that is the category q_4 in Figure 2.2, which requires the evaluation of a predicate using remote data on pre-authorised clients (`r[q3.org]`) that can only be fetched based on the event that lead to the respective match. For those cases, we instead determine the time to prefetch based on *estimated-arrival timing*.

Estimated-arrival timing. When lookahead timing is not applicable for a partial match category m, we instead determine the time to prefetch by incorporating the inter-arrival time of events that satisfy the predicates to check for partial matches of category m but are *not* based on remote data. Consider again category q_4 in Figure 5.6. The respective partial matches are extended when an event with `x.type=T`, `x.vol>50k`, and `x.ben ∉ r[q4.org]` is received. Since lookahead timing is not applicable for the remote data, we estimate the expected time until an event satisfying `x.type=T` and `x.vol>50k` is received. This way, we obtain an estimate for the time between the creation of a partial match of category q_4 and the time the data element `r[q4.org]` is actually needed. To derive this inter-arrival time, various stochastic processes for event arrival may serve as a foundation. Selecting one of them, their parameters shall be learned from historic data or through monitoring by the ESP engine.

Here, we illustrate the general procedure when the event arrival follows a Poisson process [105], as observed in many domains where events correspond to requests triggered by people. Then, events are independent and occur with a constant mean arrival rate λ, which is the only parameter that needs to be monitored. Inter-arrival times are exponentially distributed with their expectation being $E = 1/\lambda$. Based thereon, at time t, we estimate the time the remote data is needed as $t + 1/\lambda$, so that the time for prefetching becomes $t_p = t + 1/\lambda - \ell_{remote}(d)$ (line 11 in Alg. 3). Taking up the previous example, events with `x.type = T` and `x.vol > 50k` may be rare, such that the expected inter-arrival time is $E = 300ms$. Then, with a data transmission latency of $\ell_{remote} = 50ms$, we trigger prefetching only $250ms$ after the creation of the respective partial match (*i.e.*, when processing the next input event after this period).

(P2): Prefetch selection. Using the above techniques, we determine the point in time at which prefetching is beneficial for a particular data element. However, at a specific moment while processing the event stream, we still need to select those data elements for which prefetching shall actually be invoked. This decision is taken using our utility model: We only prefetch data elements, for which prefetching is beneficial at that point in time and for which the utility is higher than the minimum utility value of an element currently in the cache (line 9-10 in Alg. 1). At time (stream index) k, let $\mathbb{C}(k) \subseteq \mathscr{D}$ be the content of the cache and $D(k)$ the set of data elements

Algorithm 4: Lazy evaluation (**LzEval**)

Input: Input event $S(k+1)$;
Partial matches $P(k)$;
Remote data $D(k+1)$;
Cache $\mathbb{C}$;
ESP engine f_Q;
Estimated number of partial matches $\#P(k)$ partitioned in categories $\#P^i(k), 1 \le i \le n$.

State: Monitored event input rates $\lambda[n]$.

Output: Matches $C(k+1)$; partial matches $P(k+1)$.

1 $C(k+1) \leftarrow \emptyset$; $P(k+1) \leftarrow \emptyset$; $s\hat{u}cc \leftarrow \emptyset$;
2 **foreach** $d \in D(k+1) \setminus \mathbb{C}$ *required by partial matches of category* j **do**
3 $\quad \lambda \leftarrow \lambda[j]$; $\ell \leftarrow \ell_{remote}(d)$;
4 $\quad$ **foreach** category m, $j \prec m$ **do**
$\quad\quad$ // if not yet known, estimate if postponing is beneficial
5 $\quad\quad$ **if** $\langle m, \ell \rangle$ is not checked against $s\hat{u}cc(j, \ell)$ **then**
6 $\quad\quad\quad \lambda \leftarrow \lambda + \lambda[m]$; $\quad E(j,m) \leftarrow 1/\lambda$;
7 $\quad\quad\quad \Delta_- \ell_{remote} \leftarrow \mathbf{min}(E(j,m), \ell)$;
8 $\quad\quad\quad \Delta_+ \ell_{match} \leftarrow \ell_{pm} \prod_{1 \le i \le m} \left(\#P^i(k) \cdot \lambda[i+1] \cdot E(j,m)\right)$;
9 $\quad\quad\quad$ **if** $\Delta_- \ell_{remote} > \Delta_+ \ell_{match}$ **then**
10 $\quad\quad\quad\quad s\hat{u}cc(j, \ell) \leftarrow s\hat{u}cc(j, \ell) \cup \{m\}$;
11 $\quad\quad\quad$ **else** $s\hat{u}cc(j, \ell) \leftarrow s\hat{u}cc(j, \ell) \setminus \{m\}$;
$\quad\quad$ // benefit expected, postpone evaluation of d's predicates
12 $\quad\quad$ **if** $\langle m, \ell \rangle \in s\hat{u}cc(j, \ell)$ **then**
13 $\quad\quad\quad$ **Fetch** d;
14 $\quad\quad\quad P' \leftarrow f_Q(S(k+1), P(k))$; $\quad$ // ignore d's predicates
15 $\quad\quad$ **else Fetch** d, block stream processing until *after* d arrives at $\mathbb{C}$;
16 $\quad\quad P'', C'' \leftarrow f_Q(S(k+1), P(k) \cup P', D(k+1) \cap \mathbb{C})$;
17 $\quad\quad P(k+1) \leftarrow P(k+1) \cup P''$;
18 $\quad\quad C(k+1) \leftarrow C(k+1) \cup C''$;
19 **return** $C(k+1), P(k+1)$;

for which prefetching at time k would be beneficial. We select $D'(k)$ for prefetching, defined as

$$D'(k) = \{d \in D(k) \mid U(d) > \min_{d' \in \mathbb{C}(k)} U(d')\}. \tag{5.5}$$

5.4.2 Lazy Evaluation

For LzEval, in turn, three operations need to be instantiated (Section 5.2.3): Operation (L1), fetching remote data on demand is trivial. Therefore, we mainly discuss operation (L2), selecting the partial matches for which the strategy is applied and operation (L3), adapting the evaluation procedure for those matches. The workflow of LzEval is sketched in Alg. 4.

(L2): Selection of partial matches. LzEval triggers a fetch operation for remote data d when required, but postpones the actual evaluation of the respective predicates. While this hides the transmission latency $\ell_{remote}(d)$, it also makes event selection less strict, possibly resulting in an exponentially increasing number of partial matches. As such, the inherent evaluation latency ℓ_{match} may increase, which may thwart the benefit of the reduction of $\ell_{remote}(d)$. Recognizing this trade-off, we therefore only apply LzEval to those partial match categories where an actual benefit is expected.

To determine these categories, we estimate the time gained by hiding the transmission latency, denoted as $\Delta_{-}\ell_{remote}$, and the overhead incurred by the additional partial matches, $\Delta_{+}\ell_{match}$. The latency gain, $\Delta_{-}\ell_{remote}$, depends on the difference between the time t_n at which a data element is needed (and fetching is triggered), and when the predicate is actually evaluated t_e. Ideally, evaluation happens after the data is available (*i.e.*, $t_a < t_e$), so that the entire transmission latency is hidden, *i.e.*, $\Delta_{-}\ell_{remote} = \ell_{remote}(d)$. The overhead $\Delta_{+}\ell_{match}$ depends on the number of additional partial matches caused by lazy evaluation during $t_e - t_n$ and the additional latency incurred per partial match. While the latter is assumed to be a known constant, ℓ_{pm}, the other parameters need to be estimated.

To estimate the number of additional partial matches, we follow an approach similar to the estimated-arrival timing (Section 5.4.1). Assume that the evaluation of a predicate of category j requires remote data, yet that this evaluation may be postponed until the creation of a partial match of a later category m, $j \prec m$. Then, we estimate the average time $t_{j\to m}$ for a partial match of category j to develop into one of category m, ignoring all predicates that are related to remote data. As before, we assume that, for each possible extension of a partial match, the respective event arrivals follow a separate Poisson process of a monitored rate. Then, we derive an estimate for $t_{j\to m}$ based on a compound Poisson process induced by the sequence of intermediate categories $\{r_1,\ldots,r_s\}$ of partial matches, $j \prec r_1 \prec \ldots \prec r_s \prec m$. With λ_i as the rate of the process describing the arrivals that construct partial matches of category i, the expectation of the compound Poisson process is $E(j,m) = {}^{1}/_{\sum_{i\in\{r_1,\ldots,r_s,m\}}\lambda_i}$ (line 6 in Alg. 4). Although the estimate assumes all Poisson processes to be independent, *i.e.*, it ignores sequential dependencies between the categories, it suffices as an upper bound for our purposes.

Using $E(j,m)$ as an estimate for $t_{j\to m}$, we derive the remaining parameters. The hidden part of the transmission latency is given as $\Delta_{-}\ell_{remote}(j,m) = \min(E(j,m),\ell_{remote}(d))$ (line 7 in Alg. 4). The estimation of $\Delta_{+}\ell_{match}(j,m)$ is time-varying and incorporates $\#P^i(k)$, *i.e.*, the expected number of partial matches of category i, as estimated at time point (stream index) k, see Section 5.3.2. For each category i, this number is multiplied with the arrivals of events that may extend the match by satisfying all the predicates not based on remote data (λ_{i+1}) and the estimate for $t_{j\to m}$ ($E(j,m)$) (line 8 in Alg. 4). While this estimates the number of additional partial matches during $t_e - t_n$, multiplying it with the constant additional evaluation latency

per partial match (ℓ_{pm}) yields the estimate for accumulated increase in evaluation latency (to simplify notation, we define $r_{s+1} = m$):

$$\Delta_{+}\ell_{match}(j,k)(k) = \ell_{pm} \prod_{1 \leq i \leq s} \left(\#P^{r_i}(k) \cdot \lambda_{r_{i+1}} \cdot E(j,m)\right) \tag{5.6}$$

For each category j that requires remote data during query evaluation, we determine the set of succeeding categories for which lazy evaluation is beneficial. This set is defined as $\hat{succ}(j) = \{m \in \{1,\ldots,n\} \mid j \prec m \wedge \Delta_{-}\ell_{remote} > \Delta_{+}\ell_{match}\}$, *i.e.*, the hidden part of the transmission latency is larger than the overhead by the increased evaluation latency. Then, all partial matches of a category j, for which $\hat{succ}(j)$ is non-empty, are selected for lazy evaluation (line 9-11 in Alg. 4). Conceptually, $\hat{succ}(j)$ must be computed for every different transmission latency, $\ell_{remote}(d)$ (line 5 in Alg. 4). In practice, to improve efficiency and to reuse results, transmission latency may be lifted to coarser granularities, *e.g.*, to a millisecond-level.

(L3): Adapted evaluation procedure. Consider partial matches of a category j and $\hat{succ}(j)$ as the succeeding categories for which lazy evaluation is beneficial. Also, let $S(..k)$ be the time the stream prefix was processed and $\mathbb{C}(k) \subseteq \mathscr{D}$ as the cache content. Then, query evaluation to process event $S(k+1)$ is adapted for the partial matches of category j, as follows. For each predicate that requires a remote data element d, the availability in the cache is checked. If $d \in \mathbb{C}(k)$, the predicate is directly evaluated (line 16 in Alg. 4). If not, a fetch request for d is triggered and the predicate is marked as postponed (line 12-14 in Alg. 4). The same procedure is followed as long as the partial match of category j develops into one of a category $m \in \hat{succ}(j)$. Upon construction of a respective partial match, the cache is checked and, upon data availability in the cache, each predicate is evaluated.

Once a partial match of category j develops into a category $m' \notin \hat{succ}(j)$ for which lazy evaluation is not beneficial, a different strategy is implemented. Postponing the predicate evaluation further would increase the overall latency. Hence, query evaluation is blocked and only continues after the data element d becomes available (line 15 in Alg. 4).

5.5 Cache Management

Cache management in EIRES shall retain the data elements that are most beneficial in the cache, thereby realising operation (C1) as introduced in Section 5.2.3. Although the benefit of a data element strongly relates to its expected utility (see Section 5.3), certain query semantics suggests cache management based on a relatively simple policy. Recalling the greediness of pattern detection queries in event selection (Section 2.2.2), either semantics (greedy selection or non-greedy selection) motivates a different policy for cache management.

LRU policy. Under a greedy query semantics, a large number of partial matches that require the same data elements can be expected to materialise. This, in turn, will induce a large number

of access requests to the cache for the respective data. In our utility model, the urgent utility then becomes a good estimator for the future utility. We can exploit this effect, even without using any computed utility value, by adopting the widely-established *least-recently-used* (LRU) policy for cache management. Data elements in the cache are ranked by the time of the last access to them, evicting those that have not been accessed for the longest time.

Cost-based policy. With non-greedy query semantics, individual data elements are expected to be accessed less often and in a diminishing manner. Hence, the current access frequency, *i.e.*, the urgent utility, no longer provides a good estimator for the future utility. In that case, cache management shall exploit the computed utility. In our cost-based policy, we separate the handling of data elements based on the fetching strategy that led to their retrieval. The reason being that, while any element requested through LzEval will certainly be required by some partial matches, this is not necessarily the case for data elements requested through, possibly inaccurate, predictions of PFetch. Against this background, we adopt two, purely conceptual, cache tiers, T_1 and T_2, to separate elements that will certainly be used (T_1) from those for which usage is uncertain (T_2). Then, elements in T_1 will be retained over all elements of T_2, but are moved to T_2 after a first access. This two-tier cache management is illustrated in Figure 5.7.

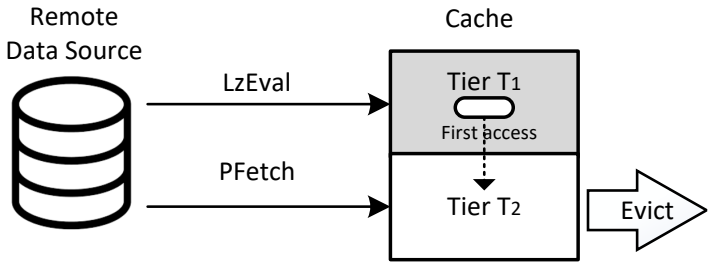


Figure 5.7: The two-tier cache management.

Once the cache capacity is reached, data elements (from T_2 before T_1) are evicted based on their utility. Let $\mathbb{C} \subseteq \mathscr{D}$ denote the current content of the cache and $b \in \mathbb{N}$ its capacity. Then, incorporating the size of data elements, the selection of the data elements $\mathscr{R} \subseteq \mathbb{C}$ to retain can be formulated as a Knapsack problem:

$$\textbf{select } \mathscr{R} \subseteq \mathbb{C} \textbf{ that maximizes } \sum_{d \in \mathscr{R}} U(d) \textbf{ subject to } \sum_{d \in \mathscr{R}} |d| \leq b$$

This is a standard, NP-hard, Knapsack problem [104]. Yet, similar to Section 4.3.3, the Knapsack problem may be approximated [46], *e.g.*, by selecting data elements in the order of their utility and size ratios, until the capacity is reached.

5.6 Evaluations

We evaluated the EIRES framework on various scenarios. Section 5.6.1 first outlines the experimental setup including datasets, queries, measures, implementations and hardware. For

Table 5.1: Synthetic datasets for efficient remote data integration.

	Attribute	Value Distribution
DS3	Type	$\mathcal{U}(\{A,B,C,D\})$
	ID	$\mathcal{U}(1,100)$
	V_1	$\mathcal{U}(1,100000)$
	V_2	$\mathcal{U}(1,100000)$
DS4	Type	$\mathcal{U}(\{A,B,C,D\})$
	ID	$\mathcal{U}(1,100)$
	V_1	Zipf distribution, probability density $P(x) = \frac{x^{-1.01}}{\zeta(1.01)}$
	V_2	Zipf distribution, probability density $P(x) = \frac{x^{-1.01}}{\zeta(1.01)}$

```
Q5: PATTERN SEQ(A a, B b, C c, D d, B e, C f, A g, D h)
    WHERE SAME[ID] AND a.v1=REMOTE[d.v1] AND a.v2=h.v2
    WITHIN 8min
Q6: PATTERN SEQ(A a, (SEQ(B b, C d, D f) OR SEQ(C c, B e))
    WHERE a.v1=b.v1 AND a.v2=e.v1
    AND d.v1=REMOTE[a.v1] AND c.v2=REMOTE[a.v2]
    WITHIN 50K
```

Listing 5.1: Queries for the synthetic dataset.

controlled experiments, Section 5.6.2 reports on the overall efficiency and effectiveness, while Section 5.6.3 explores the sensitivity of a range of parameters that affect the processing latency. Finally, Section 5.6.4 discusses experiments on two real-world scenarios.

5.6.1 Experimental Setup

Datasets and queries. For controlled experiments, we generated two synthetic datasets, DS3 and DS4, with event schemas and value distributions as given in Table 5.1. Each of them has a numeric ID, two numeric attributes v_1 and v_2. These attributes are uniformly distributed in DS3 but subject to zipf distribution in DS4. These datasets enable us to evaluate common queries that test for sequences of events of different types, which are correlated by an ID. Further correlation predicates and references to remote data may be defined for attributes v_1 and v_2. Remote data elements are also referenced by v_1 and v_2. Their attribute value distributions represent two extreme cases. DS3 considers the uniformly distributed values where variance is low. DS4, in turn, considers an extremely skewed distribution. For the synthetic datasets, we evaluate the queries Q5 and Q6 of Listing 5.1. They differ in their structures (pure sequence vs. disjunction of sequences) and the remote data integration in predicates. We further use two real-world dataset for low-latency bushfire detection and low-latency cluster monitoring. The details are be presented in Section 5.6.4.

Baselines. We compare our approaches for data fetching, PFetch, LzEval, and their combination (Hybrid) against three baselines.

- The first baseline, *BL1*, denotes the naive integration of data fetching. It interrupts query evaluation when remote data is needed and continues once it has been fetched.
- The second baseline, *BL2*, employs a cache of remote data to improve the efficiency of query evaluation. For this cache, we consider both policies discussed in Section 5.5, *i.e.*, LRU and the cost-based policy.
- The third baseline, *BL3*, first ignores all predicates related to remote data. Upon reaching a final state of the evaluation model, it fetches the remote data and conducts the respective predicates evaluation and event selection.

Measures. Our focus is the overall latency of query evaluation (defined in Section 2.3), *i.e.*, the time between the ingestion of the last event needed to construct a match and its actual detection. Depending on the experimental setting, our latency measurements are based on from 100k to 5 million matches. Specifically, we report the 5^{th}, 25^{th}, 50^{th}, 75^{th}, and 95^{th} percentiles of the latency values. Moreover, we report the throughput (defined in Section 2.3), *i.e.*, the number of events processed per second.

Implementation and environment. For the evaluation, we implemented EIRES in an automata-based ESP engine[1], written in C++. All reported results are averaged over 20 runs on a NUMA node with 4 Intel Xeon E7-4880 CPUs (60 cores, 120 threads) and 1TB RAM, running openSUSE 15.0 operating system.

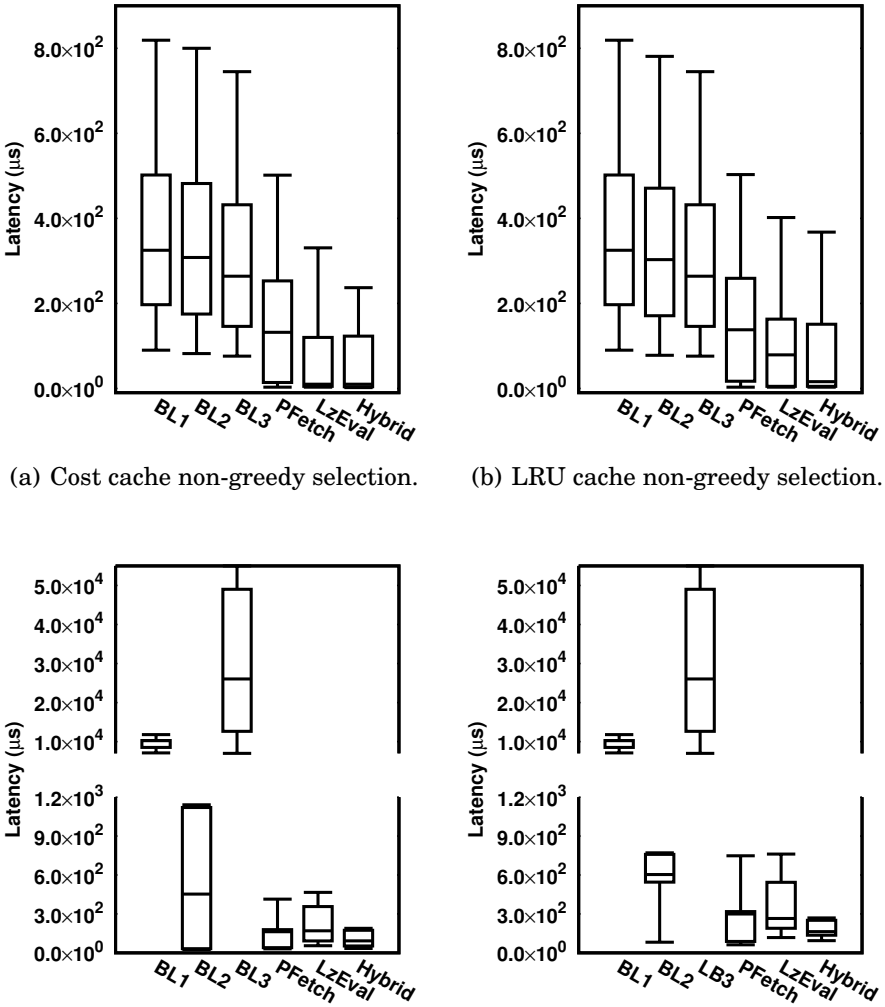
5.6.2 Overall Effectiveness and Efficiency

Figure 5.8-5.11, respectively, depict the overall performance obtained for Q5 and Q6 on the synthetic datasets DS3 and DS4. We compare the PFetch, LzEval, and Hybrid strategy against the three baselines under both greedy and non-greedy selection policies for two cache eviction policies. Here, the cache capacity is set to 10% of a remote key's value range, *i.e.*, 10,000 items, while the transmission latency of remote data is uniformly distributed between 10μs and 100μs.

As shown in the figures, the Hybrid strategy consistently outperforms all other approaches. Furthermore, both PFetch and LzEval also always outperform all three baselines considerably, though it depends on the context which of these achieves better results. We also observe similar trends for both datasets DS3 and DS4. Therefore, in the following, we focus detailed analysis on DS3, and also report results from DS4.

Selection strategies. For non-greedy selection in Q5 (Figure 5.8(a)-5.8(b)), the median latencies of Hybrid are 10μs (cost-based cache) and 16μs (LRU). Since the median latencies for the baselines range between 264μs (*BL3*) and 314μs (*BL1*), Hybrid reduces the median latency by at least

[1]The implementation and datasets are publicly available at https://github.com/zbjob/EIRES.



(a) Cost cache non-greedy selection.

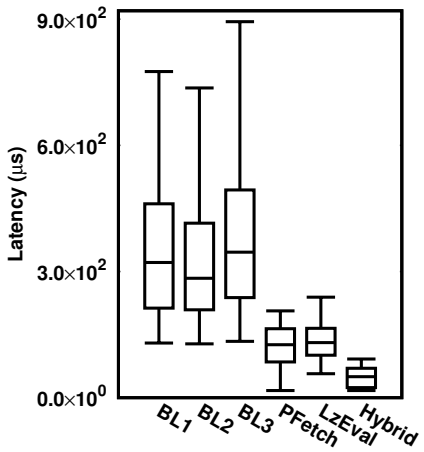
(b) LRU cache non-greedy selection.

(c) Cost cache greedy selection.

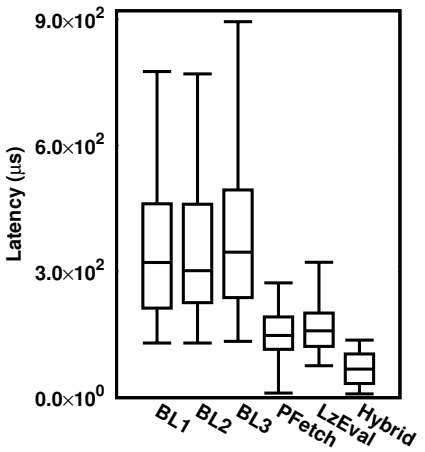
(d) LRU cache greedy selection.

Figure 5.8: Overall effectiveness and efficiency for Q5 over DS3.

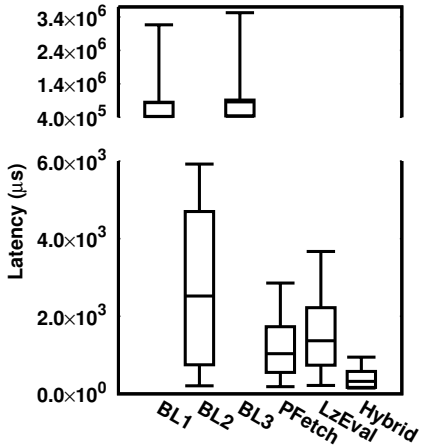
26×. For the 95th percentiles, the reduction is at least 4× (cost-based cache) and 2.5× (LRU). For Q6 (Figure 5.9(a)-5.9(b)), we observe smaller gains in median latencies, yet bigger gains for the 95th percentile latencies. We also observe different performance of *BL3* for Q5 and Q6: *BL3* outperforms both *BL1* and *BL2* for Q5, but the trend is opposite for Q6. The reason is that Q5 has two states requiring different remote data in its execution model, whereas Q6 has one at each branch of disjunction sequences. *BL3* is able to hide some transmission latency by fetching different data items at once. Therefore, for a single complete match of Q5, *BL3*'s aggregated transmission latency is the maximal latency of all required remote data, instead of their sum (as in *BL1* and *BL2*). This benefit outweighs the overhead caused by extra partial matches for non-greedy selection. Hence, *BL3* performs better. But for Q6, *BL3* has little to hide but creates extra partial matches and therefore performs worse than *BL1* and *BL2* even under non-greedy



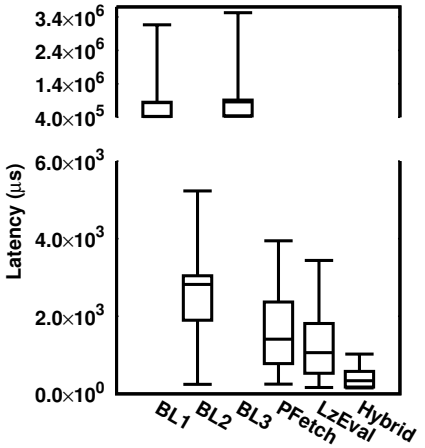
(a) Cost cache non-greedy selection.



(b) LRU cache non-greedy selection.



(c) Cost cache greedy selection.



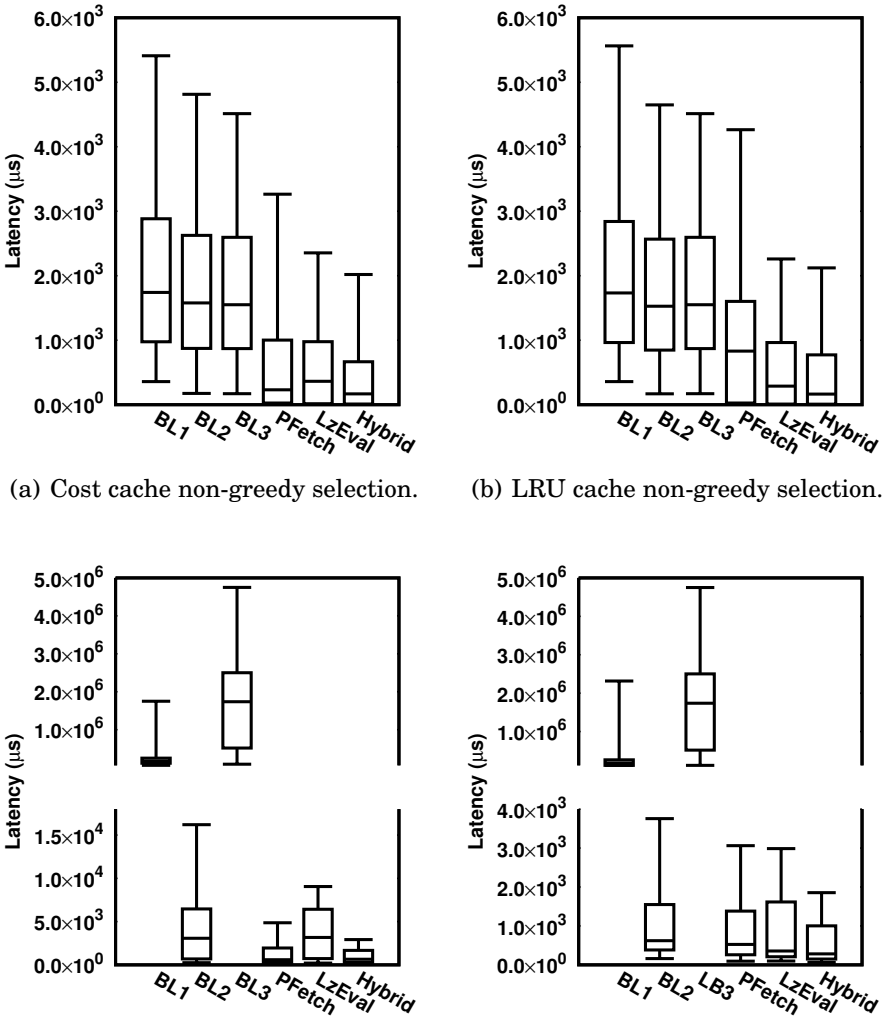
(d) LRU cache greedy selection.

Figure 5.9: Overall effectiveness and efficiency for Q6 over DS3.

selection.

The results show greater variety for greedy selection policies. For Q5 (Figure 5.8(c)-5.8(d)), Hybrid reduces the median latency by 111× (cost-based cache) and 63× (LRU) when compared to *BL1*. Further more, the reductions for *BL3* are 283× (cost-based cache) and 160× (LRU), yet the reductions are only 6× (cost-based cache) and 2.8× (LRU) when compared to *BL2*. The reductions for the 95$^{\text{th}}$ latencies are 62× (*BL1*, cost-based), 44× (*BL1*, LRU), 6× (*BL2*, cost-based), 2.8× (*BL2*, LRU), 558× (*BL3*, cost-based) and 392× (*BL3*, LRU). For Q6 (Figure 5.9(c)-5.9(d)), gains are even more extreme, *i.e.*, Hybrid reduces median latencies by up to 2,726× and the 95$^{\text{th}}$ percentiles latency by up to 3,752×.

Similar trends are observed for DS4, under non-greedy selection policy for Q5 (Figure 5.10(a)-



(a) Cost cache non-greedy selection.

(b) LRU cache non-greedy selection.

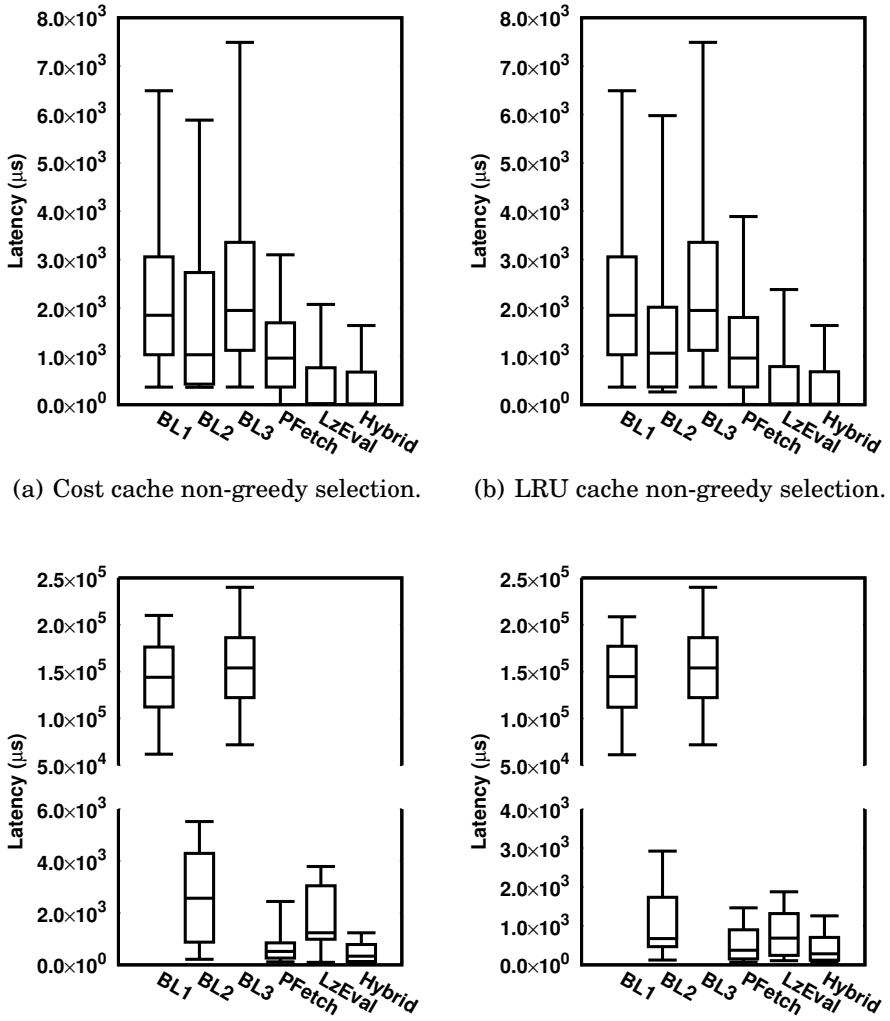
(c) Cost cache greedy selection.

(d) LRU cache greedy selection.

Figure 5.10: Overall effectiveness and efficiency for Q5 over DS4.

5.10(b)), compared to *BL1*, *BL2*, and *BL3*, with cost-based and LRU cache, Hybrid reduces median latency by 10×, 10×, 5×, 5×, 5× and 5×. It reduces the 95^{th} percentile latency by 2.7×, 2.7×, 2.4×, 2.3×, 2.2× and 2.2×. Turning to greedy selection policy (Figure 5.10(c)-5.10(d)), Hybrid reduces median latency by 266×, 1,818×, 5×, 2.2×, 2664× and 6,160×. It reduces the 95^{th} percentile latency by 599×, 1,248×, 5.5×, 2×, 1,625× and 1,348×. For Q6 non-greedy selection (Figure 5.11(a)-5.11(b)), the median latency reductions are 132×, 168×, 73×, 96×, 96× and 177×. The 95^{th} latency reductions are 4×, 4×, 3.6×, 3.6×, 4.6× and 4.5×. For Q6 under greedy selections (Figure 5.11(c)-5.11(d)), the median latency reductions are 266×, 644×, 2.3×, 4.7×, 2,664×, 6,161×. The 95^{th} latency reductions are 599×, 1,248×, 5.5×, 2.4×, 1,625×, 2,560×.

Caching policies. The impact of a local cache differs considerably depending on the scenario.



(a) Cost cache non-greedy selection.

(b) LRU cache non-greedy selection.

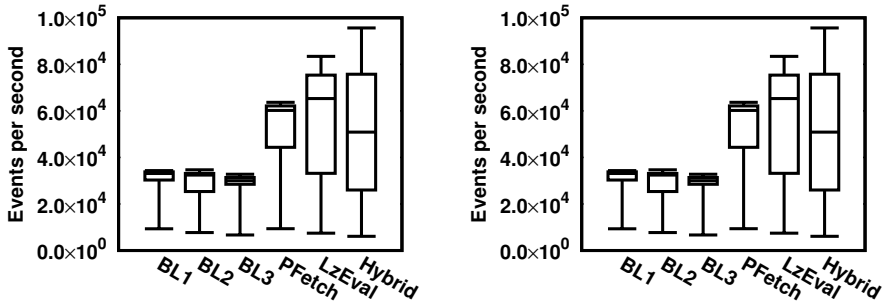
(c) Cost cache greedy selection.

(d) LRU cache greedy selection.

Figure 5.11: Overall effectiveness and efficiency for Q6 over DS4.

Adding a cache contributes little for non-greedy selection, as illustrated by the comparable performance of *BL1*, *BL2* and *BL3* in Figure 5.8(a)-5.8(b) and Figure 5.9(a)-5.9(b). Reusability of data elements is low for non-greedy selection, since few partial matches have overlapping payloads. For greedy selection, in turn, adding a cache reduces latencies by at least two orders of magnitude, see *BL1*, *BL2* and *BL3* in Figure 5.8(c)-5.8(d) and Figure 5.9(c)-5.9(d). In such settings, LRU outperforms cost cache for *BL2* due to small computation overhead.

By contrast, when combining a cache with PFetch or LzEval, the cost-based policy achieves better performance. For instance, LzEval with a cost-based cache (Figure 5.8(a)) outperforms its counterpart with LRU (Figure 5.8(b)), especially for the median latency. While both employ the same LzEval procedure, the LRU policy ignores utilities, so that promising data elements may be fetched, but not kept in cache for sufficient time. This same trend is observed for PFetch.



(a) Cost-based cache. (b) LRU cache.

Figure 5.12: Throughput for Q5 under non-greedy selection.

We also observe that for DS4 (Zipf-distributed attributes), under greedy selection, different fetching approaches combined with an LRU cache performs better than the cost-based cache. This is because for such highly skewed value distribution, remote data access patterns show more locality. In this situation, a small fraction of remote data elements are frequently accessed, which is well served by an LRU cache. Cost-based cache, on the other hand, is still able to exploit such locality, but incurs big overhead to compute utilities compared to simple LRU eviction policy.

Benefits of Hybrid. When comparing the performance among PFetch, LzEval and Hybrid, Hybrid is shown to always outperform the others. The reason is that the benefits of PFetch and LzEval are complementary, whereas their combination does not come with any side effects and actually mitigates some of the strategies' downsides.

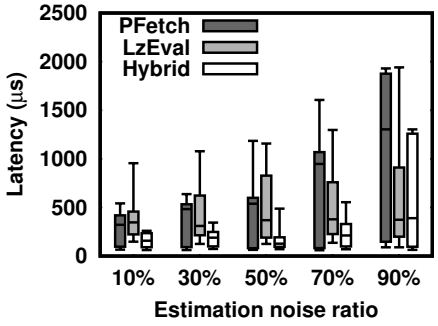
The performance of PFetch is closely related to the quality of the prefetching predictions that are made (based on the estimated future utility of data elements). Therefore, when it fails to prefetch a data element, PFetch still interrupts query evaluation and starts fetching on demand, resulting in high tail latencies (95^{th} percentile), see Figure 5.8(d). LzEval, in turn, generally has lower median and 75^{th} percentile latencies (Figure 5.8(a)-5.8(b)) because its fetching decision is always accurate. Yet, the resulting additional partial matches affect the latency under greedy selection, see Figure 5.8(c)-5.8(d) and Figure 5.9(c)-5.9(d).

Hybrid combines their advantages and alleviates the aforementioned side effects. If the PFetch part of Hybrid fails to prepare the correct data beforehand due to an inaccurate prediction, it still performs lazy evaluation, thereby avoiding the problem of blocking query processing for PFetch. The latency caused by the overhead of additional partial matches is also significantly reduced, compared to LzEval alone, since a lot of matches are already handled by prefetching.

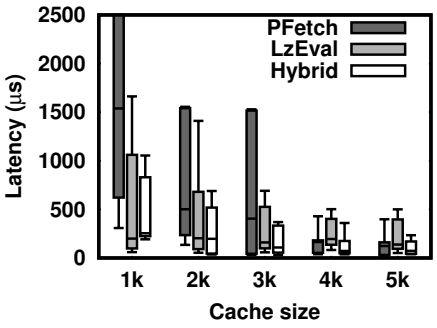
Throughput. We also investigate the throughput performance. Without loss of generality, we show the throughput for Q5 under non-greedy selection with either policy for cache management in Figure 5.12. Throughput performance is largely in line with the observed latencies, with a few deviations. For instance, while LzEval with a cost-based cache has a lower median latency

compared to LzEval with an LRU cache, the observed throughput is virtually equivalent.

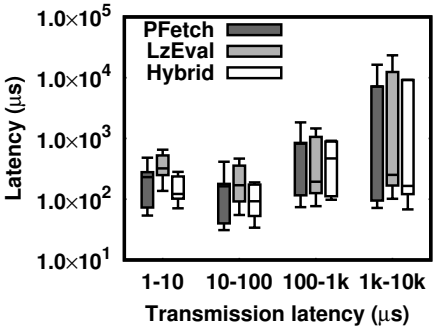
5.6.3 Sensitivity Analysis



(a) Sensitivity of utility estimation.



(b) Sensitivity of cache size.



(c) Sensitivity of transmission latency.

Figure 5.13: Sensitivity analysis for utility estimation, cache size and remote data transmission latency.

We assess the sensitivity of the proposed strategies concerning the utility estimation quality, the cache size, the remote data transmission latency, and the weighing factor ω which tunes the share of urgent utility and future utility in Equation 5.3. All results were obtained for query Q5, with a cost-based cache and greedy selection.

Utility estimation quality. The utility model lays the foundation for remote data fetching and cache management, so that its quality affects the efficacy of all strategies. We assess the impact of the estimation quality by injecting noise into the employed estimations, where a noisy estimation means that an expected partial match will not actually materialise. We compare the corresponding latency variations by injecting noise into 10% to 90% of the estimations.

Figure 5.13(a) illustrates that PFetch is sensitive to such noise, since its median latency grows significantly for higher noise levels. The reason is twofold: Latency increases because PFetch prefetches the wrong data elements, whereas poor utility estimation also negatively impacts the

elements that are evicted from the cache. Above a 50% noise ratio, its median latency already exceeds LzEval's 75^{th} percentile and Hybrid's 95^{th} percentile.

LzEval is less sensitive to noise, even obtaining stable latencies across noise levels for the 5^{th} and 25^{th} percentile, as well as the median. This is because LzEval's initial decision about what to fetch is not affected by utility, unlike for PFetch. Still, low quality utility estimations can lead to poor decisions on which partial match evaluations to postpone and, thus, to additional partial match growth and associated overhead. Again, the cache management also deteriorates for higher noise levels. As a result, especially LzEval's 95^{th} percentile latency grows along with an increasing noise ratio.

Since Hybrid combines the advantages of both PFetch and LzEval, it generally outperforms the individual strategies. However, an exception is observed at the 90% noise ratio, where Hybrid's 75^{th} percentile latency is higher than LzEval's. Given such inaccurate utility estimations, the downsides of PFetch outweigh its benefits, resulting in a better latency for LzEval than Hybrid.

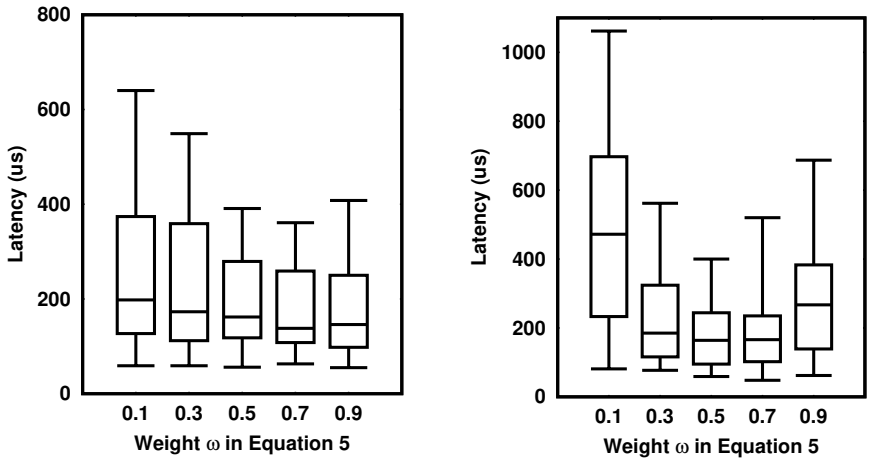
Cache size. The size of the employed cache naturally affects all fetching strategies, where a larger cache is beneficial, see Figure 5.13(b). In these results, we also observe that PFetch is more sensitive than the other strategies. This is because a larger cache allows for more tolerance in terms of incorrectly prefetched data elements, whereas a smaller cache will be clogged by them.

Remote data transmission latency. We consider that for higher transmission latencies, failing to fetch a data element leads to longer delays. We tested this aspect by evaluating the performance for different transmission latencies. The transmission latency is uniformly distributed in four different ranges: 1-10μs, 10-100μs, 100-1,000μs and 1,000-10,000μs. As shown in Figure 5.13(c), the overall processing latency of all strategies increases along with the transmission latency. However, PFetch is again most sensitive here. This is because prefetching needs to occur earlier for increased transmission latencies, which results in less accurate prefetch decisions.

Weighting factor in utility definition. We consider the impact of the weighting factor ω, which balances the urgent and future utility in our model (Section 5.3.1, Equation 5.3). We consider the impact of ω in the utility to guide the fetching of remote data (Section 5.4.1) and to manage the cache (Section 5.5). We refer to these, respectively, as ω_{fetch} and ω_{cache}.

Figure 5.14(a) shows the results obtained when varying ω_{fetch}, while fixing ω_{cache} at 0.5. Here, increasing the weight reduces the 75^{th} and 95^{th} percentile latencies. This is expected, as a low weight results in the current demand for remote data (urgent utility) being largely ignored. Yet, beyond $\omega_{fetch} = 0.7$, performance starts to deteriorate. For such high weights, decisions are strongly based on the current demand for remote data, but ignore the future demand. As such, we observe optimal results for $\omega_{fetch} = 0.7$.

Yet, utility modelling turns out to be robust: For any factor that emphasizes the urgent demand, but does not ignore the future usage of remote data, *i.e.*, $\omega_{fetch} \in [0.5, 0.9]$, the performance

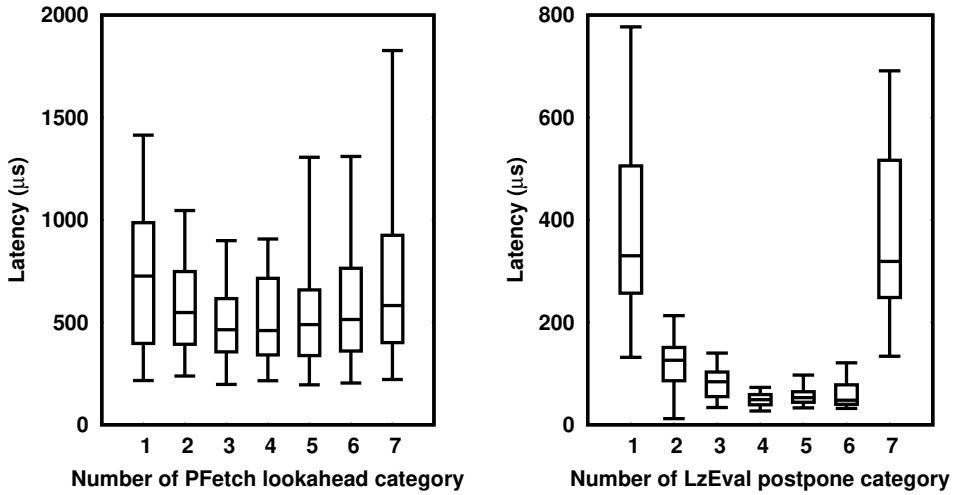


(a) Fetch utility (ω_{fetch}). (b) Cache utility (ω_{cache}).

Figure 5.14: Sensitivity analysis for utility weighting factor.

is close to the optimal, with especially stable median latencies.

With ω_{fetch} set to 0.7, varying ω_{cache} shows a similar trend, as shown in Figure 5.14(b). Here, the optimal value turns out to be 0.5, which indicates that cache management considers the current demand and the future usage of remote data equally. Values in the range $[0.3, 0.7]$ achieve comparable performance, though, which again points to a certain robustness of our utility model.



(a) PFetch lookahead category selection. (b) LzEval postpone category selection.

Figure 5.15: Sensitivity analysis for PFetch and LzEval category selection.

Selection of lookahead categories for PFetch. PFetch's performance largely depends on when to trigger prefetching – lookahead categories of partial matches. We demonstrate this aspect by manually selecting lookahead categories, how early prefetching occurs, rather than

determining it dynamically based on the strategies in Section 5.4.1. To this end, we modify Q5 by moving the remote data evaluation to the end of the pattern sequence, providing up to seven preceding categories for prefetching. As shown in Figure 5.15(a), the optimal performance is achieved when prefetching three preceding categories in advance. These results demonstrate the intuition that when prefetching occurs too late (*i.e.*, less than two preceding categories), it only hides a fraction of the transmission latency. In contrast, prefetching earlier than three preceding categories means that the prediction of which elements to fetch is inaccurate, compromising PFetch's benefits.

Selection of categories of partial matches for LzEval. We also test the impact of the selection of categories of partial matches for which LzEval is applied. We again modify Q5, setting the remote data evaluation categories to the beginning of the pattern sequence, allowing us to control the number of its succeeding categories (candidates for LzEval) between one and seven. It should be noted that changing remote data evaluation point actually results in different queries and the partial match distribution and latencies are changed with it. The results are demonstrated in Figure 5.15(b). The optimal results are achieved when selecting four succeeding categories for lazy evaluation, though the performance between selecting three and six succeeding categories is comparable. However, lazy evaluation for only one or two succeeding categories hides little latency, whereas the exponential growth in the number of additional partial matches becomes too large when selecting seven of them. There, the additional overhead mitigates the potential benefit of lazy evaluation.

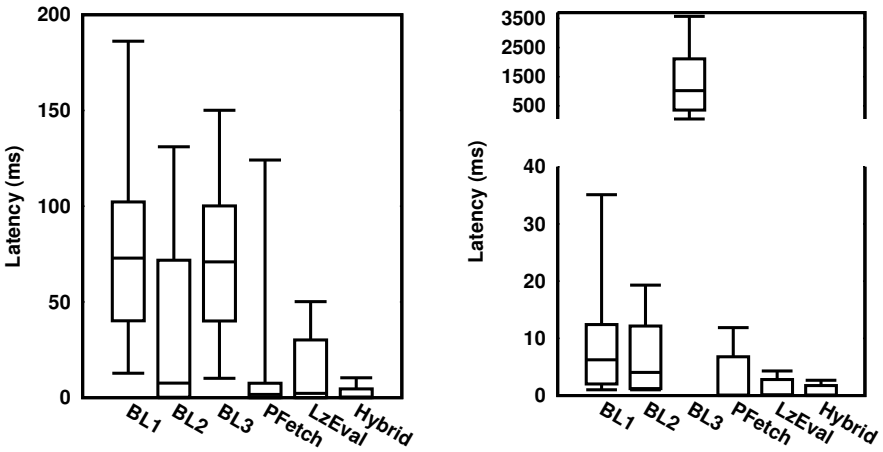
5.6.4 Case Studies

Finally, we applied our approach in the aforementioned real-world scenarios, comparing our fetching strategies against the baselines, with a cost-based cache under greedy selection.

```
PATTERN SEQ(Satellite a, Satellite b, Satellite c)+)
WHERE SAME[OVERLAP(boundary)] AND a.channel=7 AND a.time>=6:00
AND a.level=high AND b.channel=14 AND UDF1[a.level,b.level]=high
AND c.channel=16  AND c.time<18:00 AND REMOTE[c.temperature]>=46.51°C
AND c.level=high AND REMOTE[c.humidity]<=23.38
WITHIN 1h
```

Listing 5.2: Query for the satellite/sensor-network dataset.

Bushfire detection. We use a real-world dataset of a bushfire detection system composed of ground-based sensor networks and the geostationary operational environmental satellite, GOES-16 [136]. GOES-16 captures the earth's radiance in 16 spectral channels with different wavelengths and is capable of detecting heat signatures produced by fires [137]. While it may track fires in real-time, the obtained information is combined with data from a sensor network for fine-grained validation. For this dataset, Listing 5.2 shows a query to detect a bushfire in the



(a) Bushfire detection.

(b) Cluster monitoring.

Figure 5.16: Case studies.

```
PATTERN SEQ(Submit a, Schedule b, Evict c, Schedule d, Evict e, Schedule f,
    Fail g)
WHERE [task_id] AND b.machine=c.machine
AND b.region NOT IN REMOTE[d.machine]
AND d.machine=e.machine AND f.machine=g.machine
AND d.region NOT IN REMOTE[f.machine]
WITHIN 1h
```

Listing 5.3: Query for the Google Cluster Traces dataset.

daytime [137]. In essence, it describes the repeated occurrence of a specific radiation pattern for a geographical area (`boundary`) in the satellite data: Within one hour, the energy level of channel 7 is high, mid-wave IR bands (channel 7) signals are stronger than long wave IR bands (channel 14), and the CO_2 (channel 16) level is above a threshold. This area is probably on fire, which is further validated by remote data reference fetched from ground sensor networks. If its temperature is above 46.51°C and its humidify measure is above 23.38, the ESP engine reports a bushfire detection. We configure the data transmission latency uniformly distributed from 10ms to 100ms.

As shown in Figure 5.16(a), the results obtained for this case follow the trend observed for the synthetic dataset, with all our approaches outperforming the baselines and Hybrid performing best. Hybrid reduces median latencies for *BL1* and *BL2* by 206× and 20×, respectively. For the 95^{th} percentile latencies, the improvements are 18× and 13×. We also observe that PFetch has similar performance as Hybrid, except for the long tail, 95^{th} percentile latency, which shows that PFetch is able to accurately anticipate the need for remote data.

Cluster monitoring. As a second real-world dataset, we rely on the Google Cluster Traces [160]. It contains events that indicate the lifecycle (*e.g.*, submit, schedule, evict, and fail) of tasks

running in a large-scale cluster of 12.5k machines. We employ the query of Listing 5.3, which detects the following pattern: A task is submitted, scheduled, and evicted on one machine; later, in a different region, it is rescheduled and evicted on another machine; and finally it is rescheduled on a third machine, in another region, but fails execution. Here, information on the regions needs to be fetched from a remote database. we adopted a data transmission latency drawn uniformly from 1ms to 10ms. The results in Figure 5.16(b) again highlight that Hybrid outperforms all other approaches. It reduces the median latencies by 73× and 47× for *BL1* and *BL2*, respectively. For the 95^{th} percentile latencies, Hybrid improves 13× and 7×.

5.7 Summary

In this chapter, we proposed EIRES, a framework for efficient integration of remote data in evaluating patter detection queries over event streams. Our core idea is to decouple the fetching of remote data and its use in query evaluation. Data elements may be fetched before the need for them materialises and the evaluation of partial matches may be postponed until after the data is available. The EIRES framework facilitates these ideas through a cost model to evaluate data utility; strategies for fetching remote data, either through prefetching or lazy evaluation; and policies for cache management. Our experimental results show that EIRES improves the latency of query evaluation by up to 3,752× for synthetic data and 47× for real-world data.

CHAPTER 6

CASE STUDY: DEMAND RESPONSE MANAGEMENT IN SMART GRIDS

This chapter illustrates the usefulness of the event pattern detection and the optimisation techniques presented in Chapter 4 and Chapter 5 in a real-world scenario: We present a complete use case of applying efficient event pattern detection in demand-response (DR) management in smart grid systems. DR management aims to shift consumers' peak-hour energy demand load to off peak times by offering them financial incentives, in order to use energy in an efficient, economic, and environmentally friendly manner. However, traditional DR management approaches are static and do not react to users' behaviour in terms of peak energy reduction. Therefore, such approaches may fail to fully exploit the flexibility of users' energy use patterns and therefore, fail to meet the peak energy reduction goal. To tackle this problem, this chapter employs event stream processing to monitor the users' behaviour in response to the peak energy reduction requests and predicts potential non-compliance in real time. Based thereon, a second DR program is scheduled with higher incentives to compensate for the deficit in energy reduction.

This chapter is organised as follows: Section 6.1 explains the foundations of DR management in smart grid systems and illustrates the limitations of the state-of-the-art DR management approaches. Section 6.2 discusses how ESP for pattern detection enables adaptive DR management to achieve best-effort peak energy reduction. Evaluations are conducted in Section 6.3 to illustrate the effectiveness and efficiency of the proposed approach.

6.1 Problem Illustration

A smart electrical grid (smart grid) strives to improve efficiency, flexibility, and stability of the electric energy generation and distribution system. It offers energy-related services to the power grid operator and consumers. This is achieved by augmenting a traditional power grid with a

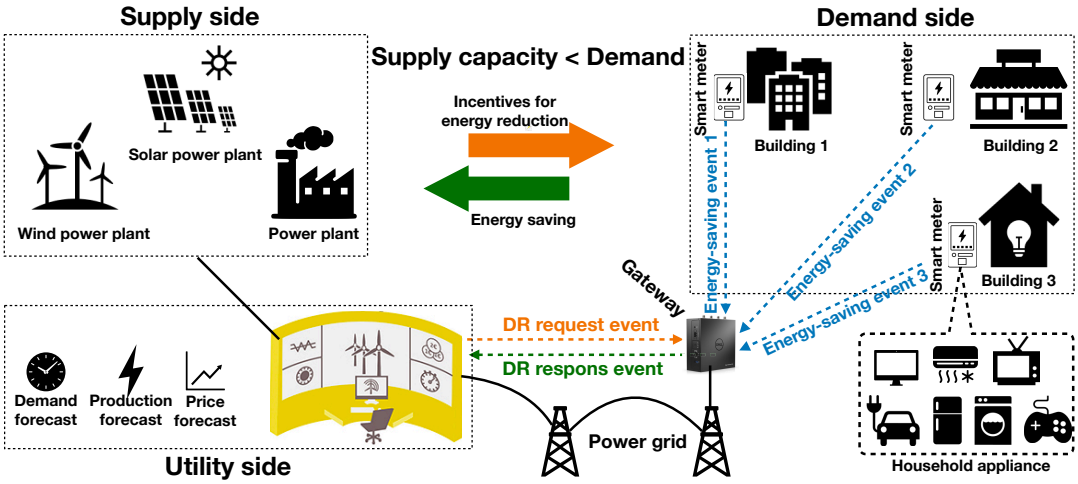


Figure 6.1: Demand response management in smart grid.

layer of information and communication technologies (ICT) to automatically collect, analyse, and act on meter data. Such an ICT layer is usually materialised by routers, gateways, and smart meters installed at users' premises that enable two-way communication between the power utility and users.

Figure 6.1 illustrates a typical framework of a smart grid that consists of three major (conceptual) components, supply side, utility side, and demand side, connected by a power grid. The supply side *produces* electricity from different sources including fossil-fuel-based power plants and renewable energy, such as solar farms and wind farms. The demand side, in turn, *consumes* electricity through the smart grid. It consists of households with smart meters installed. A smart meter monitors the energy consumption of household appliances, such as televisions, air conditioning, washing machines, and refrigerators. Furthermore, smart meters are able to switch on and off appliances if permission is granted by users. Lastly, the utility side matches the supply and the demand sides. Specifically, it analyses collected meter data from the smart grid and, based thereon, forecasts the capacities of both demand and supply side as well as the electricity price. The ideal scenario is that demand capacity and supply capacity are roughly the same at any point in time.

However, in practice, the demand side may require more energy than is available at the supply side during peak time. For instance, during 6 PM-8 PM, people return home from work and switch on a range of household appliances, resulting in spikes of energy demand beyond the capacity of the supply side. This mismatch could be solved by increasing the capacity of the supply side, *e.g.*, by running more generators at power plants. Alternatively, a more energy-efficient and economical approach is to shift the energy demand spikes to off-peak times, so that the reduced peak demand is below the supply side's capacity, as illustrated in Figure 6.2. Such load shift is achieved by *demand response (DR) management* that offers households financial incentives to

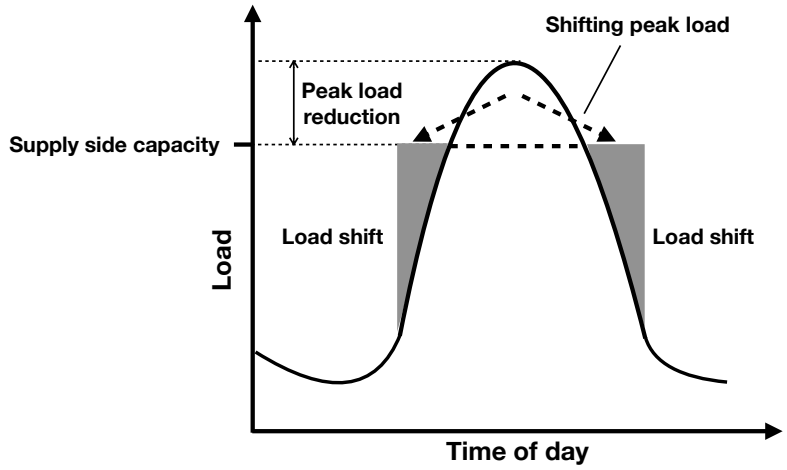


Figure 6.2: Peak load shift.

postpone or bring forward the use of appliances.

Figure 6.1 illustrates how DR management is supported by the ICT layer of a smart grid. Here, energy reduction requests and incentives are incorporated in *DR request events* (coloured in orange) sent by the utility side. These DR request events are transmitted to gateways of residential neighbourhoods and then forwarded to smart meters installed at households. Some smart meters may have permit (granted by users) to directly control the energy consumption of certain appliances. Others send notifications to users who may be willing to *postpone* unnecessary appliance usage such as washing machines, or switch on an air conditioner *in advance* to cool down a room before returning home. As long as users make responses, smart meters monitor energy savings and send them as *energy-saving events* (coloured in blue) to gateways which further aggregate energy saving measurements into *DR response events* (coloured in green) and send them back to the utility side for billing and further scheduling analysis. Such a two-way DR request-response process is called a *DR program*. A DR program lasts for the (estimated) peak time duration and consists of the following steps.

Step 1: The utility generates a DR program for a set of participants and sends DR request events with the volume of energy reduction, and the corresponding time frame $[t_{start}, t_{end}]$).

Step 2: Consumers respond by either accepting, or declining the DR request [99]. The consumers' responses are collected by the utility to estimate the available system flexibility.

Step 3: During the DR program period, the consumers who accept the request change their appliance usage patterns to meet the specified energy reduction.

Step 4: After the DR program ends, the utility evaluates the participants' contributions, and compensates them accordingly.

The problem of traditional demand response management approaches is that they are static, and do not involve feedback to users during the DR program. Specifically, as shown in Figure 6.3,

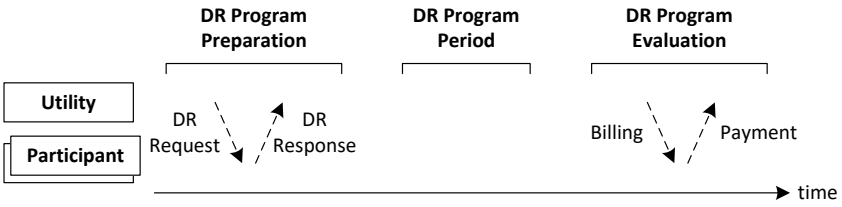


Figure 6.3: Traditional approach for DR management.

the utility informs the DR participants of an upcoming DR program while specifying the timing, required energy reduction, and incentive for compliance. During the DR program, the utility does not monitor the performance of DR participants while some users may decline to respond because the financial incentives are insufficient. Therefore, the overall energy saving during the DR program may not meet the reduction goal. However, traditional DR management approaches only recognise this non-compliance *after* the DR program ends and collect meter data for post-analysis to predict proper incentives for the next DR program at a different day. This is not economically optimal due to the under-utilization of the residential demand flexibility.

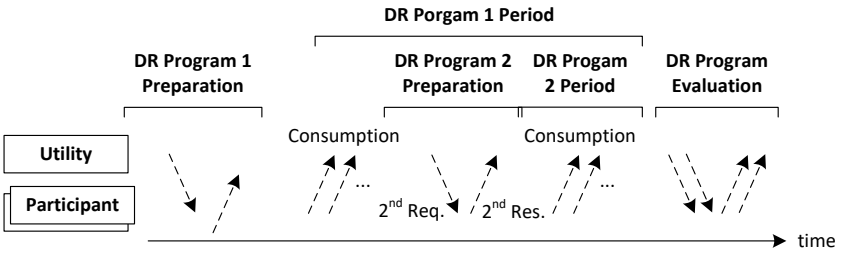


Figure 6.4: Adaptive DR management via event stream processing.

To address this challenge, it was recently suggested to leverage real-time feedback to adaptively modify the participants' incentives [158]. This chapter takes up this idea and proposes a distributed ESP framework to detect users' responses in real-time and predict potential non-compliance *before* the initial DR program ends, so that multiple DR request events may be scheduled with modified incentives, as demonstrated in Figure 6.4.

6.2 Adaptive DR Management with Event Stream Processing

This section shows how to realise the adaptive DR management through event stream processing thereby, enabling the predication of DR non-compliance with low latency. To this end, we formally define the problem of DR compliance assessment in Section 6.2.1, including an approach to predict user-level DR non-compliance before the DR program ends. Section 6.2.2 employs scalable distributed event stream processing to predict DR non-compliance at utility level with low latency. Section 6.2.3 explains how the utility decides the energy reduction, the incentive, and participants to schedule the second DR request, if a non-compliance is predicted.

6.2.1 DR Compliance Assessment and Prediction

We first define the problem of demand response compliance. A smart grid is fully compliant with a DR program if its users' accumulated energy saving is greater or equal than the energy reduction specified by the DR request event. It is formally defined as follows.

Definition 6.1 (DR compliance at utility level)**.** Let K be the number of smart grid users, t_{start} be the starting time of a DR program, t_{end} be the ending time, P_t be the users' actual power demand at time point t with the DR program, B_t be the users' power demand at time point t without the DR program, and λ be the specified energy reduction. Then, the smart grid is compliant with the DR program at utility level if the following condition is satisfied:

$$\sum_{t=t_{start}}^{t_{end}} (B_t - P_t) \geq \lambda. \tag{6.1}$$

Note that B_t is actually the *hypothetical* power demand of all users had there been no DR program. It can be estimated from users' historical load profiles (see [88] for an overview).

Problem 6.1. *The problem of adaptive DR management is to (1) predicate the earliest time point t, $t_{begin} < t < t_{end}$, when Equation 6.1 cannot hold and (2) modify incentives and schedule follow-up DR requests such that Equation 6.1 will be satisfied at time t_{end}.*

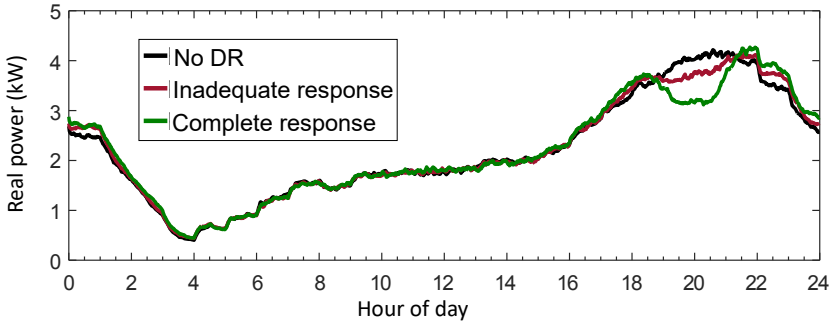
To address the problems of traditional DR management (Section 6.1) and realise the idea of adaptive DR management (Figure 6.4), we first consider the compliance assessment of each individual user with low latency. Based thereon, we propose a bottom-up approach to predict non-compliance at utility level before the DR program ends.

Similar to Definition 6.1, we define the DR compliance for a user (household k as follows.

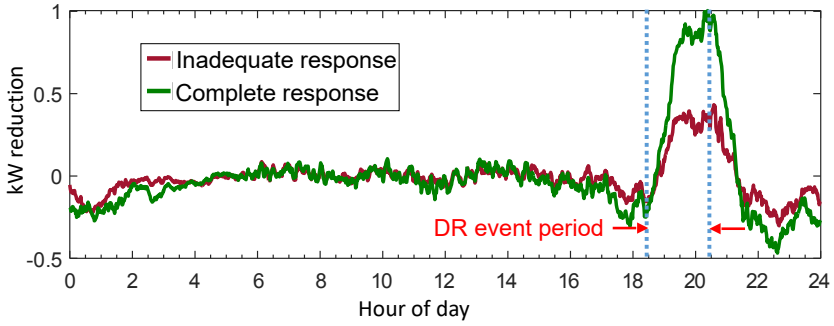
Definition 6.2 (DR compliance at user level)**.** Let t_{start} be the starting time of a DR program, t_{end} be the ending time, $P_{k,t}$ be the user's actual power demand at time point t with the DR program, $B_{k,t}$ be the users' power demand at time point t without the DR program, and λ_k be the specified energy reduction. user k is compliant with DR program if the following condition is satisfied:

$$\sum_{t=t_{start}}^{t_{end}} (B_{k,t} - P_{k,t}) \geq \lambda_k. \tag{6.2}$$

To illustrate the intuition behind the prediction of non-compliance at low latency, we consider the average load profile of a typical user shown in Figure 6.5(a), which is obtained based on the probabilistic generation techniques validated in [53, 157]. Here, a DR program is scheduled from 6:30 PM to 8:30 PM. The black curve corresponds to the case when no DR event exists. The green curve, in turn, represents the user's typical response for a given incentive. However, given the same DR program, for some reason (*e.g.*, hot weather) on a different day, the user does not



(a) Residential load profiles.



(b) Power reduction during DR program.

Figure 6.5: Inadequate and complete demand responses of a residential consumer.

respond adequately, resulting in the load profile depicted in brown. The difference between the load reduction with complete and inadequate DR compliance is demonstrated in Figure 6.5(b). It indicates that unless the behaviour of the user changes partly through the DR program period, the utility can predict whether this user is able to fully meet the energy reduction goal by comparing the (estimated) complete (green) and measured (brown) profiles from the start of the DR program. Based on this observation, we now derive a non-compliance prediction procedure.

We examine energy reduction profiles (Figure 6.5(b)) at a finer granularity in Figure 6.6. The complete (green) and inadequate (brown) response curves indicate that if the actual response (in terms of kWh reduction achieved) is lower than the estimated complete response for some time span from the beginning of the DR program, it is very likely that the energy reduction at the end of the DR program would also be insufficient. This leads to the proposed non-compliance prediction procedure for user k, which is also based on the 'Baseline Type-I' M&V technique proposed by the North American Energy Standards Board (NAESB) [88]:

Step 1: Wait for a fixed time duration d_{wait} after the DR program begins for the pre-DR rebound to disappear (see Figure 6.6).

Step 2: Monitor the energy reduction at a constant interval δ at times $t_{test} = t_1, t_2, \cdots, t_{end}$ to

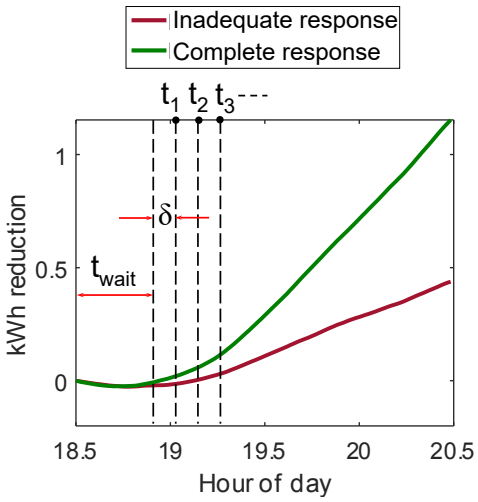


Figure 6.6: Profile of the cumulative energy reduction from the baseline during a DR event.

verify if the Equation 6.3 is satisfied for user k:

$$\sum_{t=t_{start}}^{t_{test}} (B_{k,t} - P_{k,t}) < \sum_{t=t_{start}}^{t_{test}} (B_{k,t} - \hat{P}_{k,t}). \tag{6.3}$$

Note that $[t_{start}, t_{end}]$ is the DR program period, $B_{k,t}$ the baseline demand without DR, $P_{k,t}$ the measured demand during the DR program, and $\hat{P}_{k,t}$ the load profile for the estimated complete response. $B_{k,t}$ and $\hat{P}_{k,t}$ are estimated using standard DR measurement and verification techniques based on historical performance data and randomised control trials [88]. Furthermore, Equation 6.3 can be simplified into the following Equation 6.4:

$$\sum_{t=t_{start}}^{t_{test}} P_{k,t} > \sum_{t=t_{start}}^{t_{test}} \hat{P}_{k,t} \, . \tag{6.4}$$

Step 3: Predict non-compliance if Equation 6.4 is satisfied for m consecutive testing times.

The above procedure is applied for at least m consecutive testing intervals before issuing a prediction (see step 3 above). The reason for step 3, testing at least m consecutive intervals, is because in reality, the measured and estimated load profiles would suffer from high variability as opposed to the average profiles presented in Figure 6.5. Therefore, repeated testing is performed to avoid false predictions of non-compliance. Values for the parameters, *i.e.*, t_{wait}, the testing interval δ, and m need to be tuned for a given system before accurate predictions can be obtained.

6.2.2 Scalable Monitoring through Event Stream Processing

To realise the adaptive DR management based on low-latency compliance assessment in a scalable manner, we employ a distributed event stream processing infrastructure as illustrated in Figure 6.7. That is, following the topology of a smart grid communication network (ICT layer), all users are divided into groups to enable decentralised prediction of their compliance during

a DR program. The compliance predictions for individual users are used to predict group-level compliance which is in turn used for compliance prediction at the global level for the utility. Based thereon, decisions on additional DR requests are taken.

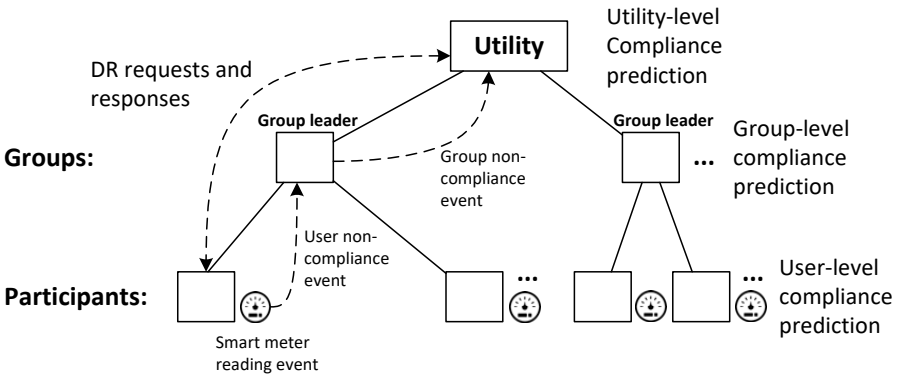


Figure 6.7: Infrastructure for dynamic DR based on distributed event stream processing.

The proposed group-based approach to monitoring means that the approach is inherently distributed. Conceptually, non-compliance predictions are performed at user level, group level and utility level. As detailed below, the accumulation of energy measurements and the compliance prediction is further grounded in queries over event streams to meet the requirement of traceable and online processing.

User-level non-compliance prediction. A Smart meter monitors and accumulates energy consumption of household appliances and digitalise such information through smart meter reading events of type `smartMeterEvent`. Table 6.1 shows the schema of the `smartMeterEvent` type. Note that we only list the relevant attributes to DR compliance assessment. Based thereon, a smart meter performs the three-step-procedure of user-level DR compliance prediction explained in Section 6.2.1. This is achieved by evaluating the query defined in Listing 6.1. It detects a sequential pattern with a Kleene closure operator that selects smart meter reading events for users whose accumulated load satisfies Equation 6.4 for at least m_{user} consecutive time intervals per sliding window. Note that `a[i]` refers to each selected event of the Kleene closure sequence. Also, λ`[a[i].housID,a[i].time]` represents the estimated complete response load for a given user at a given time point ($\hat{P}_{k,t}$ in Equation 6.4). The query's output event is a user-level non-compliance event of type `userNonComplianceEvent`, which contains the timestamp when non-compliance is predicted, as well as the household ID, the group ID, and the accumulated load.

Group-level non-compliance prediction. A *group leader* is assigned to each group and group members' smart meters continuously send events of type `userNonComplianceEvent` to the group leader. A group leader may be realised by a gateway device as illustrated in Figure 6.1, or by a selected smart meter that analyses other smart meters' reading events within the group. Similar to the prediction procedure at user level (Section 6.2.1), a group is not compliant to DR management if more than ratio γ_{user} of its residents are predicted to be non-compliant for at

Table 6.1: Schema of smart meter reading event `smartMeterEvent`.

Attribute:	id	time	houseID	groupID	accLoad (kW)
Domain:	$\mathbb{N}$	$\mathbb{N}$	*text*	*text*	$\mathbb{R}$
	11	24234982	'H1'	'G1'	12.20
	12	24235323	'H2'	'G2'	14.78
	23	24236728	'H1'	'G1'	24.41

```
SEQ (smartMeterEvent a+[])
WHERE SAME[houseID] AND a[i].time>t_start AND a[i].time<t_end
AND a[i].accLoad>λ[a[i].housID,a[i].time]    //test Equation 6.4
AND a[i]time+1=a[i+1].time     //test consecutive time points
AND LENGTH(a+[])>m_user    //test more than m consecutive time points
WITHIN 2h
RETURN userNonComplianceEvent(a[m_user].time, a[m_user].houseID,
                              a[m_user].groupID, a[m_user].accLoad)
```

Listing 6.1: Query for individual level compliance assessment.

least m_{group} consecutive time intervals. This is achieved by evaluating queries in Listing 6.2 at group leaders. Here, the top query selects groups whose ratio γ_{user} of users fail to comply DR request event's energy reduction and generate an alert event of type `groupAlertEvent`, grouped by time stamps. An array, `GroupSize`, stores the number of residents for each group which is indexed by group ID. The bottom query, in turn, takes alert events of type `groupAlertEvent` as the input stream and detects the pattern that alert events happen consecutively for at least m_{group} time intervals. It then generates group non-compliant events of type `groupNonComplianceEvent` as an output stream.

```
SELECT COUNT(DISTINCT houseID) AS alertNum, SUM(accLoad) AS sumAccLoad
FROM userNonComplianceEvent STREAM
WHERE SAME[groupID] AND alertNum > γ_user×GroupSize[groupID]
GROUP BY time
RETURN groupAlertEvent(time, groupID,sumAccLoad)

SEQ (groupAlertEvent a+[])
FROM groupAlertEvent STREAM
WHERE SAME[groupID] AND a[i].time>t_start AND a[i].time<t_end
AND a[i]time+1=a[i+1].time     //test consecutive time points
AND LENGTH(a+[])>m_group    //test more than M consecutive time points
WITHIN 2h
RETURN groupNonComplianceEvent(a[m_group].time, a[m_group].groupID,
                               a[m_group].sumAccLoad)
```

Listing 6.2: Queries for group level compliance assessment.

Utility-level non-compliance prediction. The utility keeps receiving event streams of type `groupNonComplianceEvent` from group leaders. Non-compliant prediction at utility level is based on

group-level non-compliance. Specifically, if more than ratio γ_{group} of groups fail to comply DR request energy reduction for more than at least $m_{utility}$ consecutive time intervals, the utility is predicted to be non-compliant at the end of DR program. This is realised by evaluating queries in Listing 6.3. Similar to group-level prediction, the top query in Listing 6.3 selects accumulated load and time intervals when more than γ_{group} groups fail to comply DR request and generate an alert event of type `utilityAlertEvent`, grouped by time stamp. The bottom query, in turn, takes events of type `utilityAlertEvent` as the input stream and detects the pattern that the alter events appears consecutively for at least $m_{utility}$ times. Finally, it generates non-compliance event at utility level of type `utilityNonComplianceEvent`, which triggers the schedule of a second DR program.

```
SELECT COUNT(DISTINCT groupID) AS alertNum, SUM(sumAccLoad) AS load
FROM groupNonComplianceEvent STREAM
WHERE alertNum > γ_group×GROUPNUM
GROUP BY time
RETURN utilityAlertEvent(time, load)

SEQ (utilityAlertEvent a+[])
FROM utilityAlertEvent STREAM
WHERE a[i].time>t_start AND a[i].time<t_end
AND a[i]time+1=a[i+1].time     //test consecutive time points
AND LENGTH(a+[])>m_utility      //test more than M consecutive time points
WITHIN 2h
RETURN utilityNonComplianceEvent(a[m_utility].time, a[m_utility].load)
```

Listing 6.3: Query for utility level compliance assessment.

In the proposed adaptive DR implementation, the monitoring rate at the various levels of the system is reduced once the compliance assessment is complete. For instance, consider the case where a group of DR participants is predicted to be non-compliant at time t_{nc} during the DR program. For this group, further compliance testing is not performed during the DR program period, and the monitoring rates for smart meters in this group are lowered to the rate required for settlement purposes. Therefore, even if the smart meter reading event streams were initially monitored at a higher rate, *e.g.*, at 1 min intervals, the rate can be reduced to 15 min intervals after non-compliance prediction. Meanwhile, the monitoring in other groups remains unaffected, and the utility-level monitoring queries continue to predict the smart grid system compliance status. Once the overall smart grid system is detected to be non-compliant, the monitoring rate for all the smart meters is reduced. This approach significantly reduces the monitoring cost and enables a scalable implementation for large systems, as will be demonstrated in Section 6.3.3.

6.2.3 Utility Intervention during an Unsuccessful DR Event

When the utility predicts a non-compliant response to an incentive λ_o, it offers a higher incentive λ_{new} for users to offset the deficit in the peak energy reduction. λ_{new} is incorporated in another

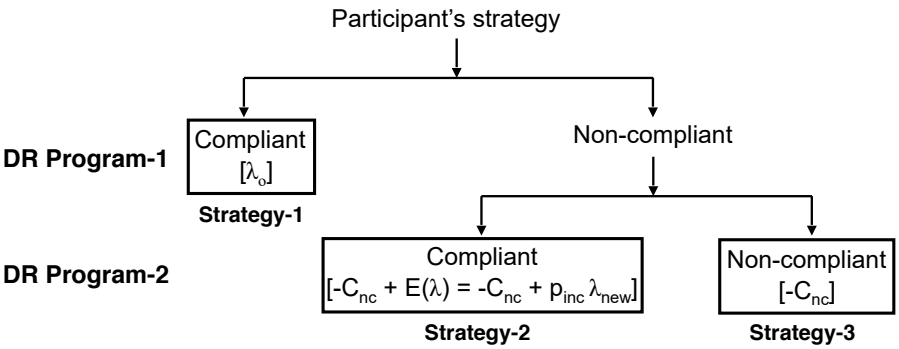


Figure 6.8: DR participants' expected payoffs (in square braces) under three possible strategies.

DR request event for the follow-up DR program (see Figure 6.4). However, a large number of DR programs in single day may result in the risk of participation fatigue [24], which deters users from participating in future DR programs. Such a number shall be tuned for different smart grids and here we focus on at most two DR requests.

6.2.3.1 Incentive for the Second DR Program

Determining the new incentive λ_{new} requires the utility to have some knowledge about the behavior of the consumers in the system, which is a reasonable expectation [142]. Without loss of generality, we assume two constraints to prevent over-compensation and potential gaming of the DR management by the participating users.

First, while the second DR request is offered to *all* participants, we use a fixed probability p_{inc} to randomly select a *subset* of the participants that are compliant in the second event, who are actually paid the additional incentive. Indeed, such a lottery-based reward scheme has been found to elicit more response than a fixed reward [99], while also reducing the overall cost of the second event to the utility. The value of p_{inc}, or equivalently, the number of participants $\kappa = (p_{inc} \times \text{total participants})$ actually offered the higher incentive is determined as follows:

$$\text{Monitoring cost} + \kappa\ (\lambda_{new} - \lambda_o)\ \leq\ C_{saved}. \tag{6.5}$$

Equation 6.5 compares the cost and benefit of the proposed scheme, with C_{saved} referring to the cost savings achieved by reducing the peak load through the second DR program. C_{saved} may also refer to the cost saving of using alternates such as direct load control to compensate the deficit in the first DR program.

Second, to encourage participants to provide compliant responses for the first DR program instead of waiting for higher incentives during the second one (*i.e.*, to game the DR program), a penalty C_{nc} is imposed on participants who are non-compliant in the first DR program. Figure 6.8 shows the expected payoffs under different possible scenarios. By setting:

$$C_{nc} > p_{inc}\ \lambda_{new} - \lambda_o\ , \tag{6.6}$$

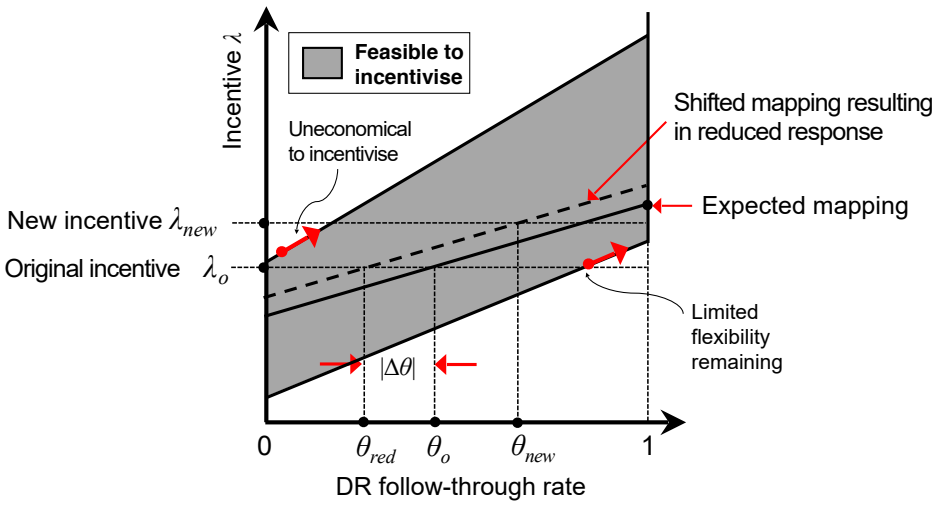


Figure 6.9: Mapping the incentive offered by the utility to the consumer response.

the utility achieves that the dominant strategy for each participant is to fully participate in the initial DR program (Strategy-1) and thereby maximise their expected payoffs.

6.2.3.2 Correlations between Additional Incentive and Consumer Response

The responses of the various participants to the second DR program are not uniform. On the one hand, users who respond to the first DR request event with high enthusiasm may not have additional flexibility left to provide, and therefore, any additional incentives may not result in meaningful returns. On the other hand, users who do not respond at all to the first DR program may be reluctant to respond during the second one as well, even for an increased incentive. These aspects are represented by the lower and upper bounds of the hatched region in Figure 6.9, considering a mapping between the incentive offered and the response to it (similar models have been widely adopted, *e.g.*, see [142, 177]). Here, the DR follow-through rate represents the probability of a user's response to a DR request. *i.e.*, a value of one means deferring the use of appliances (*e.g.*, dishwashers), while a value of zero indicates that the user takes no action to change the energy usage pattern. The second DR request event targets those users in the middle of the above spectrum, *i.e.*, those whose characteristic lies within the hatched region.

For instance, we may assume a linear correlation between incentives and follow-through rates. For a user, the first DR program offers incentive λ_o and the expected median follow-through rate is θ_{exp}. However, in practice, during the DR program, we monitor a shifted mapping which results in a reduced follow-through rate θ_{red}. If a utility-level non-compliance is predicted, the utility computes the new incentive λ_{new} based on the estimated follow-through rate θ_{new}, see Equation 6.5 and Equation 6.6. Note that the mappings are modelled based on a survey [156] and metrics reported from previous DR field trials [99].

6.3 Evaluations

To illustrate the effectiveness and efficiency of the adaptive DR management with event stream processing, we conduct a comprehensive case study of a (micro) smart grid with home appliances and electric vehicles (EV) participating in the DR program. Section 6.3.1 demonstrates the energy load profiles and case description. We show the effectiveness of the adaptive DR management in Section 6.3.2, followed by its efficiency in Section 6.3.3.

6.3.1 Experimental Setup and Case Description

Home appliance load profiles. The load profile of each residence's home appliances is simulated in a bottom-up fashion, using the model and specifications introduced and validated in [53, 157]. In brief, this approach generates load profiles based on the probabilities of starting an appliance at a given time of the day. If a DR program is scheduled by the utility, the appliance-use probabilities are modified to reflect users' actions in deferring their energy usage.

Residential EV charging load profiles. We simulate the EV profiles based on the approach presented in [154] when no DR event is requested by the utility. The behaviour of each EV user is modelled by the probability distribution of the number of charging procedures per day, probability that each charging begins at a given time of the day, and the probability distributions of the initial and final state-of-charge values for the EV battery. When an EV user participates in a DR program, the user defers the EV charging and this behaviour is modelled as a reduction of the probabilities mentioned above.

Figure 6.10 illustrates the resulting average load profiles of appliances in one residence and an EV while considering a DR event between 6:30-8:30 PM, with $\theta_{ft} = 0.3$ as an example.

Use case description. To illustrate the effectiveness of the adaptive DR management, we consider a (micro) smart grid system of 100,000 users divided in five groups. All the users have enrolled in the utility's incentive-based DR program. Each user is uniformly assigned from 20 to 25 different common house appliances including microwave oven, refrigerator, TV, coffee machine, dish washer, washing machine, among others. Besides, residents are randomly assigned EVs depending on the EV penetration level, 22.48%, which is the 2030 forecast for EV adoption in the UK. Users' EV response actions (follow-through rates) are modelled based a survey [156], where the average value is $\theta_{ft} = 0.45$. The follow-through rate of each household is assigned to a randomly-chosen survey response. A DR program is scheduled from 6:30 PM to 8:30 PM and the expected energy reduction is 83.28 MWh.

The load profiles are simulated in Matlab and the event stream processing infrastructure is implemented based on Esper [69]. The simulation parameters are listed in Table 6.2. Here, the parameters t_{wait}, δ, m, γ_{group} and m_{user}, m_{group} and $m_{utility}$ were selected empirically in order to accurately detect non-compliance for a 20% reduction in follow-through rates of consumers. Note

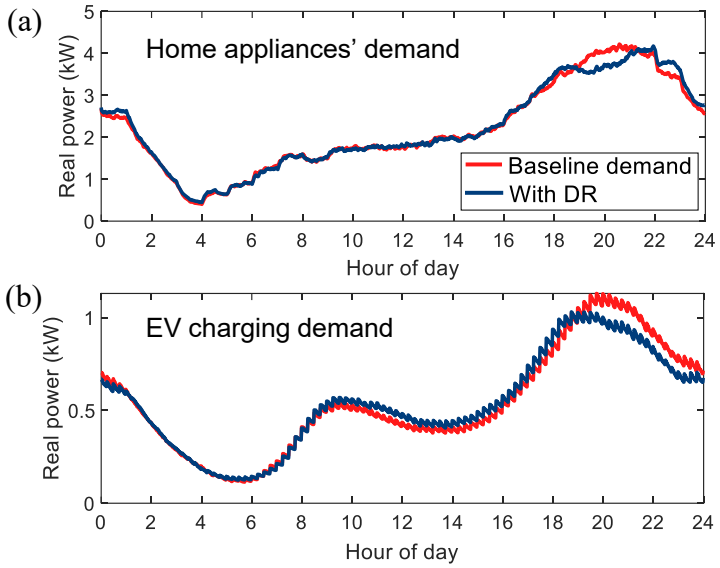


Figure 6.10: Average load profile for: (a) home appliances in one residence, and (b) charging one residential EV.

Table 6.2: Smart grid DR management simulation parameters.

	Parameter	Value
DR program	No. of DR participants	100,000
	Start time t_{start}	6:30 PM
	End time t_{end}	8:30 PM
	Energy reduction goal	83.28 MWh
Monitoring	No. of groups	5
	Waiting time t_{wait}	20 minutes
	Monitoring interval δ	5 minutes
	No. of consecutive user-level alert times m_{user}	3
	No. of consecutive group-level alert times m_{group}	3
	No. of consecutive utility-level alert times $m_{utility}$	3
	Utility-group non-compliance threshold ratio γ_{group}	0.4
Grouping	Group G1	consumer No.1-No.10,000
	Group G2	consumer No.10,001-No.30,000
	Group G3	consumer No.30,001-No.35,000
	Group G4	consumer No.35,001-No.60,000
	Group G5	consumer No.60,001-No.100,000

that, in practice, the participants' incentive-response correlation varies based on the location, demographics, and other factors. To capture such difference, we leverage four different linear mapping schemes (Section 6.2.3.2).

Measurement. We investigate effectiveness and efficiency of the proposed adaptive DR program. To this end, we measure the energy reduction during peak time of our approach and compare it to

the baseline (without DR), the expected reduction, and traditional single DR program approaches. Specifically, both baseline and expected reduction are estimated. In contrast, both traditional and proposed DR approaches are actually evaluated separately. Furthermore, we measure the energy reduction under different speeds of non-compliance detection. For efficiency, we measure the communication and computation cost of the proposed adaptive DR approach for different numbers of users, reaching up to 1.6 million.

6.3.2 Effectiveness of Adaptive Demand Response Approach

Peak energy reduction. The energy load profile of the baseline demand, of the expected (complete) response, of the traditional DR, and of the proposed adaptive DR are depicted in Figure 6.11. Recall that the expected peak energy reduction of the initial DR program is 83.28 MWh. However, in practice, participants mean follow-through rate, $\overline{\theta}_{red}$ is reduced by 40% compared the expected mean rate $\overline{\theta}_o$. Therefore, with no changes to the original incentive λ_o, the actual energy reduction by the traditional single DR program is only 47.89 MWh, see Figure 6.11. Note that, without loss of generality, λ_o is assigned an abstract unit for illustrative purposes.

In the adaptive DR approach, in turn, the utility analyses the real-time demand, and predicts that only two of the five groups are expected to be compliant at 7:11 PM. Given the threshold γ_{group} = 0.4, it then predicts that the overall system would be non-compliant, and schedules a second DR request event beginning ten minutes later with a new increased incentive λ_{new} for each user. λ_{new} is computed based on Equation 6.7, given constrains presented in Section 6.2.3.1.

$$\lambda_{new} = \lambda_o + \frac{(u-1)\lambda_o}{1-\overline{\theta}_o}\,(\beta\,\overline{\theta}_o - \overline{\theta}_{red}). \tag{6.7}$$

The users then respond to λ_{new} with the improved follow-through rate θ_{new} that is computed based on Equation 6.8.

$$\theta_{new} = \theta_{red} + \left(\frac{1-\theta_o}{1-\overline{\theta}_o}\right)(\beta\,\overline{\theta}_o - \overline{\theta}_{red}). \tag{6.8}$$

Here, the factor u is set as 1.05, 1.10, 1.15 and 1.20 for the four different linear mappings (see Section 6.3.1). The factor β, in turn, is to *overcompensate* the drop in the follow-through rates during the second DR program to mitigate the overall energy reduction deficit. We empirically tuned it to 1.8.

Finally, we observe that the energy reduction achieved by the proposed technique is 82.61 MWh, which is within 1% of the expected value of 83.28 MWh, thereby demonstrating the effectiveness of the proposed DR design.

Speed of non-compliance detection. The effectiveness of the proposed adaptive DR design depends on the speed of non-compliance detection. This is illustrated in Figure 6.12. While assuming the same DR program parameters as in Figure 6.11. we manually delay the non-compliance detection time at ten-minute intervals starting from the earliest time 7:11 PM and

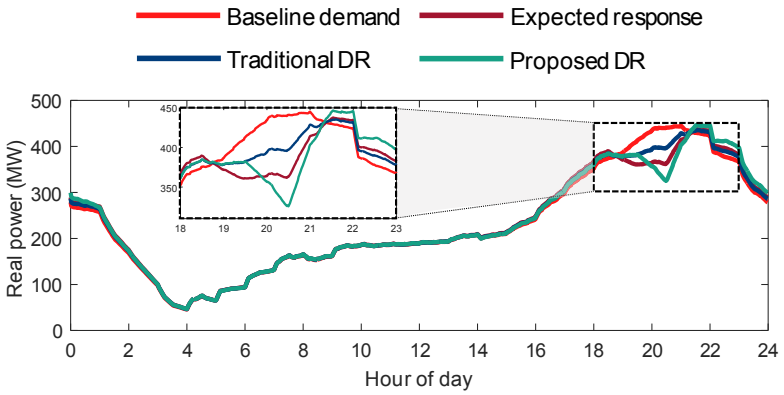


Figure 6.11: System demand for different baseline and DR approaches.

compare the final energy reduction achieved. Notably, the faster the detection of non-compliance, the larger the window to mitigate the deficit in response. Therefore, more peak energy reduction is achieved.

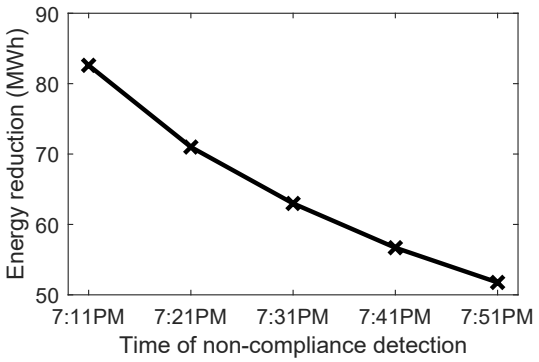
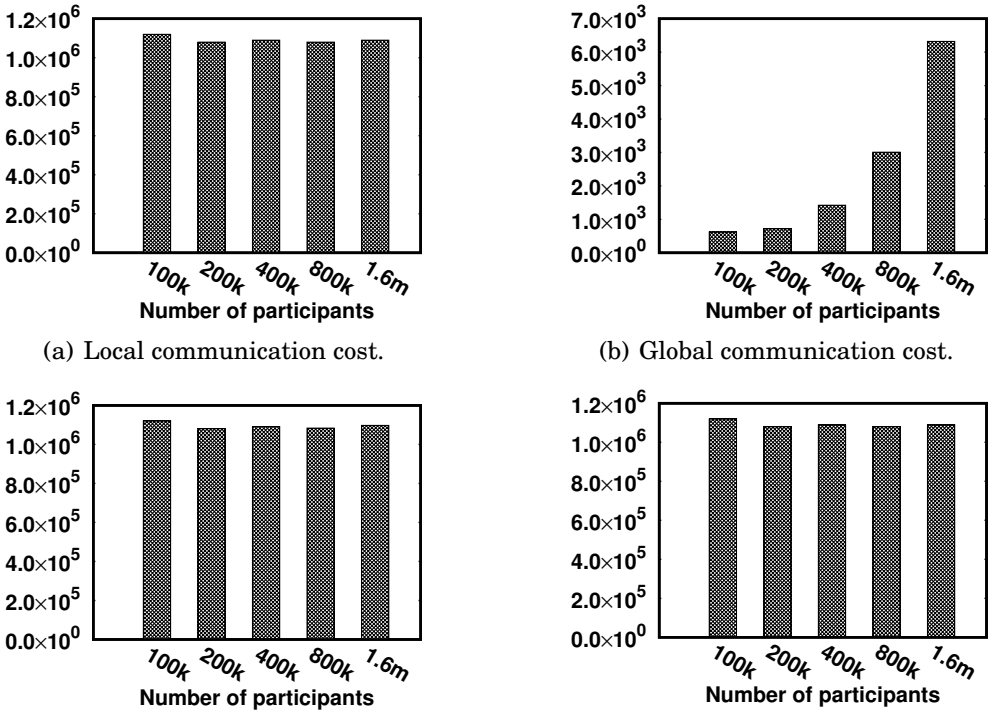


Figure 6.12: Impact of the speed of non-compliance detection to the DR performance.

6.3.3 Efficiency of Distributed Event Stream Processing

To illustrate the scalability of the proposed DR scheme, we explore the communication and computation costs involved in the monitoring process. Specifically, the communication cost is defined as the number of event transmissions (*e.g.*, events of type `smartMeterEvent` in Section 6.2.2), while the computation cost is defined as the number of events and partial matches processed by the ESP engine. We simulate a smart grid system where the number of DR participants increases from 100,000 to 1.6 million. The results are presented in Figure 6.13, assuming that every 10,000 consumers form one group. Evidently, the monitoring costs remain nearly constant as the system size increases. This is because in the proposed distributed monitoring system depicted in Figure 6.7, the monitoring of the participants is local within the group, allowing for parallel compliance assessments within the various groups. Global communication is only required between the group leaders and the utility. Therefore, scaling out the system actually



(a) Local communication cost.

(b) Global communication cost.

(c) Total communication cost.

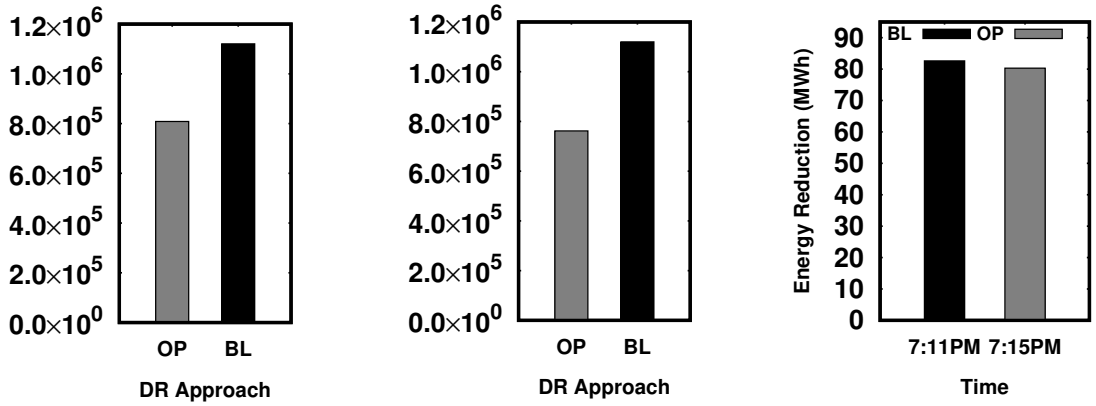
(d) Computation cost.

Figure 6.13: Communication and computation costs as the number of DR participants increases.

only adds groups, which increases the number of global communication massages by the number of newly-added groups. Furthermore, the resolution at which the smart meter event streams are monitored is reduced from one-minute intervals to 15-minute intervals after non-compliance detection, which further reduces the communication costs.

6.3.4 Hybrid Load Shedding and Remote Data Integration in DR Management

DR management serves as a supportive enhancement for smart grid scheduling rather than exact analysis (*e.g.*, billing purpose): The DR management delivers best-effort energy reduction during peak time. Yet, it does not guarantee to meet the required energy reduction goal, which is also not required by the utility. Therefore, lossy optimisation such as the hybrid load shedding proposed in Chapter 4 is particularly useful here. By discarding input events and partial matches to reduce the query processing latency on edge devices (*e.g.*, smart meters and gateways), hybrid load shedding enables the utility operator to maintain a similar DR management service quality with less powerful edge devices (*e.g.*, 100,000 smart meters with less computing power in this evaluation). Alternatively, the utility is also able to save computing power for other services while keeping the quality of DR management (*e.g.*, performing more complex data analysis for incentive prediction and billing purposes). We set latency bounds as 50% of the average latencies of the



(a) Communication cost. (b) Computation cost. (c) Energy reduction.

Figure 6.14: Comparison of DR management approaches with (denoted by **OP**) and without (denoted by **BL**) hybrid load shedding and EIRES framework.

original (without load shedding) execution of queries in Listing 6.1, 6.2, and 6.3, and investigate the performance of event stream processing and DR management.

To improve the quality of the baseline load profile estimation ($B_{k,t}$ in Equation 6.2), we incorporate the estimation as remote data sources maintained at the utility. The baseline load profiles are continuously updated based on the latest feedback from DR response monitoring according to [88], whereas smart meters and gateways fetch the corresponding load profiles when evaluating pattern detection queries in Listing 6.1, 6.2, and 6.3. The rationale behind this setting is that the utility has global monitoring information about the DR response performance of the whole smart grid, so that continuously updating the baseline load profiles improves the quality of the estimation. Here, the EIRES framework proposed in Chapter 5 is beneficial to reduce the data transmission latency.

In this evaluation, we simulated the execution of DR management on a single server. We applied both hybrid load shedding and the EIRES framework to the introduced approach for adaptive DR management, denoted as the optimised approach (OP). Using the relative measurements of performance, we compare the communication and computation costs as defined in Section 6.3.3 against the DR monitoring without hybrid load shedding and the EIRES framework, denoted as the baseline approach (BL). In addition, we also compare the effectiveness of DR management — the time of non-compliance prediction and the actual volume of energy reduction, using the same incentive scheme. We assume that similar relative performance improvement will be achieved on real-world edge devices deployed in smart grids. The results are illustrated in Figure 6.14.

From Figure 6.14(a) and Figure 6.14(b), we obverse that the optimised approach reduces 30% communication and computation cost compared to the baseline approach. The reason is that the hybrid load shedding discards input events and partial matches to reduce computation cost

thereby, generating less communication messages in the smart grid, which improves the efficiency. Meanwhile, Figure 6.14(c) illustrates that both approaches achieve similar DR performance. The baseline approach predicted the non-compliance at 7:11 PM and scheduled the second DR request accordingly, whereas, the optimised approach made the non-compliant predication and scheduled its second DR request only four minutes later, at 7:15 PM. At the end of the DR program, the baseline approach achieved the energy reduction of 82.61 MWh and the optimised approach reduced 80.30 MWh energy. This is because the hybrid load shedding discards unimportant input events and partial matches to maximise the results quality, achieving similar energy reduction with the baseline approach. In addition, the EIRES framework efficiently integrated baseline load profiles from the utility to smart meters and gateways, so that the optimised approach predicted the non-compliance and scheduled the second DR request only four minutes slower than the baseline approach, despite of side effects of the hybrid load shedding. In summary, the optimised approach is able to achieve similar DR performance using 70% of the computing power compared to the baseline approach.

6.4 Summary

This chapter shows how adaptive DR management can be realised using event stream processing. In the case of potential non-compliance to the energy reduction goal of the DR program, it adaptively creates an additional DR request with higher incentives for participation. This is implemented using a distributed event stream processing framework, which enables the scalable realisation with low computation and communication overheads. It does not require new hardware and can be integrated easily into existing DR management systems, and can be modified by utilities to suit their respective customer bases. The merits of the adaptive DR management were demonstrated for a system of 100,000 residential consumers, using bottom-up simulations of home appliances and electric vehicles, and realistic consumer behaviour models. We also illustrate that the hybrid load shedding and the EIRES framework enable the adaptive DR management to scale to 1.6 million users.

CHAPTER 7

CONCLUSION

This chapter concludes this dissertation. Section 7.1 provides a summary of the main results and contributions of this PhD work and discusses the implications for research and practice. Section 7.2 presents an outlook on directions for future research.

7.1 Summary and Impact

In this dissertation, we focused on the optimisation of state management for efficient pattern detection in event stream processing. As a basis for developing these optimisation techniques, we discussed why state management plays a pivotal role in efficient event pattern detection and general data stream processing, especially in the light of requirements for low-latency processing. We analysed that the query processing latency consists of the pattern evaluation latency and the remote data transmission latency. This dissertation's main contributions include a lossy optimisation technique to reduce the pattern evaluation latency, a holistic optimisation framework, EIRES, to reduce remote data transmission latency during query execution, and a comprehensive case study that employs the above techniques in event stream processing for smart grid management. We summarise the main results as follows:

- We proposed a hybrid load shedding approach to reduce pattern matching latency when detecting patterns over event streams. It enables best-effort query evaluation, striving for maximal query results quality while staying within a latency bound. Since the utility of an input event in a stream may be highly dynamic in the presence of different states, we complemented traditional input-based load shedding with a novel perspective: shedding of states—partial matches. We presented a cost model based on states to balance various shedding strategies and decide on what and how much data to shed.

- We proposed the EIRES framework for efficient integration of remote data in the evaluation of queries over event streams. Our core idea is to decouple the fetching of remote data and its use in query evaluation. Remote data elements may be fetched before the need for them materialises and the evaluation of corresponding query predicates may be postponed until after the remote data elements are available. The EIRES framework facilitates these ideas through a cost model to evaluate remote data utility based on states; strategies for fetching remote data, either through prefetching or lazy evaluation; and policies for cache management.
- We showed how to achieve adaptive DR management using event stream processing. The scheme creates an additional energy reduction request with higher incentives in potential non-compliance to achieve the original energy reduction goal. We explained how the non-compliance prediction can be realised by pattern detection over event streams of smarter meter readings. Using the hybrid load shedding technique and the EIRES framework, we applied the proposed approach at utility scale, with low computation and communication overheads.

The work presented in this dissertation has several implications for the research in event stream processing.

First, it stresses the importance of state management for efficient event stream processing. Previous strategies for lossy optimisation work on the streaming data used as input. However, we are the first to argue that the state dominates the effectiveness of lossy optimisations in pattern detection queries. In addition, we argue that the irregular remote data access patterns in event stream processing are determined by the dynamic state rather than static or dynamic program analysis. Optimisations for hiding remote data transmission latency should therefore be based on state management.

Second, our optimisation techniques are independent of any specific query language and execution model. As long as the query evaluation is based on the state and state transitions, our proposed approaches can be applied.

Third, we provide efficient prototype implementations as open-source projects and illustrate that estimation and approximation techniques (*e.g.*, sketching) can be employed to reduce computational overhead for online processing.

7.2 Future Work

The research reported in this dissertation opens up several directions for future research. First of all, techniques to control of the trade-offs between query results quality and processing latency could be added to the hybrid load shedding technique. Given a certain bound of query results quality (*e.g.*, 60% recall or 70% accuracy), the hybrid load shedding technique could be adapted to choose what and how much to discard in order to achieve minimum latency. This could be

further extended towards a pay-as-you-go computing mode. This means that based on the budget of compute resources, the requirement of the query results' quality, and the latency bound, the hybrid load shedding shall select the most valuable state and events to process first.

Second, the interactions of hybrid load shedding and optimisation techniques in the EIRES framework could be further investigated. When the hybrid load shedding discards partial matches that need remote data elements, the remote data access patterns are affected, which directly relates to the effectiveness of prefetching, lazy evaluation, and caching of within EIRES. In turn, the selection of remote data to fetch also affects the growth of partial matches. Since partial matches for which required remote data has been returned to the ESP engine will be processed first, which in turn affects the hybrid load shedding. Therefore, a comprehensive cost model combining both, the hybrid load shedding and the EIRES framework, is a valuable direction for future research.

Last, the core concept of state management, especially the cost models based on states, may be applied beyond the efficiency of event stream processing. For instance, state management could also guide the fairness of processing multiple, distributed data streams. That is, one could sue them to optimise how much compute resources should be allocated to each stream to avoid unfairness according to quality of service (QoS) requirements of specific applications (*e.g.*, avoiding starvation of any input stream). In another example of machine learning systems, one performance bottleneck is the efficiency linked to input pipelines for training data [133], According to [133], "30% of the total compute time is spent ingesting data" and "20% of jobs spend more than a third of their compute time in the input pipeline". However, the selection of which input data stream to consider in training also affects the quality of the final trained model (*e.g.*, there is a bias towards the stream from which most training has been obtained). We believe that state management of data streams will shed light on optimisation opportunities in this field.

BIBLIOGRAPHY

[1] D. J. ABADI, D. CARNEY, U. ÇETINTEMEL, M. CHERNIACK, C. CONVEY, S. LEE, M. STONEBRAKER, N. TATBUL, AND S. B. ZDONIK, *Aurora: a new model and architecture for data stream management*, VLDB J., 12 (2003), pp. 120–139.

[2] A. S. ABDELHAMID, A. R. MAHMOOD, A. DAGHISTANI, AND W. G. AREF, *Prompt: Dynamic data-partitioning for distributed micro-batch stream processing systems*, in Proceedings of the 2020 International Conference on Management of Data, SIGMOD Conference 2020, online conference [Portland, OR, USA], June 14-19, 2020, D. Maier, R. Pottinger, A. Doan, W. Tan, A. Alawini, and H. Q. Ngo, eds., ACM, 2020, pp. 2455–2469.

[3] A. ADI AND O. ETZION, *Amit - the situation manager*, VLDB J., 13 (2004), pp. 177–203.

[4] S. AGARWAL, B. MOZAFARI, A. PANDA, H. MILNER, S. MADDEN, AND I. STOICA, *Blinkdb: queries with bounded errors and bounded response times on very large data*, in Eighth Eurosys Conference 2013, EuroSys '13, Prague, Czech Republic, April 14-17, 2013, Z. Hanzálek, H. Härtig, M. Castro, and M. F. Kaashoek, eds., ACM, 2013, pp. 29–42.

[5] J. AGRAWAL, Y. DIAO, D. GYLLSTROM, AND N. IMMERMAN, *Efficient pattern matching over event streams*, in Proceedings of the ACM SIGMOD International Conference on Management of Data, SIGMOD 2008, Vancouver, BC, Canada, June 10-12, 2008, J. T. Wang, ed., ACM, 2008, pp. 147–160.

[6] T. AKIDAU, A. BALIKOV, K. BEKIROGLU, S. CHERNYAK, J. HABERMAN, R. LAX, S. MCVEETY, D. MILLS, P. NORDSTROM, AND S. WHITTLE, *Millwheel: Fault-tolerant stream processing at internet scale*, Proc. VLDB Endow., 6 (2013), pp. 1033–1044.

[7] M. ALTINEL AND M. J. FRANKLIN, *Efficient filtering of XML documents for selective dissemination of information*, in VLDB 2000, Proceedings of 26th International Conference on Very Large Data Bases, September 10-14, 2000, Cairo, Egypt, A. E. Abbadi, M. L. Brodie, S. Chakravarthy, U. Dayal, N. Kamel, G. Schlageter, and K. Whang, eds., Morgan Kaufmann, 2000, pp. 53–64.

[8] AMAZON KINESE, *Amazon Kinese Data Streaming FAQs*. https://aws.amazon.com/kinesis/data-streams/faqs/, 2019.

Last access: 05/08/21.

[9] K. AMIRI, S. PARK, R. TEWARI, AND S. PADMANABHAN, *Dbproxy: A dynamic data cache for web applications*, in Proceedings of the 19th International Conference on Data Engineering, March 5-8, 2003, Bangalore, India, U. Dayal, K. Ramamritham, and T. M. Vijayaraman, eds., IEEE Computer Society, 2003, pp. 821–831.

[10] R. ANANTHANARAYANAN, V. BASKER, S. DAS, A. GUPTA, H. JIANG, T. QIU, A. REZNICHENKO, D. RYABKOV, M. SINGH, AND S. VENKATARAMAN, *Photon: fault-tolerant and scalable joining of continuous data streams*, in Proceedings of the ACM SIGMOD International Conference on Management of Data, SIGMOD 2013, New York, NY, USA, June 22-27, 2013, K. A. Ross, D. Srivastava, and D. Papadias, eds., ACM, 2013, pp. 577–588.

[11] APACHE PULSAR, *Pulsar Load Distribution*. https://pulsar.apache.org/docs/v2.0.1-incubating/admin/LoadDistribution/, 2019.
Last access: 05/08/21.

[12] A. ARASU, B. BABCOCK, S. BABU, M. DATAR, K. ITO, I. NISHIZAWA, J. ROSENSTEIN, AND J. WIDOM, *STREAM: the stanford stream data manager*, in Proceedings of the 2003 ACM SIGMOD International Conference on Management of Data, San Diego, California, USA, June 9-12, 2003, A. Y. Halevy, Z. G. Ives, and A. Doan, eds., ACM, 2003, p. 665.

[13] A. ARASU, S. BABU, AND J. WIDOM, *The CQL continuous query language: semantic foundations and query execution*, VLDB J., 15 (2006), pp. 121–142.

[14] A. ARASU AND J. WIDOM, *Resource sharing in continuous sliding-window aggregates*, in (e)Proceedings of the Thirtieth International Conference on Very Large Data Bases, VLDB 2004, Toronto, Canada, August 31 - September 3 2004, M. A. Nascimento, M. T. Özsu, D. Kossmann, R. J. Miller, J. A. Blakeley, and K. B. Schiefer, eds., Morgan Kaufmann, 2004, pp. 336–347.

[15] A. ARTIKIS, A. MARGARA, M. UGARTE, S. VANSUMMEREN, AND M. WEIDLICH, *Complex event recognition languages: Tutorial*, in Proceedings of the 11th ACM International Conference on Distributed and Event-based Systems, DEBS 2017, Barcelona, Spain, June 19-23, 2017, ACM, 2017, pp. 7–10.

[16] A. ARTIKIS, M. WEIDLICH, F. SCHNITZLER, I. BOUTSIS, T. LIEBIG, N. PIATKOWSKI, C. BOCKERMANN, K. MORIK, V. KALOGERAKI, J. MARECEK, A. GAL, S. MANNOR, D. GUNOPULOS, AND D. KINANE, *Heterogeneous stream processing and crowdsourcing*

for urban traffic management, in Proceedings of the 17th International Conference on Extending Database Technology, EDBT 2014, Athens, Greece, March 24-28, 2014., 2014, pp. 712–723.

[17] M. BALAZINSKA, H. BALAKRISHNAN, S. MADDEN, AND M. STONEBRAKER, *Fault-tolerance in the borealis distributed stream processing system*, in Proceedings of the ACM SIGMOD International Conference on Management of Data, Baltimore, Maryland, USA, June 14-16, 2005, F. Özcan, ed., ACM, 2005, pp. 13–24.

[18] C. BALKESEN, N. DINDAR, M. WETTER, AND N. TATBUL, *RIP: run-based intra-query parallelism for scalable complex event processing*, in The 7th ACM International Conference on Distributed Event-Based Systems, DEBS '13, Arlington, TX, USA - June 29 - July 03, 2013, 2013, pp. 3–14.

[19] C. BALKESEN AND N. TATBUL, *Scalable data partitioning techniques for parallel sliding window processing over data streams*, 2011.

[20] C. BALKESEN, N. TATBUL, AND M. T. ÖZSU, *Adaptive input admission and management for parallel stream processing*, in The 7th ACM International Conference on Distributed Event-Based Systems, DEBS '13, Arlington, TX, USA - June 29 - July 03, 2013, S. Chakravarthy, S. D. Urban, P. R. Pietzuch, and E. A. Rundensteiner, eds., ACM, 2013, pp. 15–26.

[21] J. BANG, S. SON, H. KIM, Y. MOON, AND M. CHOI, *Design and implementation of a load shedding engine for solving starvation problems in apache kafka*, in 2018 IEEE/IFIP Network Operations and Management Symposium, NOMS 2018, Taipei, Taiwan, April 23-27, 2018, IEEE, 2018, pp. 1–4.

[22] C. BARTON, P. CHARLES, D. GOYAL, M. RAGHAVACHARI, M. FONTOURA, AND V. JOSIFOVSKI, *Streaming xpath processing with forward and backward axes*, in Proceedings of the 19th International Conference on Data Engineering, March 5-8, 2003, Bangalore, India, U. Dayal, K. Ramamritham, and T. M. Vijayaraman, eds., IEEE Computer Society, 2003, pp. 455–466.

[23] L. BATTLE, R. CHANG, AND M. STONEBRAKER, *Dynamic prefetching of data tiles for interactive visualization*, in Proceedings of the 2016 International Conference on Management of Data, SIGMOD Conference 2016, San Francisco, CA, USA, June 26 - July 01, 2016, F. Özcan, G. Koutrika, and S. Madden, eds., ACM, 2016, pp. 1363–1375.

[24] P. BEN-NUN, *Respondent fatigue*, Encyclopedia of survey research methods, 2 (2008), pp. 742–743.

[25] M. B. BLAKE, D. J. CUMMINGS, A. BANSAL, AND S. K. BANSAL, *Workflow composition of service level agreements for web services*, Decision Support Systems, 53 (2012), pp. 234–244.

[26] R. BLANCO, E. BORTNIKOV, F. JUNQUEIRA, R. LEMPEL, L. TELLOLI, AND H. ZARAGOZA, *Caching search engine results over incremental indices*, in Proceeding of the 33rd International ACM SIGIR Conference on Research and Development in Information Retrieval, SIGIR 2010, Geneva, Switzerland, July 19-23, 2010, F. Crestani, S. Marchand-Maillet, H. Chen, E. N. Efthimiadis, and J. Savoy, eds., ACM, 2010, pp. 82–89.

[27] S. BOAG, D. CHAMBERLIN, M. F. FERNANDEZ, D. FLORESCU, J. ROBIE, AND J. SIMÉON, *Xquery 1.0: An xml query languages*, in World Wide Web Consortium, November, 2003.

[28] C. BORNHÖVD, M. ALTINEL, C. MOHAN, H. PIRAHESH, AND B. REINWALD, *Adaptive database caching with dbcache*, IEEE Data Eng. Bull., 27 (2004), pp. 11–18.

[29] S. BÖRZSÖNYI, D. KOSSMANN, AND K. STOCKER, *The skyline operator*, in Proceedings of the 17th International Conference on Data Engineering, April 2-6, 2001, Heidelberg, Germany, D. Georgakopoulos and A. Buchmann, eds., IEEE Computer Society, 2001, pp. 421–430.

[30] I. BOTAN, R. DERAKHSHAN, N. DINDAR, L. M. HAAS, R. J. MILLER, AND N. TATBUL, *SECRET: A model for analysis of the execution semantics of stream processing systems*, Proc. VLDB Endow., 3 (2010), pp. 232–243.

[31] I. T. BOWMAN AND K. SALEM, *Optimization of query streams using semantic prefetching*, in Proceedings of the ACM SIGMOD International Conference on Management of Data, Paris, France, June 13-18, 2004, G. Weikum, A. C. König, and S. Deßloch, eds., ACM, 2004, pp. 179–190.

[32] L. BRENNA, A. J. DEMERS, J. GEHRKE, M. HONG, J. OSSHER, B. PANDA, M. RIEDEWALD, M. THATTE, AND W. M. WHITE, *Cayuga: a high-performance event processing engine*, in Proceedings of the ACM SIGMOD International Conference on Management of Data, Beijing, China, June 12-14, 2007, C. Y. Chan, B. C. Ooi, and A. Zhou, eds., ACM, 2007, pp. 1100–1102.

[33] A. BROWN AND W. LAVALLE, *Hailing a change: comparing taxi and ridehail service quality in los angeles*, Transportation, 48 (2021), p. 1007–1031.

[34] J. CALBIMONTE, J. MORA, AND Ó. CORCHO, *Query rewriting in RDF stream processing*, in The Semantic Web. Latest Advances and New Domains - 13th International Conference, ESWC 2016, Heraklion, Crete, Greece, May 29 - June 2, 2016, Proceedings, H. Sack,

E. Blomqvist, M. d'Aquin, C. Ghidini, S. P. Ponzetto, and C. Lange, eds., vol. 9678 of Lecture Notes in Computer Science, Springer, 2016, pp. 486–502.

[35] K. S. CANDAN, W. LI, Q. LUO, W. HSIUNG, AND D. AGRAWAL, *Enabling dynamic content caching for database-driven web sites*, in Proceedings of the 2001 ACM SIGMOD international conference on Management of data, Santa Barbara, CA, USA, May 21-24, 2001, S. Mehrotra and T. K. Sellis, eds., ACM, 2001, pp. 532–543.

[36] L. CAO, J. WANG, AND E. A. RUNDENSTEINER, *Sharing-aware outlier analytics over high-volume data streams*, in Proceedings of the 2016 International Conference on Management of Data, SIGMOD Conference 2016, San Francisco, CA, USA, June 26 - July 01, 2016, F. Özcan, G. Koutrika, and S. Madden, eds., ACM, 2016, pp. 527–540.

[37] P. CARBONE, S. EWEN, G. FÓRA, S. HARIDI, S. RICHTER, AND K. TZOUMAS, *State management in apache flink®: Consistent stateful distributed stream processing*, Proc. VLDB Endow., 10 (2017), pp. 1718–1729.

[38] P. CARBONE, A. KATSIFODIMOS, S. EWEN, V. MARKL, S. HARIDI, AND K. TZOUMAS, *Apache flink™: Stream and batch processing in a single engine*, IEEE Data Eng. Bull., 38 (2015), pp. 28–38.

[39] U. ÇETINTEMEL, D. J. ABADI, Y. AHMAD, H. BALAKRISHNAN, M. BALAZINSKA, M. CHERNIACK, J. HWANG, S. MADDEN, A. MASKEY, A. RASIN, E. RYVKINA, M. STONEBRAKER, N. TATBUL, Y. XING, AND S. ZDONIK, *The aurora and borealis stream processing engines*, in Data Stream Management - Processing High-Speed Data Streams, M. N. Garofalakis, J. Gehrke, and R. Rastogi, eds., Data-Centric Systems and Applications, Springer, 2016, pp. 337–359.

[40] S. CHAKRAVARTHY AND D. MISHRA, *Snoop: An expressive event specification language for active databases*, Data Knowl. Eng., 14 (1994), pp. 1–26.

[41] C. Y. CHAN, W. FAN, P. FELBER, M. N. GAROFALAKIS, AND R. RASTOGI, *Tree pattern aggregation for scalable XML data dissemination*, in Proceedings of 28th International Conference on Very Large Data Bases, VLDB 2002, Hong Kong, August 20-23, 2002, Morgan Kaufmann, 2002, pp. 826–837.

[42] C. Y. CHAN, P. FELBER, M. N. GAROFALAKIS, AND R. RASTOGI, *Efficient filtering of XML documents with xpath expressions*, VLDB J., 11 (2002), pp. 354–379.

[43] B. CHANDRAMOULI, J. GOLDSTEIN, M. BARNETT, R. DELINE, J. C. PLATT, J. F. TERWILLIGER, AND J. WERNSING, *Trill: A high-performance incremental query processor for diverse analytics*, Proc. VLDB Endow., 8 (2014), pp. 401–412.

[44] S. CHANDRASEKARAN, O. COOPER, A. DESHPANDE, M. J. FRANKLIN, J. M. HELLERSTEIN, W. HONG, S. KRISHNAMURTHY, S. MADDEN, F. REISS, AND M. A. SHAH, *Telegraphcq: Continuous dataflow processing*, in Proceedings of the 2003 ACM SIGMOD International Conference on Management of Data, San Diego, California, USA, June 9-12, 2003, A. Y. Halevy, Z. G. Ives, and A. Doan, eds., ACM, 2003, p. 668.

[45] S. CHAUDHURI, B. DING, AND S. KANDULA, *Approximate query processing: No silver bullet*, in Proceedings of the 2017 ACM International Conference on Management of Data, SIGMOD Conference 2017, Chicago, IL, USA, May 14-19, 2017, 2017, pp. 511–519.

[46] C. CHEKURI AND S. KHANNA, *A polynomial time approximation scheme for the multiple knapsack problem*, SIAM J. Comput., 35 (2005), pp. 713–728.

[47] J. CHEN, D. J. DEWITT, F. TIAN, AND Y. WANG, *Niagaracq: A scalable continuous query system for internet databases*, in Proceedings of the 2000 ACM SIGMOD International Conference on Management of Data, May 16-18, 2000, Dallas, Texas, USA, W. Chen, J. F. Naughton, and P. A. Bernstein, eds., ACM, 2000, pp. 379–390.

[48] S. CHEN, A. AILAMAKI, P. B. GIBBONS, AND T. C. MOWRY, *Improving hash join performance through prefetching*, in Proceedings of the 20th International Conference on Data Engineering, ICDE 2004, 30 March - 2 April 2004, Boston, MA, USA, Z. M. Özsoyoglu and S. B. Zdonik, eds., IEEE Computer Society, 2004, pp. 116–127.

[49] S. CHEN, P. B. GIBBONS, AND T. C. MOWRY, *Improving index performance through prefetching*, in Proceedings of the 2001 ACM SIGMOD international conference on Management of data, Santa Barbara, CA, USA, May 21-24, 2001, S. Mehrotra and T. K. Sellis, eds., ACM, 2001, pp. 235–246.

[50] S. CHEN, P. B. GIBBONS, T. C. MOWRY, AND G. VALENTIN, *Fractal prefetching b±trees: optimizing both cache and disk performance*, in Proceedings of the 2002 ACM SIGMOD International Conference on Management of Data, Madison, Wisconsin, USA, June 3-6, 2002, M. J. Franklin, B. Moon, and A. Ailamaki, eds., ACM, 2002, pp. 157–168.

[51] Y. CHEN, S. B. DAVIDSON, AND Y. ZHENG, *An efficient xpath query processor for XML streams*, in Proceedings of the 22nd International Conference on Data Engineering, ICDE 2006, 3-8 April 2006, Atlanta, GA, USA, L. Liu, A. Reuter, K. Whang, and J. Zhang, eds., IEEE Computer Society, 2006, p. 79.

[52] Z. CHENG, Q. HUANG, AND P. P. C. LEE, *On the performance and convergence of distributed stream processing via approximate fault tolerance*, VLDB J., 28 (2019), pp. 821–846.

[53] L. CHUAN AND A. UKIL, *Modeling and validation of electrical load profiling in residential buildings in singapore*, IEEE Trans. Power Syst., 30 (2015), pp. 2800–2809.

[54] CITI BIKE, *System Data*.
http://www.citibikenyc.com/system-data, 2019.
Last access: 05/08/21.

[55] J. CLARK AND S. DEROSE, *Xml path language (xpath) version 1.0*, in World Wide Web Consortium, November, 1999.

[56] G. CORMODE, M. N. GAROFALAKIS, P. J. HAAS, AND C. JERMAINE, *Synopses for massive data: Samples, histograms, wavelets, sketches*, Foundations and Trends in Databases, 4 (2012), pp. 1–294.

[57] G. CORMODE AND S. MUTHUKRISHNAN, *An improved data stream summary: the count-min sketch and its applications*, J. Algorithms, 55 (2005), pp. 58–75.

[58] G. CORMODE AND S. M. MUTHUKRISHNAN, *Approximating data with the count-min sketch*, IEEE Software, 29 (2012), pp. 64–69.

[59] C. D. CRANOR, T. JOHNSON, O. SPATSCHECK, AND V. SHKAPENYUK, *Gigascope: A stream database for network applications*, in Proceedings of the 2003 ACM SIGMOD International Conference on Management of Data, San Diego, California, USA, June 9-12, 2003, A. Y. Halevy, Z. G. Ives, and A. Doan, eds., ACM, 2003, pp. 647–651.

[60] G. CUGOLA AND A. MARGARA, *Complex event processing with T-REX*, J. Syst. Softw., 85 (2012), pp. 1709–1728.

[61] G. CUGOLA AND A. MARGARA, *Processing flows of information: From data stream to complex event processing*, ACM Comput. Surv., 44 (2012), pp. 15:1–15:62.

[62] A. DAS, J. GEHRKE, AND M. RIEDEWALD, *Approximate join processing over data streams*, in Proceedings of the 2003 ACM SIGMOD International Conference on Management of Data, San Diego, California, USA, June 9-12, 2003, A. Y. Halevy, Z. G. Ives, and A. Doan, eds., ACM, 2003, pp. 40–51.

[63] T. DAS, Y. ZHONG, I. STOICA, AND S. SHENKER, *Adaptive stream processing using dynamic batch sizing*, in Proceedings of the ACM Symposium on Cloud Computing, Seattle, WA, USA, November 3-5, 2014, E. Lazowska, D. Terry, R. H. Arpaci-Dusseau, and J. Gehrke, eds., ACM, 2014, pp. 16:1–16:13.

[64] M. DAYARATHNA AND S. PERERA, *Recent advancements in event processing*, ACM Comput. Surv., 51 (2018), pp. 33:1–33:36.

[65] J. DEAN AND S. GHEMAWAT, *Mapreduce: Simplified data processing on large clusters*, in 6th Symposium on Operating System Design and Implementation (OSDI 2004),

San Francisco, California, USA, December 6-8, 2004, E. A. Brewer and P. Chen, eds., USENIX Association, 2004, pp. 137–150.

[66] Y. DIAO, S. RIZVI, AND M. J. FRANKLIN, *Towards an internet-scale XML dissemination service*, in (e)Proceedings of the Thirtieth International Conference on Very Large Data Bases, VLDB 2004, Toronto, Canada, August 31 - September 3 2004, M. A. Nascimento, M. T. Özsu, D. Kossmann, R. J. Miller, J. A. Blakeley, and K. B. Schiefer, eds., Morgan Kaufmann, 2004, pp. 612–623.

[67] L. DING, S. CHEN, E. A. RUNDENSTEINER, J. TATEMURA, W. HSIUNG, AND K. S. CANDAN, *Runtime semantic query optimization for event stream processing*, in Proceedings of the 24th International Conference on Data Engineering, ICDE 2008, April 7-12, 2008, Cancún, Mexico, G. Alonso, J. A. Blakeley, and A. L. P. Chen, eds., IEEE Computer Society, 2008, pp. 676–685.

[68] L. DING, K. WORKS, AND E. A. RUNDENSTEINER, *Semantic stream query optimization exploiting dynamic metadata*, in Proceedings of the 27th International Conference on Data Engineering, ICDE 2011, April 11-16, 2011, Hannover, Germany, 2011, pp. 111–122.

[69] ESPER, *Complex Event Processing, Streaming Analytics*.
https://www.espertech.com/, 2021.
Last access: 05/08/21.

[70] O. ETZION AND P. NIBLETT, *Event Processing in Action*, Manning Publications Company, 2010.

[71] FEEDZAI.COM, *Modern Payment Fraud Prevention at Big Data Scale*.
https://www.pymnts.com/assets/Uploads/Feedzai-Whitepaper-Modern-Payment-Fraud-Prevention-at-Big-Data-Scale-v2.pdf, 2013.
Last access: 05/08/21.

[72] R. C. FERNANDEZ, M. MIGLIAVACCA, E. KALYVIANAKI, AND P. R. PIETZUCH, *Integrating scale out and fault tolerance in stream processing using operator state management*, in Proceedings of the ACM SIGMOD International Conference on Management of Data, SIGMOD 2013, New York, NY, USA, June 22-27, 2013, K. A. Ross, D. Srivastava, and D. Papadias, eds., ACM, 2013, pp. 725–736.

[73] R. C. FERNANDEZ, M. MIGLIAVACCA, E. KALYVIANAKI, AND P. R. PIETZUCH, *Making state explicit for imperative big data processing*, in 2014 USENIX Annual Technical Conference, USENIX ATC '14, Philadelphia, PA, USA, June 19-20, 2014, G. Gibson and N. Zeldovich, eds., USENIX Association, 2014, pp. 49–60.

[74] A. FLORATOU, A. AGRAWAL, B. GRAHAM, S. RAO, AND K. RAMASAMY, *Dhalion: Self-regulating stream processing in heron*, PVLDB, 10 (2017), pp. 1825–1836.

[75] D. FLORESCU, C. HILLERY, D. KOSSMANN, P. LUCAS, F. RICCARDI, T. WESTMANN, M. J. CAREY, AND A. SUNDARARAJAN, *The BEA streaming xquery processor*, VLDB J., 13 (2004), pp. 294–315.

[76] S. FOSSO WAMBA AND H. BOECK, *Enhancing information flow in a retail supply chain using rfid and the epc network: a proof-of-concept approach*, (2008).

[77] M. FRAGKOULIS, P. CARBONE, V. KALAVRI, AND A. KATSIFODIMOS, *A survey on the evolution of stream processing systems*, CoRR, abs/2008.00842 (2020).

[78] FRAUGSTER.
https://fraugster.com/, 2019.

[79] M. FU, S. MITTAL, V. KEDIGEHALLI, K. RAMASAMY, M. BARRY, A. JORGENSEN, C. KELLOGG, N. LU, B. GRAHAM, AND J. WU, *Streaming@twitter*, IEEE Data Eng. Bull., 38 (2015), pp. 15–27.

[80] C. GARROD, A. MANJHI, A. AILAMAKI, B. M. MAGGS, T. C. MOWRY, C. OLSTON, AND A. TOMASIC, *Scalable query result caching for web applications*, Proc. VLDB Endow., 1 (2008), pp. 550–561.

[81] B. GEDIK, *Partitioning functions for stateful data parallelism in stream processing*, VLDB J., 23 (2014), pp. 517–539.

[82] B. GEDIK, H. ANDRADE, K. WU, P. S. YU, AND M. DOO, *SPADE: the system s declarative stream processing engine*, in Proceedings of the ACM SIGMOD International Conference on Management of Data, SIGMOD 2008, Vancouver, BC, Canada, June 10-12, 2008, J. T. Wang, ed., ACM, 2008, pp. 1123–1134.

[83] B. GEDIK, K. WU, AND P. S. YU, *Efficient construction of compact shedding filters for data stream processing*, in Proceedings of the 24th International Conference on Data Engineering, ICDE 2008, April 7-12, 2008, Cancún, México, G. Alonso, J. A. Blakeley, and A. L. P. Chen, eds., IEEE Computer Society, 2008, pp. 396–405.

[84] B. GEDIK, K. WU, P. S. YU, AND L. LIU, *Adaptive load shedding for windowed stream joins*, in Proceedings of the 2005 ACM CIKM International Conference on Information and Knowledge Management, Bremen, Germany, October 31 - November 5, 2005, O. Herzog, H. Schek, N. Fuhr, A. Chowdhury, and W. Teiken, eds., ACM, 2005, pp. 171–178.

[85] B. GEDIK, K. WU, P. S. YU, AND L. LIU, *A load shedding framework and optimizations for m-way windowed stream joins*, in Proceedings of the 23rd International Conference

on Data Engineering, ICDE 2007, The Marmara Hotel, Istanbul, Turkey, April 15-20, 2007, R. Chirkova, A. Dogac, M. T. Özsu, and T. K. Sellis, eds., IEEE Computer Society, 2007, pp. 536–545.

[86] F. GESSERT, F. BUCKLERS, AND N. RITTER, *Orestes: A scalable database-as-a-service architecture for low latency*, in Workshops Proceedings of the 30th International Conference on Data Engineering Workshops, ICDE 2014, Chicago, IL, USA, March 31 - April 4, 2014, IEEE Computer Society, 2014, pp. 215–222.

[87] F. GESSERT, M. SCHAARSCHMIDT, W. WINGERATH, E. WITT, E. YONEKI, AND N. RITTER, *Quaestor: Query web caching for database-as-a-service providers*, Proc. VLDB Endow., 10 (2017), pp. 1670–1681.

[88] M. L. GOLDBERG AND G. K. AGNEW, *Measurement and verification for demand response*, US Department of Energy: Washington, DC, USA, (2013).

[89] A. K. GUPTA AND D. SUCIU, *Stream processing of xpath queries with predicates*, in Proceedings of the 2003 ACM SIGMOD International Conference on Management of Data, San Diego, California, USA, June 9-12, 2003, A. Y. Halevy, Z. G. Ives, and A. Doan, eds., ACM, 2003, pp. 419–430.

[90] D. GYLLSTROM, J. AGRAWAL, Y. DIAO, AND N. IMMERMAN, *On supporting kleene closure over event streams*, in Proceedings of the 24th International Conference on Data Engineering, ICDE 2008, April 7-12, 2008, Cancún, Mexico, G. Alonso, J. A. Blakeley, and A. L. P. Chen, eds., IEEE Computer Society, 2008, pp. 1391–1393.

[91] D. GYLLSTROM, E. WU, H. CHAE, Y. DIAO, P. STAHLBERG, AND G. ANDERSON, *SASE: complex event processing over streams (demo)*, in Third Biennial Conference on Innovative Data Systems Research, CIDR 2007, Asilomar, CA, USA, January 7-10, 2007, Online Proceedings, www.cidrdb.org, 2007, pp. 407–411.

[92] HADOOP, .
https://hadoop.apache.org/, 2021.
Last access: 05/08/21.

[93] A. HAGIESCU, W. WONG, D. F. BACON, AND R. M. RABBAH, *A computing origami: folding streams in fpgas*, in Proceedings of the 46th Design Automation Conference, DAC 2009, San Francisco, CA, USA, July 26-31, 2009, ACM, 2009, pp. 282–287.

[94] Y. HE, S. BARMAN, AND J. F. NAUGHTON, *On load shedding in complex event processing*, in Proc. 17th International Conference on Database Theory (ICDT), Athens, Greece, March 24-28, 2014., 2014, pp. 213–224.

[95] M. HIRZEL, *Partition and compose: parallel complex event processing*, in Proceedings of the Sixth ACM International Conference on Distributed Event-Based Systems, DEBS 2012, Berlin, Germany, July 16-20, 2012, 2012, pp. 191–200.

[96] M. HIRZEL, G. BAUDART, A. BONIFATI, E. D. VALLE, S. SAKR, AND A. VLACHOU, *Stream processing languages in the big data era*, SIGMOD Rec., 47 (2018), pp. 29–40.

[97] Q. HUANG AND P. P. C. LEE, *Toward high-performance distributed stream processing via approximate fault tolerance*, Proc. VLDB Endow., 10 (2016), pp. 73–84.

[98] J. HWANG, M. BALAZINSKA, A. RASIN, U. ÇETINTEMEL, M. STONEBRAKER, AND S. B. ZDONIK, *High-availability algorithms for distributed stream processing*, in Proceedings of the 21st International Conference on Data Engineering, ICDE 2005, 5-8 April 2005, Tokyo, Japan, K. Aberer, M. J. Franklin, and S. Nishio, eds., IEEE Computer Society, 2005, pp. 779–790.

[99] M. JAIN ET AL., *Methodologies for effective demand response messaging*, in IEEE Int. Conf. Smart Grid Commun., 2015, pp. 453–458.

[100] Y. JI, H. ZHOU, Z. JERZAK, A. NICA, G. HACKENBROICH, AND C. FETZER, *Quality-driven continuous query execution over out-of-order data streams*, in Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data, Melbourne, Victoria, Australia, May 31 - June 4, 2015, T. K. Sellis, S. B. Davidson, and Z. G. Ives, eds., ACM, 2015, pp. 889–894.

[101] J. KANG, J. F. NAUGHTON, AND S. VIGLAS, *Evaluating window joins over unbounded streams*, in Proceedings of the 19th International Conference on Data Engineering, March 5-8, 2003, Bangalore, India, U. Dayal, K. Ramamritham, and T. M. Vijayaraman, eds., IEEE Computer Society, 2003, pp. 341–352.

[102] N. R. KATSIPOULAKIS, A. LABRINIDIS, AND P. K. CHRYSANTHIS, *Concept-driven load shedding: Reducing size and error of voluminous and variable data streams*, in IEEE International Conference on Big Data, Big Data 2018, Seattle, WA, USA, December 10-13, 2018, N. Abe, H. Liu, C. Pu, X. Hu, N. Ahmed, M. Qiao, Y. Song, D. Kossmann, B. Liu, K. Lee, J. Tang, J. He, and J. Saltz, eds., IEEE, 2018, pp. 418–427.

[103] O. KAYA, *Research Briefing: High-frequency trading*. https://www.dbresearch.com/PROD/RPS_EN-PROD/PROD0000000000454703/Research_Briefing%3A_High-frequency_trading.PDF, 2016.
Last access: 05/08/21.

[104] H. KELLERER, U. PFERSCHY, AND D. PISINGER, *Knapsack problems*, Springer, 2004.

[105] J. KINGMAN, *Poisson Processes*, Oxford Studies in Probability, Clarendon Press, 1992.

[106] Y. O. KOÇBERBER, B. FALSAFI, AND B. GROT, *Asynchronous memory access chaining*, Proc. VLDB Endow., 9 (2015), pp. 252–263.

[107] C. KOCH, *XML stream processing*, in Encyclopedia of Database Systems, Second Edition, L. Liu and M. T. Özsu, eds., Springer, 2018.

[108] C. KOCH, S. SCHERZINGER, AND M. SCHMIDT, *The GCX system: Dynamic buffer minimization in streaming xquery evaluation*, in Proceedings of the 33rd International Conference on Very Large Data Bases, University of Vienna, Austria, September 23-27, 2007, C. Koch, J. Gehrke, M. N. Garofalakis, D. Srivastava, K. Aberer, A. Deshpande, D. Florescu, C. Y. Chan, V. Ganti, C. Kanne, W. Klas, and E. J. Neuhold, eds., ACM, 2007, pp. 1378–1381.

[109] C. KOCH, S. SCHERZINGER, N. SCHWEIKARDT, AND B. STEGMAIER, *Fluxquery: An optimizing xquery processor for streaming XML data*, in (e)Proceedings of the Thirtieth International Conference on Very Large Data Bases, VLDB 2004, Toronto, Canada, August 31 - September 3 2004, M. A. Nascimento, M. T. Özsu, D. Kossmann, R. J. Miller, J. A. Blakeley, and K. B. Schiefer, eds., Morgan Kaufmann, 2004, pp. 1309–1312.

[110] A. KOLIOUSIS, M. WEIDLICH, R. C. FERNANDEZ, A. L. WOLF, P. COSTA, AND P. R. PIETZUCH, *SABER: window-based hybrid stream processing for heterogeneous architectures*, in Proceedings of the 2016 International Conference on Management of Data, SIGMOD Conference 2016, San Francisco, CA, USA, June 26 - July 01, 2016, F. Özcan, G. Koutrika, and S. Madden, eds., ACM, 2016, pp. 555–569.

[111] D. KOSSMANN, *The state of the art in distributed query processing*, ACM Comput. Surv., 32 (2000), pp. 422–469.

[112] A. KUNDU, S. PANIGRAHI, S. SURAL, AND A. K. MAJUMDAR, *Blast-ssaha hybridization for credit card fraud detection*, IEEE transactions on dependable and Secure Computing, 6 (2009), pp. 309–315.

[113] Y. KWON, M. BALAZINSKA, AND A. G. GREENBERG, *Fault-tolerant stream processing using a distributed, replicated file system*, Proc. VLDB Endow., 1 (2008), pp. 574–585.

[114] P. LARSON, J. GOLDSTEIN, AND J. ZHOU, *Mtcache: Transparent mid-tier database caching in SQL server*, in Proceedings of the 20th International Conference on Data Engineering, ICDE 2004, 30 March - 2 April 2004, Boston, MA, USA, Z. M. Özsoyoglu and S. B. Zdonik, eds., IEEE Computer Society, 2004, pp. 177–188.

[115] C. LEI, E. A. RUNDENSTEINER, AND J. D. GUTTMAN, *Robust distributed stream processing*, in 29th IEEE International Conference on Data Engineering, ICDE 2013, Brisbane, Australia, April 8-12, 2013, C. S. Jensen, C. M. Jermaine, and X. Zhou, eds., IEEE Computer Society, 2013, pp. 817–828.

[116] J. LI, D. MAIER, K. TUFTE, V. PAPADIMOS, AND P. A. TUCKER, *No pane, no gain: efficient evaluation of sliding-window aggregates over data streams*, SIGMOD Rec., 34 (2005), pp. 39–44.

[117] J. LI, D. MAIER, K. TUFTE, V. PAPADIMOS, AND P. A. TUCKER, *Semantics and evaluation techniques for window aggregates in data streams*, in Proceedings of the ACM SIGMOD International Conference on Management of Data, Baltimore, Maryland, USA, June 14-16, 2005, F. Özcan, ed., ACM, 2005, pp. 311–322.

[118] J. LI, K. TUFTE, V. SHKAPENYUK, V. PAPADIMOS, T. JOHNSON, AND D. MAIER, *Out-of-order processing: a new architecture for high-performance stream systems*, Proc. VLDB Endow., 1 (2008), pp. 274–288.

[119] M. LI, M. MANI, E. A. RUNDENSTEINER, AND T. LIN, *Constraint-aware complex event pattern detection over streams*, in Database Systems for Advanced Applications, 15th International Conference, DASFAA 2010, Tsukuba, Japan, April 1-4, 2010, Proceedings, Part II, H. Kitagawa, Y. Ishikawa, Q. Li, and C. Watanabe, eds., vol. 5982 of Lecture Notes in Computer Science, Springer, 2010, pp. 199–215.

[120] Z. LI AND T. GE, *History is a mirror to the future: Best-effort approximate complex event matching with insufficient resources*, PVLDB, 10 (2016), pp. 397–408.

[121] W. LIN, H. FAN, Z. QIAN, J. XU, S. YANG, J. ZHOU, AND L. ZHOU, *Streamscope: Continuous reliable distributed processing of big data streams*, in 13th USENIX Symposium on Networked Systems Design and Implementation, NSDI 2016, Santa Clara, CA, USA, March 16-18, 2016, K. J. Argyraki and R. Isaacs, eds., USENIX Association, 2016, pp. 439–453.

[122] B. LIU, Y. ZHU, AND E. A. RUNDENSTEINER, *Run-time operator state spilling for memory intensive long-running queries*, in Proceedings of the ACM SIGMOD International Conference on Management of Data, Chicago, Illinois, USA, June 27-29, 2006, S. Chaudhuri, V. Hristidis, and N. Polyzotis, eds., ACM, 2006, pp. 347–358.

[123] M. LIU, E. A. RUNDENSTEINER, K. GREENFIELD, C. GUPTA, S. WANG, I. ARI, AND A. MEHTA, *E-cube: multi-dimensional event sequence analysis using hierarchical pattern query sharing*, in Proceedings of the ACM SIGMOD International Conference on Management of Data, SIGMOD 2011, Athens, Greece, June 12-16, 2011, T. K. Sellis, R. J. Miller, A. Kementsietsidis, and Y. Velegrakis, eds., ACM, 2011, pp. 889–900.

[124] X. LIU AND R. BUYYA, *Resource management and scheduling in distributed stream processing systems: A taxonomy, review, and future directions*, ACM Comput. Surv., 53 (2020), pp. 50:1–50:41.

[125] R. MAYER, C. MAYER, M. A. TARIQ, AND K. ROTHERMEL, *Graphcep: real-time data analytics using parallel complex event and graph processing*, in Proceedings of the 10th ACM International Conference on Distributed and Event-based Systems, DEBS '16, Irvine, CA, USA, June 20 - 24, 2016, A. Gal, M. Weidlich, V. Kalogeraki, and N. Venkasubramanian, eds., ACM, 2016, pp. 309–316.

[126] R. MAYER, M. A. TARIQ, AND K. ROTHERMEL, *Minimizing communication overhead in window-based parallel complex event processing*, in Proceedings of the 11th ACM International Conference on Distributed and Event-based Systems, DEBS 2017, Barcelona, Spain, June 19-23, 2017, 2017, pp. 54–65.

[127] Y. MEI AND S. MADDEN, *Zstream: a cost-based query processor for adaptively detecting composite events*, in Proceedings of the ACM SIGMOD International Conference on Management of Data, SIGMOD 2009, Providence, Rhode Island, USA, June 29 - July 2, 2009, U. Çetintemel, S. B. Zdonik, D. Kossmann, and N. Tatbul, eds., ACM, 2009, pp. 193–206.

[128] P. MENON, A. PAVLO, AND T. C. MOWRY, *Relaxed operator fusion for in-memory databases: Making compilation, vectorization, and prefetching work together at last*, Proc. VLDB Endow., 11 (2017), pp. 1–13.

[129] H. MIAO, M. JEON, G. PEKHIMENKO, K. S. MCKINLEY, AND F. X. LIN, *Streambox-hbm: Stream analytics on high bandwidth hybrid memory*, in Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS 2019, Providence, RI, USA, April 13-17, 2019, I. Bahar, M. Herlihy, E. Witchel, and A. R. Lebeck, eds., ACM, 2019, pp. 167–181.

[130] H. MIAO, H. PARK, M. JEON, G. PEKHIMENKO, K. S. MCKINLEY, AND F. X. LIN, *Streambox: Modern stream processing on a multicore machine*, in 2017 USENIX Annual Technical Conference, USENIX ATC 2017, Santa Clara, CA, USA, July 12-14, 2017, D. D. Silva and B. Ford, eds., USENIX Association, 2017, pp. 617–629.

[131] S. MITTAL, *A survey of recent prefetching techniques for processor caches*, ACM Comput. Surv., 49 (2016), pp. 35:1–35:35.

[132] B. MOZAFARI, K. ZENG, AND C. ZANIOLO, *High-performance complex event processing over XML streams*, in Proceedings of the ACM SIGMOD International Conference on Management of Data, SIGMOD 2012, Scottsdale, AZ, USA, May 20-24, 2012, K. S.

Candan, Y. Chen, R. T. Snodgrass, L. Gravano, and A. Fuxman, eds., ACM, 2012, pp. 253–264.

[133] D. MURRAY, J. SIMSA, A. KLIMOVIC, AND I. INDYK, *tf.data: A machine learning data processing framework*, ArXiv, abs/2101.12127 (2021).

[134] H. NAJDATAEI, Y. NIKOLAKOPOULOS, M. PAPATRIANTAFILOU, P. TSIGAS, AND V. GULISANO, *STRETCH: scalable and elastic deterministic streaming analysis with virtual shared-nothing parallelism*, in Proceedings of the 13th ACM International Conference on Distributed and Event-based Systems, DEBS 2019, Darmstadt, Germany, June 24-28, 2019, ACM, 2019, pp. 7–18.

[135] B. NICOLAE AND F. CAPPELLO, *Ai-ckpt: leveraging memory access patterns for adaptive asynchronous incremental checkpointing*, in The 22nd International Symposium on High-Performance Parallel and Distributed Computing, HPDC'13, New York, NY, USA - June 17 - 21, 2013, M. Parashar, J. B. Weissman, D. H. J. Epema, and R. J. O. Figueiredo, eds., ACM, 2013, pp. 155–166.

[136] NOAA, *GOES-16: A GAME-CHANGER FOR FIGHTING DEADLY WILDFIRES*. https://www.goes-r.gov/featureStories/goes16Wildfires.html, 2018. Last access: 05/08/21.

[137] NOAA, *GOES-R Fire Detection and Characterization*. https://www.goes-r.gov/education/docs/fs_fire.pdf, 2019. Last access: 05/08/21.

[138] S. A. NOGHABI, K. PARAMASIVAM, Y. PAN, N. RAMESH, J. BRINGHURST, I. GUPTA, AND R. H. CAMPBELL, *Stateful scalable stream processing at linkedin*, Proc. VLDB Endow., 10 (2017), pp. 1634–1645.

[139] D. OLTEANU, T. KIESLING, AND F. BRY, *An evaluation of regular path expressions with qualifiers against XML streams*, in Proceedings of the 19th International Conference on Data Engineering, March 5-8, 2003, Bangalore, India, U. Dayal, K. Ramamritham, and T. M. Vijayaraman, eds., IEEE Computer Society, 2003, pp. 702–704.

[140] ORACLE, *Overview of Oracle CEP*. https://docs.oracle.com/cd/E17904_01/doc.1111/e14476/overview.htm#CEPGS106, 2021. Last access: 05/08/21.

[141] F. ÖZCAN, G. KOUTRIKA, AND S. MADDEN, eds., *Proceedings of the 2016 International Conference on Management of Data, SIGMOD Conference 2016, San Francisco, CA, USA, June 26 - July 01, 2016*, ACM, 2016.

[142] PACIFIC NORTHWEST NATIONAL LABORATORY, *Transactive system*. https://tinyurl.com/y3d5xxcf, Dec. 2017.

[143] N. PANSARE, V. R. BORKAR, C. JERMAINE, AND T. CONDIE, *Online aggregation for large mapreduce jobs*, PVLDB, 4 (2011), pp. 1135–1145.

[144] Y. PARK, B. MOZAFARI, J. SORENSON, AND J. WANG, *Verdictdb: Universalizing approximate query processing*, in Proceedings of the 2018 International Conference on Management of Data, SIGMOD Conference 2018, Houston, TX, USA, June 10-15, 2018, G. Das, C. M. Jermaine, and P. A. Bernstein, eds., ACM, 2018, pp. 1461–1476.

[145] F. PENG AND S. S. CHAWATHE, *Xpath queries on streaming data*, in Proceedings of the 2003 ACM SIGMOD International Conference on Management of Data, San Diego, California, USA, June 9-12, 2003, A. Y. Halevy, Z. G. Ives, and A. Doan, eds., ACM, 2003, pp. 431–442.

[146] T. B. G. PEREZ, X. ZHOU, AND D. CHENG, *Reference-distance eviction and prefetching for cache management in spark*, in Proceedings of the 47th International Conference on Parallel Processing, ICPP 2018, Eugene, OR, USA, August 13-16, 2018, ACM, 2018, pp. 88:1–88:10.

[147] U. PFERSCHY, *Dynamic programming revisited: Improving knapsack algorithms*, Computing, 63 (1999), pp. 419–430.

[148] T. N. PHAM, P. K. CHRYSANTHIS, AND A. LABRINIDIS, *Avoiding class warfare: Managing continuous queries with differentiated classes of service*, The VLDB Journal, 25 (2016), pp. 197–221.

[149] O. POPPE, C. LEI, S. AHMED, AND E. A. RUNDENSTEINER, *Complete event trend detection in high-rate event streams*, in Proceedings of the 2017 ACM International Conference on Management of Data, SIGMOD Conference 2017, Chicago, IL, USA, May 14-19, 2017, S. Salihoglu, W. Zhou, R. Chirkova, J. Yang, and D. Suciu, eds., ACM, 2017, pp. 109–124.

[150] O. POPPE, C. LEI, E. A. RUNDENSTEINER, AND D. MAIER, *GRETA: graph-based real-time event trend aggregation*, Proc. VLDB Endow., 11 (2017), pp. 80–92.

[151] O. POPPE, C. LEI, E. A. RUNDENSTEINER, AND D. MAIER, *Event trend aggregation under rich event matching semantics*, in Proceedings of the 2019 International Conference on Management of Data, SIGMOD Conference 2019, Amsterdam, The Netherlands, June 30 - July 5, 2019, P. A. Boncz, S. Manegold, A. Ailamaki, A. Deshpande, and T. Kraska, eds., ACM, 2019, pp. 555–572.

[152] O. POPPE, A. ROZET, C. LEI, E. A. RUNDENSTEINER, AND D. MAIER, *Sharon: Shared online event sequence aggregation*, in 34th IEEE International Conference on Data Engineering, ICDE 2018, Paris, France, April 16-19, 2018, IEEE Computer Society, 2018, pp. 737–748.

[153] Z. QIAN, Y. HE, C. SU, Z. WU, H. ZHU, T. ZHANG, L. ZHOU, Y. YU, AND Z. ZHANG, *Timestream: reliable stream computation in the cloud*, in Eighth Eurosys Conference 2013, EuroSys '13, Prague, Czech Republic, April 14-17, 2013, Z. Hanzálek, H. Härtig, M. Castro, and M. F. Kaashoek, eds., ACM, 2013, pp. 1–14.

[154] J. QUIRÓS-TORTÓS, L. OCHOA, AND T. BUTLER, *How electric vehicles and the grid work together: Lessons learned from one of the largest EV trials in the world*, IEEE Power Energy Mag., 16 (2018), pp. 64–76.

[155] K. RAMACHANDRA AND S. SUDARSHAN, *Holistic optimization by prefetching query results*, in Proceedings of the ACM SIGMOD International Conference on Management of Data, SIGMOD 2012, Scottsdale, AZ, USA, May 20-24, 2012, K. S. Candan, Y. Chen, R. T. Snodgrass, L. Gravano, and A. Fuxman, eds., ACM, 2012, pp. 133–144.

[156] G. RAMAN, B. ALSHEBLI, M. WANIEK, T. RAHWAN, AND J. C.-H. PENG, *How weaponizing disinformation can bring down a city's power grid*, PLOS ONE, 15 (2020), p. e0236517.

[157] G. RAMAN, J. C.-H. PENG, AND T. RAHWAN, *Manipulating residents' behavior to attack the urban power distribution system*, IEEE Trans. Ind. Informat., 15 (2019), pp. 5575–5587.

[158] G. RAMAN, J. C.-H. PENG, B. ZHAO, AND M. WEIDLICH, *Dynamic decision making for demand response through adaptive event stream monitoring*, in 2019 IEEE Power and Energy Society General Meeting (PESGM), IEEE, 2019, pp. 1–5.

[159] M. RAY, C. LEI, AND E. A. RUNDENSTEINER, *Scalable pattern sharing on event streams*, in Özcan et al. [141], pp. 495–510.

[160] C. REISS, J. WILKES, AND J. L. HELLERSTEIN, *Google cluster-usage traces: format + schema*, technical report, Google Inc., Mountain View, CA, USA, Nov. 2011.
Revised 2014-11-17 for version 2.1. Posted at https://github.com/google/cluster-data.

[161] K. REN, T. DIAMOND, D. J. ABADI, AND A. THOMSON, *Low-overhead asynchronous checkpointing in main-memory database systems*, in Proceedings of the 2016 International Conference on Management of Data, SIGMOD Conference 2016, San Francisco, CA, USA, June 26 - July 01, 2016, F. Özcan, G. Koutrika, and S. Madden, eds., ACM, 2016, pp. 1539–1551.

[162] H. A. RINGSBERG AND V. MIRZABEIKI, *Effects on logistic operations from rfid-and epcis-enabled traceability*, British Food Journal, (2014).

[163] N. RIVETTI, Y. BUSNEL, AND L. QUERZONI, *Load-aware shedding in stream processing systems*, in Proceedings of the 10th ACM International Conference on Distributed and Event-based Systems, DEBS '16, Irvine, CA, USA, June 20 - 24, 2016, A. Gal, M. Weidlich, V. Kalogeraki, and N. Venkasubramanian, eds., ACM, 2016, pp. 61–68.

[164] ROCKSDB, .
http://rocksdb.org/, 2021.
Last access: 05/08/21.

[165] H. RÖGER AND R. MAYER, *A comprehensive survey on parallelization and elasticity in stream processing*, ACM Comput. Surv., 52 (2019), pp. 36:1–36:37.

[166] P. V. ROY AND S. HARIDI, *Concepts, Techniques, and Models of Computer Programming*, MIT Press, 2004.

[167] E. A. RUNDENSTEINER, L. DING, T. M. SUTHERLAND, Y. ZHU, B. PIELECH, AND N. K. MEHTA, *CAPE: continuous query engine with heterogeneous-grained adaptivity*, in (e)Proceedings of the Thirtieth International Conference on Very Large Data Bases, VLDB 2004, Toronto, Canada, August 31 - September 3 2004, M. A. Nascimento, M. T. Özsu, D. Kossmann, R. J. Miller, J. A. Blakeley, and K. B. Schiefer, eds., Morgan Kaufmann, 2004, pp. 1353–1356.

[168] M. J. SAX, *Apache kafka*, in Encyclopedia of Big Data Technologies, S. Sakr and A. Y. Zomaya, eds., Springer, 2019.

[169] S. SCHNEIDER, M. HIRZEL, B. GEDIK, AND K. WU, *Auto-parallelizing stateful distributed streaming applications*, in International Conference on Parallel Architectures and Compilation Techniques, PACT '12, Minneapolis, MN, USA - September 19 - 23, 2012, P. Yew, S. Cho, L. DeRose, and D. J. Lilja, eds., ACM, 2012, pp. 53–64.

[170] N. P. SCHULTZ-MØLLER, M. MIGLIAVACCA, AND P. R. PIETZUCH, *Distributed complex event processing with query rewriting*, in Proceedings of the Third ACM International Conference on Distributed Event-Based Systems, DEBS 2009, Nashville, Tennessee, USA, July 6-9, 2009, A. S. Gokhale and D. C. Schmidt, eds., ACM, 2009.

[171] M. A. SHAH, J. M. HELLERSTEIN, AND E. A. BREWER, *Highly-available, fault-tolerant, parallel dataflows*, in Proceedings of the ACM SIGMOD International Conference on Management of Data, Paris, France, June 13-18, 2004, G. Weikum, A. C. König, and S. Deßloch, eds., ACM, 2004, pp. 827–838.

[172] D. SIDLER, Z. ISTVÁN, M. OWAIDA, AND G. ALONSO, *Accelerating pattern matching queries in hybrid CPU-FPGA architectures*, in Proceedings of the 2017 ACM International Conference on Management of Data, SIGMOD Conference 2017, Chicago, IL, USA, May 14-19, 2017, S. Salihoglu, W. Zhou, R. Chirkova, J. Yang, and D. Suciu, eds., ACM, 2017, pp. 403–415.

[173] A. SLO, S. BHOWMIK, A. FLAIG, AND K. ROTHERMEL, *pspice: Partial match shedding for complex event processing*, in 2019 IEEE International Conference on Big Data (IEEE BigData), Los Angeles, CA, USA, December 9-12, 2019, C. Baru, J. Huan, L. Khan, X. Hu, R. Ak, Y. Tian, R. S. Barga, C. Zaniolo, K. Lee, and Y. F. Ye, eds., IEEE, 2019, pp. 372–382.

[174] A. SLO, S. BHOWMIK, AND K. ROTHERMEL, *espice: Probabilistic load shedding from input event streams in complex event processing*, in Proceedings of the 20th International Middleware Conference, Middleware 2019, Davis, CA, USA, December 9-13, 2019, ACM, 2019, pp. 215–227.

[175] A. SLO, S. BHOWMIK, AND K. ROTHERMEL, *hspice: state-aware event shedding in complex event processing*, in DEBS '20: The 14th ACM International Conference on Distributed and Event-based Systems, Montreal, Quebec, Canada, July 13-17, 2020, J. Gascon-Samson, K. Zhang, K. Daudjee, and B. Kemme, eds., ACM, 2020, pp. 109–120.

[176] A. J. SMITH, *Sequentiality and prefetching in database systems*, ACM Trans. Database Syst., 3 (1978), pp. 223–247.

[177] K. SUBBARAO ET AL., *Transactive control and coordination of distributed assets for ancillary services*, tech. rep., Pacific Northwest National Laboratory, 2013.

[178] N. TATBUL, U. ÇETINTEMEL, S. B. ZDONIK, M. CHERNIACK, AND M. STONEBRAKER, *Load shedding in a data stream manager*, in VLDB 2003, Proceedings of 29th International Conference on Very Large Data Bases, September 9-12, 2003, Berlin, Germany, J. C. Freytag, P. C. Lockemann, S. Abiteboul, M. J. Carey, P. G. Selinger, and A. Heuer, eds., Morgan Kaufmann, 2003, pp. 309–320.

[179] D. B. TERRY, D. GOLDBERG, D. A. NICHOLS, AND B. M. OKI, *Continuous queries over append-only databases*, in Proceedings of the 1992 ACM SIGMOD International Conference on Management of Data, San Diego, California, USA, June 2-5, 1992, M. Stonebraker, ed., ACM Press, 1992, pp. 321–330.

[180] G. THEODORAKIS, A. KOLIOUSIS, P. R. PIETZUCH, AND H. PIRK, *Lightsaber: Efficient window aggregation on multi-core processors*, in Proceedings of the 2020 International

Conference on Management of Data, SIGMOD Conference 2020, online conference [Portland, OR, USA], June 14-19, 2020, D. Maier, R. Pottinger, A. Doan, W. Tan, A. Alawini, and H. Q. Ngo, eds., ACM, 2020, pp. 2505–2521.

[181] R. TIBSHIRANI, G. WALTHER, AND T. HASTIE, *Estimating the number of clusters in a dataset via the gap statistic*, J. R. Statist. Soc. B, 63 (2001), pp. 411–423.

[182] Q. TO, J. SOTO, AND V. MARKL, *A survey of state management in big data processing systems*, VLDB J., 27 (2018), pp. 847–872.

[183] A. TOSHNIWAL, S. TANEJA, A. SHUKLA, K. RAMASAMY, J. M. PATEL, S. KULKARNI, J. JACKSON, K. GADE, M. FU, J. DONHAM, N. BHAGAT, S. MITTAL, AND D. V. RYABOY, *Storm@twitter*, in International Conference on Management of Data, SIGMOD 2014, Snowbird, UT, USA, June 22-27, 2014, C. E. Dyreson, F. Li, and M. T. Özsu, eds., ACM, 2014, pp. 147–156.

[184] S. P. VANDERWIEL AND D. J. LILJA, *Data prefetch mechanisms*, ACM Comput. Surv., 32 (2000), pp. 174–199.

[185] H. WEDEKIND AND G. ZÖRNTLEIN, *Prefetching in realtime database applications*, in Proceedings of the 1986 ACM SIGMOD International Conference on Management of Data, Washington, DC, USA, May 28-30, 1986, C. Zaniolo, ed., ACM Press, 1986, pp. 215–226.

[186] M. WEI, E. A. RUNDENSTEINER, AND M. MANI, *Achieving high output quality under limited resources through structure-based spilling in XML streams*, PVLDB, 3 (2010), pp. 1267–1278.

[187] M. WEIDLICH, H. ZIEKOW, A. GAL, J. MENDLING, AND M. WESKE, *Optimizing event pattern matching using business process models*, IEEE Trans. Knowl. Data Eng., 26 (2014), pp. 2759–2773.

[188] L. WONG, N. S. ARORA, L. GAO, T. HOANG, AND J. WU, *Oracle streams: A high performance implementation for near real time asynchronous replication*, in Proceedings of the 25th International Conference on Data Engineering, ICDE 2009, March 29 2009 - April 2 2009, Shanghai, China, Y. E. Ioannidis, D. L. Lee, and R. T. Ng, eds., IEEE Computer Society, 2009, pp. 1363–1374.

[189] E. WU, Y. DIAO, AND S. RIZVI, *High-performance complex event processing over streams*, in Proceedings of the ACM SIGMOD International Conference on Management of Data, Chicago, Illinois, USA, June 27-29, 2006, S. Chaudhuri, V. Hristidis, and N. Polyzotis, eds., ACM, 2006, pp. 407–418.

[190] S. WU, B. C. OOI, AND K. TAN, *Continuous sampling for online aggregation over multiple queries*, in Proceedings of the ACM SIGMOD International Conference on Management of Data, SIGMOD 2010, Indianapolis, Indiana, USA, June 6-10, 2010, A. K. Elmagarmid and D. Agrawal, eds., ACM, 2010, pp. 651–662.

[191] Y. WU AND K. TAN, *Chronostream: Elastic stateful stream computation in the cloud*, in 31st IEEE International Conference on Data Engineering, ICDE 2015, Seoul, South Korea, April 13-17, 2015, J. Gehrke, W. Lehner, K. Shim, S. K. Cha, and G. M. Lohman, eds., IEEE Computer Society, 2015, pp. 723–734.

[192] Y. YU, W. WANG, J. ZHANG, AND K. B. LETAIEF, *LRC: dependency-aware cache management for data analytics clusters*, in 2017 IEEE Conference on Computer Communications, INFOCOM 2017, Atlanta, GA, USA, May 1-4, 2017, IEEE, 2017, pp. 1–9.

[193] M. ZAHARIA, M. CHOWDHURY, T. DAS, A. DAVE, J. MA, M. MCCAULY, M. J. FRANKLIN, S. SHENKER, AND I. STOICA, *Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing*, in Proceedings of the 9th USENIX Symposium on Networked Systems Design and Implementation, NSDI 2012, San Jose, CA, USA, April 25-27, 2012, S. D. Gribble and D. Katabi, eds., USENIX Association, 2012, pp. 15–28.

[194] H. ZHANG, G. CHEN, B. C. OOI, K. TAN, AND M. ZHANG, *In-memory big data management and processing: A survey*, IEEE Trans. Knowl. Data Eng., 27 (2015), pp. 1920–1948.

[195] H. ZHANG, Y. DIAO, AND N. IMMERMAN, *Recognizing patterns in streams with imprecise timestamps*, Proc. VLDB Endow., 3 (2010), pp. 244–255.

[196] H. ZHANG, Y. DIAO, AND N. IMMERMAN, *On complexity and optimization of expensive queries in complex event processing*, in International Conference on Management of Data, SIGMOD 2014, Snowbird, UT, USA, June 22-27, 2014, C. E. Dyreson, F. Li, and M. T. Özsu, eds., ACM, 2014, pp. 217–228.

[197] S. ZHANG, J. HE, A. C. ZHOU, AND B. HE, *Briskstream: Scaling data stream processing on shared-memory multicore architectures*, in Proceedings of the 2019 International Conference on Management of Data, SIGMOD Conference 2019, Amsterdam, The Netherlands, June 30 - July 5, 2019, P. A. Boncz, S. Manegold, A. Ailamaki, A. Deshpande, and T. Kraska, eds., ACM, 2019, pp. 705–722.

[198] S. ZHANG, H. T. VO, D. DAHLMEIER, AND B. HE, *Multi-query optimization for complex event processing in SAP ESP*, in 33rd IEEE International Conference on Data Engineering, ICDE 2017, San Diego, CA, USA, April 19-22, 2017, IEEE Computer Society, 2017, pp. 1213–1224.

[199] S. ZHANG, Y. WU, F. ZHANG, AND B. HE, *Towards concurrent stateful stream processing on multicore processors*, in 36th IEEE International Conference on Data Engineering, ICDE 2020, Dallas, TX, USA, April 20-24, 2020, IEEE, 2020, pp. 1537–1548.

[200] Y. ZHANG AND F. MUELLER, *Gstream: A general-purpose data streaming framework on GPU clusters*, in International Conference on Parallel Processing, ICPP 2011, Taipei, Taiwan, September 13-16, 2011, G. R. Gao and Y. Tseng, eds., IEEE Computer Society, 2011, pp. 245–254.

[201] B. ZHAO, *Complex event processing under constrained resources by state-based load shedding*, in 34th IEEE International Conference on Data Engineering, ICDE 2018, Paris, France, April 16-19, 2018, IEEE Computer Society, 2018, pp. 1699–1703.

[202] B. ZHAO, Q. V. H. NGUYEN, AND M. WEIDLICH, *Load shedding for complex event processing: Input-based and state-based techniques*, in 36th IEEE International Conference on Data Engineering, ICDE 2020, Dallas, TX, USA, April 20-24, 2020, IEEE, 2020, pp. 1093–1104.

[203] B. ZHAO, H. VAN DER AA, T. T. NGUYEN, Q. V. H. NGUYEN, AND M. WEIDLICH, *EIRES: efficient integration of remote data in event stream processing*, in SIGMOD '21: International Conference on Management of Data, Virtual Event, China, June 20-25, 2021, G. Li, Z. Li, S. Idreos, and D. Srivastava, eds., ACM, 2021, pp. 2128–2141.

[204] H. ZIEKOW, B. FABIAN, AND C. MÜLLER, *High-speed access to RFID data: Meeting real-time requirements in distributed value chains*, in Proceedings On the Move to Meaningful Internet Systems: OTM 2009 Workshops, Vilamoura, Portugal, November 1-6, 2009., R. Meersman, P. Herrero, and T. S. Dillon, eds., vol. 5872 of Lecture Notes in Computer Science, Springer, 2009, pp. 142–151.