Griffith UNIVERSITY
Queensland, Australia

EIRES: Efficient Integration of Remote Datain Event Stream Processing

Author

Zhao, Bo, van der Aa, Han, Nguyen, Thanh Tam, Nguyen, Quoc Viet Hung, Weidlich, Matthias

Published

2021

Conference Title

Proceedings of the 2021 International Conference on Management of Data

Version

Accepted Manuscript (AM)

DOI

https://doi.org/10.1145/3448016.3457304

Copyright Statement

© ACM, 2021. This is the author's version of the work. It is posted here by permission of ACM for your personal use. Not for redistribution. The definitive version was published in SIGMOD/ PODS '21: Proceedings of the 2021 International Conference on Management of Data, ISBN: 978-1-4503-8343-1, https://doi.org/10.1145/3448016.3457304

Downloaded from

http://hdl.handle.net/10072/405691

Griffith Research Online

https://research-repository.griffith.edu.au

© ACM, 2021. This is the author's version of the work. It is posted here by permission of ACM for your personal use. Not for redistribution. The definitive version was published in SIGMOD/PODS '21: Proceedings of the 2021 International Conference on Management of Data, ISBN: 978-1-4503-8343-1, https://doi.org/10.1145/3448016.3457304

EIRES: Efficient Integration of Remote Data in Event Stream Processing

Bo Zhao[1], Han van der Aa[2], Thanh Tam Nguyen [3], Quoc Viet Hung Nguyen[4], Matthias Weidlich[1]

[1]Humboldt-Universität zu Berlin [2]Universität Mannheim [3]Leibniz Universität Hannover [4]Griffith University

bo.zhao@hu-berlin.de; han@informatik.uni-mannheim.de; tamnguyen@l3s.de; quocviethung.nguyen@griffith.edu.au; weidlima@hu-berlin

ABSTRACT

To support reactive and predictive applications, complex event processing (CEP) systems detect patterns in event streams based on predefined queries. To determine the events that constitute a query match, their payload data may need to be assessed together with data from remote sources. 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 CEP system degrades drastically.

To tackle these issues, we present EIRES, a framework for efficient integration of static data from remote sources in CEP. It employs 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 that queries remote data based on anticipated use and (ii) lazy evaluation that postpones the event selection based on remote data without interrupting the stream processing. Our experiments indicate that the combination of these strategies improves the latency of query evaluation by up to 3,752× for synthetic data and 47× for real-world data.

CCS CONCEPTS

• **Information systems** → **Data streams**.

KEYWORDS

complex event processing; data prefetching; query result caching

ACM Reference Format:

Bo Zhao[1], Han van der Aa[2], Thanh Tam Nguyen [3], Quoc Viet Hung Nguyen[4], Matthias Weidlich[1]. 2021. EIRES: Efficient Integration of Remote Data in Event Stream Processing. In *Proceedings of the 2021 International Conference on Management of Data (SIGMOD '21), June 20–25, 2021, Virtual Event, China.* ACM, New York, NY, USA, 14 pages. https://doi.org/10.1145/3448016.3457304

1 INTRODUCTION

Complex event processing (CEP) systems evaluate queries over continuous streams of events to detect patterns [1]. They provide the ability to identify a situation of interest with low latency, thereby supporting reactive and predictive applications [2]. A CEP query describes an event pattern as the joint occurrence of events of particular types, potentially in a specific ordering, correlated through predicates over their data payload and a time window. By evaluating such a query in an online manner, an event pattern is detected in near-real-time, as soon as it materializes.

The evaluation of CEP queries is computationally challenging. A CEP system needs to maintain a set of partial matches per query [3–5], which may grow exponentially in the number of processed events. Hence, common evaluation algorithms show an exponential (worst-case) time complexity [6]. Recognising this, optimization techniques based on state sharing [6, 7], query rewriting [8, 9], or load shedding [10, 11] have been proposed. These optimizations assume that the occurrence of a pattern is fully characterised by the events and their payload data. Yet, many applications require the combination of the events' payload with data from remote sources to determine whether a partial match shall be processed further.

Consider the detection of fraud in credit card usage [12]. Here, events denote financial transactions or indicate status changes, such as an increase in spending limit. To block malicious transactions, CEP queries aim to identify patterns of suspicious use. Such patterns must be detected under tight latency bounds since, e.g., a credit card transaction needs to be cleared within 25ms [13]. Yet the detection of suspicious patterns often depends on contextual information, such as a user's spending history, transactional volume at a specific location, or the behaviour of other users. Since this information must be retrieved from remote data sources, incorporating it while still meeting the latency bounds becomes highly challenging.

A naive integration of remote data reduces the performance of a CEP system drastically. Fetching data only once it is needed for query evaluation, see Figure 1 (top), interrupts the processing of the stream. As we later demonstrate in our experiments, such an interruption 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 CEP systems achieve microsecond latencies and many applications enforce tight latency bounds. Although 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 partial matches, thereby increasing the processing latency.

In this paper, we present EIRES, a framework to address use cases such as the above one through an efficient integration of remote data in event stream processing. As shown in Figure 1, our idea is to decouple (i) the fetching of remote data from (ii) its use in query evaluation. While under a naive model, data is fetched once it is needed and then used immediately, EIRES employs a cache to decouple both operations.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
SIGMOD '21, June 20–25, 2021, Virtual Event, China
© 2021 Association for Computing Machinery.
ACM ISBN 978-1-4503-8343-1/21/06…$15.00
https://doi.org/10.1145/3448016.3457304

Preprint

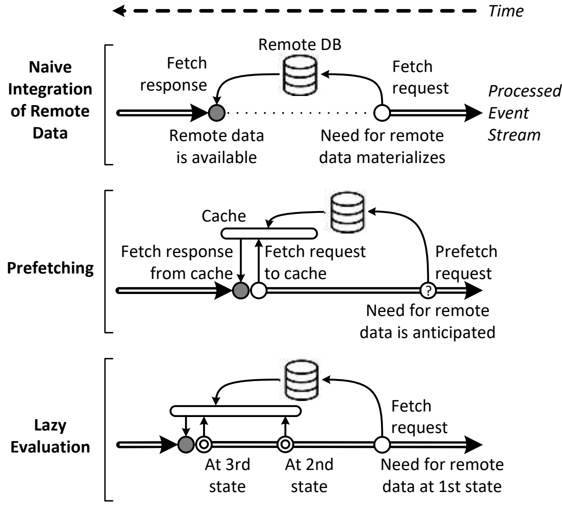


Figure 1: Strategies to integrate remote data in event stream processing: Naive integration; prefetching based on anticipated use; lazy evaluation once data is available.

Fetching may happen *before* the need for remote data materializes (i.e., *prefetching*) and the evaluation of partial matches based on it may be postponed until *after* the remote data is available (i.e., *lazy evaluation*). While both strategies hide the data transmission latency, they also come with side effects. Prefetching may fill the cache with superfluous data, while lazy evaluation may suffer from the mentioned 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.

Our contributions are summarized as follows:

- We introduce the EIRES framework. It shows how the use of a cache facilitates prefetching and lazy evaluation when integrating remote data in event stream processing.
- To instantiate the framework, we present a cost model to assess the utility of remote data elements. We show how the model may be approximated to evaluate it efficiently.
- Using the cost model, we propose mechanisms to decide when and which data elements to prefetch, and when to postpone the evaluation if remote data is not yet available.
- We elaborate on strategies for cache management, based on simple policies and based on the proposed cost model.

We evaluated our approach using both synthetic and real-world datasets. Compared to the best baseline approach to integrate remote data, EIRES improves the latency of query evaluation by up to 3,752× for synthetic data and 47× for real-world data. We further present a sensitivity analysis to shed light on the influence of various parameters of the problem setting on the obtained results.

In the remainder, §2 presents a formal problem statement. The EIRES framework is introduced in §3, before instantiating its main components (§4-§6). We give evaluation results in §7 and discuss related work in §8, before concluding the paper in §9.

2 PROBLEM FORMULATION

This section first introduces a formal model for CEP with remote data (§2.1), before we introduce the problem of latency minimization for this setting (§2.2), which is addressed in the remainder.

2.1 A Model for CEP with Remote Data

Event streams. We adopt a common, relational model of event streams. An event is an occurrence of interest at a specific point in time, which is instantaneous, unique, and atomic. Its structure is defined by a schema, as was first proposed in traditional data stream processing [14]. An event schema is given by a sequence of attributes $A = \langle A_1, \ldots, A_n \rangle$, where each attribute is assigned a primitive data type. Then, an event $e = \langle a_1, \ldots, a_n \rangle$ is an instance of this schema, where a_i is the value of attribute A_i. Moreover, events carry timestamps from a discrete, totally-ordered domain, defined by $\mathbb{N}$. The timestamp of an event e is referred to by $e.t$. Without loss of generality, we assume all events to have the same schema. Note that different types of events, as used later in our running example, are incorporated by predicates over a distinguished attribute.

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 two events e_i and e_j, it holds that $i < j$ implies $e_i.t \leq e_j.t$. We further define the notion of a finite stream prefix, up to index k, as $S(..k) = \langle e_1, \ldots, e_k \rangle$.

Remote data. The evaluation of a CEP query may require data from remote sources. Here, we focus on *what* data elements are fetched from such 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 $\mathcal{D} = \{d_1, \ldots, d_n\}$ to capture the remote data elements. Moreover, we write $|d| \in \mathbb{N}$ for the size of a data element.

Notably, data models are hierarchical in many applications, which means that there exists a containment relation between data elements. Therefore, we also consider a partial function $\rho : \mathcal{D} \nrightarrow \mathcal{D}$ 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., $|d| = \sum_{d' \in \mathcal{D}, \rho(d')=d} |d'|$.

Since fetching data elements from different sources can result in different transmission latencies, we here assume that latency is monitored per data element, and denoted as $\boldsymbol{\ell}_{remote}(d)$.

CEP queries. Common query languages describe event patterns through operators (e.g., sequence), conditions based on data of the events' payload or from remote sources, and a time window [1, 15].

For the use case of financial fraud detection, an example query is shown in Listing 1, using a syntax similar to the SASE language [3]. It abstracts from the specific queries sent to remote databases, but highlights which predicates involve remote data and which payload data of events is used for the respective lookup. In this scenario, remote data captures the known locations of credit card usage per client, the card limits, and the set of pre-authorized clients for an organization. The latter is organized hierarchically, i.e., it can be fetched per credit card, per user, or for the whole organization.

The query is triggered by a transaction event `t1` (of type `T`) of high volume (`>10k`). A suspicious pattern may emerge in two ways based on other events recorded for the same credit card (`SAME[cc]`):

First, the event may be followed by a denied transaction `d` (type `D`) and another high-volume transaction `t2` that occurred at a location that both differs (`t1.loc <> t2.loc`) and is not in the set of known locations for the user, which is fetched from a remote database (`REMOTE[t1.user]`).

Preprint

```
SEQ(T t1, (SEQ(D d, T t2) OR SEQ(L l, T t3))
WHERE SAME[cc] AND t1.vol > 10k AND t2.vol > 10k
AND t1.loc <> t2.loc AND (t2.loc NOT IN REMOTE[t1.user])
AND l.limit > REMOTE[t1.org]
AND t3.vol > 50k AND (t3.ben NOT IN REMOTE[t3.org])
WITHIN 5min
```

Listing 1: Query to detect fraudulent transactions.

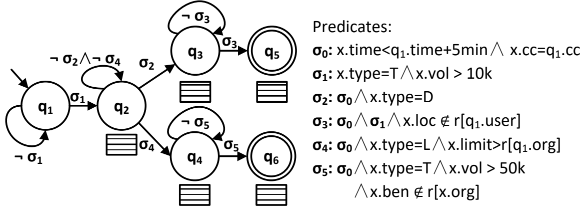


Figure 2: Evaluation model for the query of Listing 1.

Second, the initial event may be followed by a change in the spending limit (event `l` of type `L`), where the new limit is larger than the maximum limit of all credit cards within the same organization (queried from a remote database, `REMOTE[t1.org]`). Afterwards, another transaction is observed with a very high volume (`>50k`), for which the beneficiary (`t3.ben`) is not in the set of pre-authorized clients, as stored at a remote data source (`REMOTE[t3.org]`).

To evaluate such CEP queries, various formalisms have been proposed, many of them being based on automata [3, 16, 17]. Figure 2 exemplifies an evaluation model for the query of Listing 1. It defines states and state transitions to describe how a CEP system constructs matches of the query when processing a stream event by event. To this end, it maintains a set of partial matches (i.e., partial runs of the automaton). However, due to non-determinism in the automaton, the number of partial matches is potentially exponential in the number of processed events. Given the current set of partial matches, a CEP system 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.

The evolution of partial matches is influenced by the time window, the predicates, and event processing policies of the query. Transitions in an automata are guarded and link the current input event (denoted by x), events of partial matches (denoted by state identifiers, such as q_1), and data from remote sources (denoted by a parametrised variable, such as $r[q_1.user]$). For example, predicate σ_3 in Figure 2 checks the time window, $x.time < q_1.time{+}5min$ (i.e., σ_0), a condition over the event's payload data, $x.type = T \wedge x.vol > 10k$ (i.e., σ_1), and a condition using remote data, $x.loc \notin r[q_1.user]$.

Event processing policies fine-tune the consumption and selection of events [18]. Specifically, we consider a *greedy* policy, under which a partial match that, based on the transition guards in the automaton, can be extended with an input event from the 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 predicates. This semantics is known as *unconstraint* [19] or *skip-till-any-match* [3].

We also consider a *non-greedy* policy, also known as *continuous* or *skip-till-next-match*. Here, the partial match would only be extended, i.e., only events that do not satisfy the predicates are skipped to always select the *next* event that satisfies them.

Table 1: Notations.

Notation	Explanation
$e = \langle a_1, \ldots, a_n \rangle$	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
$\mathcal{D} = \{d_1, \ldots, d_n\}$	Remote data elements
$\rho : \mathcal{D} \nrightarrow \mathcal{D}$	Part-of relation for data elements
$P(k)$	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
$\boldsymbol{\ell}(k)$	Query evaluation latency after the k-th input event
$\boldsymbol{\ell}_{remote}(d)$	Transmission latency for remote data element d

Formal evaluation model. We formalise the above model for the evaluation of CEP queries as follows. Let Q be a query and τ_Q its time window. Then, the result of evaluating Q over a stream $S = \langle e_1, e_2, \ldots \rangle$ are matches. A 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 $e'_j = e_l$ it holds that $i < j$ implies $k < l$, and respects the time window of the query, i.e., $e'_m.t - e'_1.t \leq \tau_Q$.

The set of partial matches that is maintained by the CEP system for query Q, after processing a stream prefix $S(..k)$, is denoted as $P(k) = \{\langle e_1, \ldots, e_n \rangle, \ldots, \langle e'_1, \ldots, e'_m \rangle\}$. The next event in the stream is $S(k{+}1)$. Processing it potentially requires remote data. We model the set of data elements needed by a partial match $p \in P(k)$ when processing event $S(k+1)$ by a set $D(p, k+1) \subseteq \mathcal{D}$. All such elements are given by $D(k{+}1) = \bigcup_{p \in P(k)} D(p, k{+}1)$. Based thereon, the functionality of a CEP system can be described as a function that takes the event $S(k+1)$, the current partial matches $P(k)$, and the required remote data $D(k+1)$ as input and returns sets of new partial matches $P(k+1)$ and (complete) matches $C(k+1)$:

$$f_Q(S(k+1), P(k), D(k+1)) \mapsto P(k+1), C(k+1). \quad (1)$$

Applying this function repeatedly for a stream constructs a stream of sets of matches, $R = \langle C(1), C(2), \ldots \rangle$. To achieve compositionality of the model, one may order the matches per set and construct a single event per match. This way, R is transformed into an event stream again. Table 1 summarises our notations.

2.2 Problem Statement

Query evaluation incurs latency—the time between the arrival of the last event of a match at the CEP system and the actual detection of the match. In our model, this corresponds to the time needed to evaluate function f_Q. We denote the latency observed for a match c by $\boldsymbol{\ell}(c)$. For a particular point in time, i.e., when all matches $C(k)$ have been generated, the latency is derived through some aggregation function α (e.g., median) from all these matches, $\boldsymbol{\ell}(k) = \alpha\{\boldsymbol{\ell}(c) \mid c \in C(k)\}$. In practice, it is common to incorporate some smoothing in the latency assessment (e.g., using a sliding window). We assume such smoothing to be part of the definition of $\boldsymbol{\ell}(k)$.

Based thereon, we formulate the problem of efficiently evaluating a CEP query as the minimization of the respective latency.

Problem 1 (Efficient Query Evaluation)**.** Let Q be a CEP query and let $S(..k)$ be a stream prefix. The problem of *efficient query evaluation* is to compute all matches $R = \langle C(1), \ldots, C(k) \rangle$ of Q, while minimizing the latency $\boldsymbol{\ell}(k)$, with $k \to \infty$.

Preprint

3 THE EIRES FRAMEWORK

To address Problem 1, we propose the EIRES framework. Below, we summarize its intuition, before outlining its components.

Intuition. We first reflect on the factors that constitute the latency $\boldsymbol{\ell}(c)$ of match c, and hence, the aggregated latency $\boldsymbol{\ell}(k)$. A first aspect is the latency that is inherent to the evaluation of partial matches when processing all events from the first to the last event of match c, denoted by $\boldsymbol{\ell}_{match}(c)$. In addition, there is the latency of fetching remote data that was required to process these events, denoted by $\boldsymbol{\ell}_{fetch}(c)$. It is the sum of the latencies of all fetch operations required for c, each defined by two timestamps (t_n, t_a), i.e., the time $t_n \in \mathbb{N}$ at which the *need* for remote data is detected during query evaluation and the time $t_a \in \mathbb{N}$ at which the data is *available* and processing can continue. With $F(c) \subseteq \mathbb{N} \times \mathbb{N}$ as the set of the fetch operations during the construction of match c, we have:

$$\boldsymbol{\ell}(c) = \boldsymbol{\ell}_{match}(c) + \boldsymbol{\ell}_{fetch}(c) = \boldsymbol{\ell}_{match}(c) + \sum_{(t_n, t_a) \in F(c)} t_a - t_n \quad (2)$$

Existing optimizations for CEP, such as those based on state sharing [6, 7], rewriting [8, 9], or load shedding [10, 11], will ultimately improve the inherent latency of query evaluation, i.e., $\boldsymbol{\ell}_{match}(c)$.

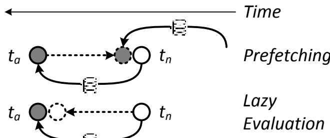


Figure 3: Intuition of the proposed strategies.

By contrast, we focus on the latency induced by fetching of remote data, $\boldsymbol{\ell}_{fetch}(c)$. As this latency corresponds to the time interval between the need (t_n) and availability of remote data (t_a), Figure 3 illustrates that it may be reduced from either end. Here, we assume that the latency of the actual transmission of a data element d, i.e., $\boldsymbol{\ell}_{remote}(d)$, as induced by the requests and responses sent over the network, is monitored. The latency $\boldsymbol{\ell}_{remote}(d)$ may 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, by postponing the evaluation of the predicates that are based on the remote data. We realize 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 CEP engine, from which it may be retrieved with negligible latency. PFetch needs to realize two main operations:

(P1) Decide when prefetching a data element may be beneficial.
(P2) At a specific time, select which data elements to prefetch.

Here, the goal is to prefetch data elements such that they arrive *right before* they are needed, since this maximizes the fetching accuracy and efficient cache usage, while still hiding the transmission latency.

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, query evaluation continues in parallel to this fetching operation. The respective predicates are evaluated only at later stages, so that fetching of remote data is no longer a blocking operation. For this, LzEval realizes two main operations:

(L1) Decide on the partial matches for which lazy evaluation is applied, i.e., for which the evaluation of predicates is postponed.
(L2) 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, the challenge is that although postponing the evaluation of predicates can hide the transmission latency of remote data, $\boldsymbol{\ell}_{remote}(d)$, it may simultaneously increase the inherent evaluation latency, $\boldsymbol{\ell}_{match}(c)$, since postponement makes event selection less strict and thus results in the creation of additional partial matches. Therefore, LzEval shall only be applied as long as its benefits outweigh the associated increase in the evaluation latency.

Both 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.

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.

EIRES components. To realize the above ideas, the EIRES framework comprises three main components. As shown in Figure 4, these components extend the functionality of a CEP engine.

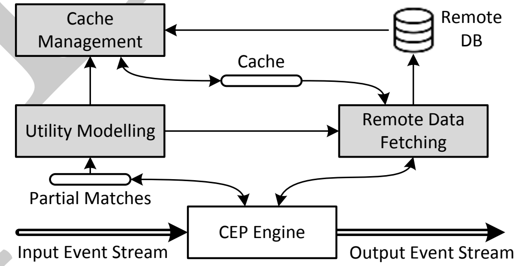


Figure 4: Components of the EIRES framework.

Utility modelling (§4) provides a cost model to assess the (expected) utility of remote data elements for 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 (L1); and to govern the cache management (C1).

Remote data fetching (§5) realizes the PFetch and LzEval strategies and manages the trade-offs induced by operations (P1) and (P2), and by (L1) and (L2), respectively, using the cost model. The trade-offs are managed in an *online* manner, i.e., based on the information available at a specific point in time. The reason being that an optimal plan to combine PFetch and LzEval for some future states would require the utility of data elements to be stable. Yet, since the utility is based on the maintained partial matches, it may differ at the time a (pre)fetching decision is taken and at the time the data arrives at the CEP engine. Hence, the operations for prefetching and lazy evaluation are decoupled from the cache management.

Cache management (§6) realizes operation (C1), i.e., it retains the data elements in the cache that are most beneficial according to the cost model. Following the above argument on the infeasibility of optimal plans, cache management is also conducted *online*, using the information currently available.

Preprint

Algorithm 1: EIRES workflow.

Input: Input event $S(k+1)$; query Q with time window τ_Q; partial matches $P(k), \ldots, P(k-\tau_Q)$; CEP engine f_Q; cache $\mathbb{C}$.
Output: Matches $C(k+1)$; partial matches $P(k+1)$.

```
1  U, #P(k) ← utilityEstimation (Q, P(k), ..., P(k − τ_Q));          // Alg. 2
2  T, O ← prefetchTiming();                                           // P1 in PFetch, Alg. 3
3  P′, C′ ← f_Q(S(k + 1), P(k), D(k + 1) ∩ ℂ);
4  D′ ← ∅;
   // Determine data elements for prefetching by lookahead timing
5  foreach p ∈ P′ \ P(k) do                                           // For each new partial match
       // For each d for which p is in its prefetch class
6      ∀ d ∈ D(p, k + 1) : if p is in class T(d) then D′ ← D′ ∪ {d};
   // Determine data elements for prefetching by estimated arrival timing
7  foreach p ∈ ⋃_{k−τ_Q ≤ i ≤ k} P(i) do                              // For each partial match
       // For each d for which the time offset to fetch has passed for p
8      ∀ d ∈ D(p, k + 1) : if p is older than O(d) then D′ ← D′ ∪ {d};
   // P2 in PFetch: Prefetch elements not in cache that have high utility
9  foreach d ∈ D′ \ ℂ do
10     if U(d, k, k + τ_Q) > min_{d′∈ℂ} U(d′, k, k + τ_Q) then Prefetch d;
11 P″, C″ ← LzEval(S(k + 1), P′, D(k + 1), ℂ, f_Q, #P(k));            // Alg. 4
12 C(k + 1) ← C′ ∪ C″;
13 P(k + 1) ← P′ ∪ P″;
14 return P(k + 1), C(K + 1);
```

EIRES workflow. As shown in Algorithm 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 (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).

4 UTILITY MODELLING

We first present measures to assess the utility of remote data elements (§4.1). Since utility is in part determined by future system states, we also present a method for its efficient estimation (§4.2).

4.1 Utility Definition

During query evaluation, the role of a remote data element is to enable the evaluation of some query predicate to determine 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. Since a fetched data element is cached, it may be used to evaluate predicates for upcoming partial matches as well. Hence, our utility assessment considers both, the current 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 CEP engine needs to handle event $S(k+1)$. Recall from §2.1 that, for a partial match $p \in P(k)$, $D(p, k+1)$ is the set of data elements required as input for processing. For a data element $d \in \mathcal{D}$, 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 $\boldsymbol{\ell}_{remote}(d)$:

$$UU(d, k) = \boldsymbol{\ell}_{remote}(d) \cdot \left|\{p \in P(k) \mid \exists\, d' \in D(p, k+1) : d \in \rho^*(d')\}\right|. \quad (3)$$

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 $k < i \leq k'$:

$$FU(d, k, k') = \sum_{k < i \leq k'} UU(d, i) \quad (4)$$

However, since information about future states is inherently uncertain, we actually need to compute an estimate for the future utility up to a time point k', 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:

$$U(d, k, k') = \omega \cdot UU(d, k) + (1 - \omega) \cdot \hat{FU}(d, k, k') \quad (5)$$

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 realization of PFetch, LzEval, and cache management. Specifically, the strategies for fetching remote data, PFetch and LzEval, assign higher weights to the urgent utility compared to 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 (§2.1) 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 states, at least on an aggregated level (aggregation from stream index k to k' in Eq. 4).

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 Eq. 3.

4.2 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 materialized in an online manner, this estimation 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 class 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. It should be noted that urgent utility UU is directly monitored (line 2 and 6).

Preprint

Number of partial matches. We partition partial matches into *classes*, where the partitioning is determined by the adopted computational model. That is, in an automata-based model, each state of the automaton denotes a class. Intuitively, the class of a partial match, through a set of query predicates, induces a class of data elements that may be needed for the evaluation of its predicates. E.g., in Figure 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 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 classes of partial matches (i.e., automaton states), then we denote the expected number of matches of a class 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 class 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 class, 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 class for which a particular data element is relevant, we employ a stochastic model. Given a class of partial matches j and a data element $d \in \mathcal{D}$, 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 class 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 $d \in \mathcal{D}$, as follows:

$$\hat{FU}(d, k, k') = (k' - k) \sum_{\substack{1 \le j \le n \\ d' \in \mathcal{D}, d \in \rho^*(d')}} \#P^j(k) \cdot Pr(j, d', k) \quad (6)$$

Both, the average number of partial matches per class and the distribution of attribute values (as the basis for the probability distribution of the data access per class), are based on simple counts, which is why they can be maintained incrementally and relatively efficiently for a sliding window (line 3-4 and 8-10 in Alg. 2). Nevertheless, since their computation still introduces overhead, instead of recomputing all utility values every time a single event is processed, a lower frequency may be employed to trade off precision of the utility model versus the incurred computational overhead.

5 REMOTE DATA FETCHING

The EIRES framework defines PFetch and LzEval as two strategies to fetch remote data, which we describe in §5.1 and §5.2, respectively. As illustrated in line 5-10, Alg. 1, EIRES always performs prefetching.

Algorithm 2: Utility estimation.

Input: Query Q with time window τ_Q; partial matches $P(k), \ldots, P(k - \tau_Q)$, partitioned in classes $1 \le j \le n$, i.e., $P^1(k), P^2(k), \ldots, P^n(k - \tau_Q)$. *System configuration parameter*: Weighting factor ω;
State: Maintained transition counts per remote reference key *tranKey* and per class of partial matches *tranClass*.
Output: Utility function U; estimated number of partial matches $\#P(k)$ partitioned in classes $\#P^i(k)$, $1 \le i \le n$.

1 **foreach** $d \in \mathcal{D}$ *required by a partial match* $p \in P(k) \setminus P(k-1)$ *of class* 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   $tranClass(j, k) \leftarrow tranClass(j, k) + 1$;
5 **foreach** $d \in \mathcal{D}$ *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** $d \in \mathcal{D}$ *required by partial matches of class* j **do**
  `// Estimate future utility`
8   $Pr(j, d, k) \leftarrow \sum_{i=k-\tau_Q}^{k} tranKey(d, j, i) \;/\; \sum_{i=k-\tau_Q}^{k} tranClass(j, i)$;
9   $\#P^j(k) \leftarrow \textbf{AVG}\left(\left|P^j(k - \tau_Q)\right|, \left|P^j(k - \tau_Q + 1)\right|, \ldots, \left|P^j(k)\right|\right)$;
10   $\hat{FU}(d, k, k + \tau_Q) \leftarrow \sum_{\substack{1 \le i \le n \\ d' \in \mathcal{D}, d \in \rho^*(d')}} \#P^i(k) \cdot Pr(i, d', k)$;
  `// 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** $U, \#P(k)$;

If prefetching did not prepare required remote data on time, EIRES performs lazy evaluation (line 11).

5.1 Prefetching

As explained in §3, two operations need to be instantiated for PFetch: deciding when prefetching is beneficial for a data element (P1) and selecting the elements to prefetch (P2).

(P1): Prefetch timing. The best time to prefetch a data element d is such that it arrives *right before* it is needed, i.e., ideally, prefetching is triggered at $t_p = t_n - \ell_{remote}(d)$. Prefetching too early should be avoided, since this will be based on a less accurate estimation of the future utility of d, given that it is generally more complex to make predictions further in advance. Hence, earlier prefetching increases the chance that d is prefetched, consumes space in the cache, but is never used in query evaluation. 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 workflow shown in Alg. 3.

Lookahead timing. We strive to identify partial match classes that provide useful indicators of when to prefetch a data element related to another match class, i.e., a *lookahead* class. To do this, we exploit the partial order of the classes of partial matches, derived from the query predicates. This partial order corresponds to the order of an automaton's states, which captures that partial matches of one class may become those of another class during query evaluation.

Let $\{1, \ldots, n\}$ be the identifiers of classes of partial matches with m being a class of matches that require a data element d (line 2 in Alg. 3). Then, we determine a lookahead class j that is a predecessor of m in the partial order of classes, denoted $j \prec m$, meaning that partial matches of class j potentially develop into those of class m.

Preprint

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

```
State: Monitored event input rates λ[n]; latest prefetch cache hit history H.
Output: Prefetch timing function T; offset timing function O.
1  T ← ∅; O ← ∅;
2  foreach d ∈ D required by partial matches of class m do
3      j ← m's directly preceding class;
4      while j ≺ m do
           // Check if lookahead timing can be applied
5          if partial matches of class 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 class ;    // Try next class
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;
```

From the set of all preceding classes $\{1 \leq j \leq n \mid j \prec m\}$, we choose j such that 1) its partial matches contain events of which the payload serves as a reference to identify d, and 2) 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 class m has one directly preceding class, though in practice it may have multiple such classes. If multiple classes 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 classes, *i.e.* , ordered by monitored transmission latencies. EIRES incorporates both of these approaches.

We determine the lookahead class j in a dynamic manner, based on the recent cache hit history from cache management (input $\mathcal{H}$ in Alg. 3). Particularly, for a given data element d, $\mathcal{H}$ maintains cache hit/miss information, where $\mathcal{H}(i, i', d)$ returns *false* if d was prefetched upon the construction of a partial match of class i, but was not available when required during the evaluation of a partial match of class i'. Such a cache miss can have two causes: First, it may hold that $t_{i \to i'} < \boldsymbol{\ell}_{remote}(d)$, i.e., prefetching happens too late because the time for a partial match to develop from class i to i' is shorter than the transmission latency. Second, when constructing a partial match of class 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 class is not suited to trigger prefetching.

In sum, for a partial match of class m that requires a data element d, we select the preceding class j closest to m for which $\mathcal{H}(d, j, m) =$ *true* (line 6). Then, the point in time to prefetch data element d for partial matches of class m is defined as the moment that a partial match for class j is constructed. For example, if $\mathcal{H}(d, j_1, m)$ and $\mathcal{H}(d, j_2, m)$ are both *true*, and $j_1 \prec j_2$, we select j_2 as the lookahead class for m, avoiding the unnecessarily early fetching that using j_1 would yield. If partial matches of multiple classes m reference d, the smallest index j, in terms of the partial order $\prec$, is chosen.

We deal with event stream fluctuations by internally maintaining counts of cache misses in $\mathcal{H}$, so that a threshold can determine what to interpret as sufficient negative evidence. Also, values are reset to zero after a fixed time period after their last increment.

Note that there may be partial match classes for which lookahead timing is unsuitable, i.e., cache misses occur since there is no lookahead class 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 lead to the creation of the partial match of class m. An example for that is the class q_4 in Figure 2, which requires the evaluation of a predicate using remote data on pre-authorized clients (`r[q3.org]`) that can only be fetched based on the event that lead to the respective match. For those cases, we determine the time to prefetch based on *estimated-arrival timing*.

Estimated-arrival timing. When lookahead timing is not applicable for a partial match class 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 class m but are *not* based on remote data (line 10-12 in Alg. 3). Consider again class q_4 in Figure 2. The respective partial matches are extended when an event with `x.type=T`, `x.vol>50k`, and `x.ben` $\notin$ `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 class q_4 and the time the data element r[q_4.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.

Here, we illustrate the general procedure when event arrival follows a Poisson process [20], 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. This is similar to Akdere *et al.*'s work [21], but differs in granularity: Akdere *et al.* [21] works on the level of processing states, whereas our model estimates the arrival of events that have a certain payload.

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 d is needed as $t + {}^1/_\lambda$, so that the time for prefetching becomes $t_p = t + {}^1/_\lambda - \boldsymbol{\ell}_{remote}(d)$ (line 11 in Alg. 3). Taking up the previous example, events with `x.type=T x.vol>50k` may be rare, such that the expected inter-arrival time is $E = 300ms$. Then, with a data transmission latency of $\boldsymbol{\ell}_{remote} = 50ms$, we trigger prefetching only $250ms$ after the creation of the respective partial match (i.e., when processing the next input event after this time 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 $\mathbb{C}(k) \subseteq \mathcal{D}$ be the content of the cache and $D(k)$ the set of data elements for which prefetching at time k would be beneficial. Then, we select $D'(k)$ for prefetching, defined as

$$D'(k) = \{d \in D(k) \mid U(d) > \min_{d' \in \mathbb{C}(k)} U(d')\}. \tag{7}$$

Preprint

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

```
Input: Input event S(k + 1); partial matches P(k); remote data D(k + 1); cache
       ℂ; CEP engine f_Q; estimated number of partial matches #P(k)
       partitioned in classes #P^i(k), 1 ≤ i ≤ n.
State: Monitored event input rates λ[n].
Output: Matches C(k + 1); partial matches P(k + 1).
1  C(k + 1) ← ∅; P(k + 1) ← ∅; succ^ ← ∅;
2  foreach d ∈ D(k + 1) \ ℂ required by partial matches of class j do
3      λ ← λ[j]; ℓ ← ℓ_remote(d);
4      foreach class m, j ≺ m do
           // if not yet known, estimate if postponing is beneficial
5          if ⟨m, ℓ⟩ is not checked against succ^(j, ℓ) then
6              λ ← λ + λ[m];    E(j, m) ← 1/λ;
7              Δ_-ℓ_remote ← min(E(j, m), ℓ);
8              Δ_+ℓ_match ← ℓ_pm ∏_{1≤i≤m} (#P^i(k) · λ[i + 1] · E(j, m));
9              if Δ_-ℓ_remote > Δ_+ℓ_match then
10                 succ^(j, ℓ) ← succ^(j, ℓ) ∪ {m};
11             else succ^(j, ℓ) ← succ^(j, ℓ) \ {m} ;
           // benefit expected, postpone evaluation of d's predicates
12         if ⟨m, ℓ⟩ ∈ succ^(j, ℓ) then
13             Fetch d;
14             P′ ← f_Q(S(k + 1), P(k)) ;            // ignore d's predicates
15         else Fetch d, block stream processing until after d arrives at ℂ ;
16         P″, C″ ← f_Q(S(k + 1), P(k) ∪ P′, D(k + 1) ∩ ℂ);
17         P(k + 1) ← P(k + 1) ∪ P″;
18         C(k + 1) ← C(k + 1) ∪ C″;
19 return C(k + 1), P(k + 1);
```

5.2 Lazy Evaluation

For LzEval, in turn, two operations need to be instantiated: selecting the partial matches for which the strategy is applied (L1) and adapting the evaluation procedure for those matches (L2). The workflow of LzEval is sketched in Alg. 4.

(L1): Selection of partial matches. LzEval triggers a fetch operation for remote data d when it is required, but postpones the actual evaluation of the respective predicates. While this hides the transmission latency $\boldsymbol{\ell}_{remote}(d)$, it also makes event selection less strict, possibly resulting in an exponential growth of the number of partial matches. As such, the inherent evaluation latency $\boldsymbol{\ell}_{match}(c)$ may increase, which may thwart the benefit of the reduction of $\boldsymbol{\ell}_{remote}(d)$. Recognizing this trade-off, we therefore only apply LzEval to those partial match classes where an actual benefit is expected.

To determine these classes, we estimate the time gained by hiding the transmission latency, denoted as $\Delta_{-}\boldsymbol{\ell}_{remote}$, and the overhead incurred by the additional partial matches, $\Delta_{+}\boldsymbol{\ell}_{match}$. The latency gain, $\Delta_{-}\boldsymbol{\ell}_{remote}$, 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., $t_a \prec t_e$), so that the entire transmission latency is hidden, i.e., $\Delta_{-}\boldsymbol{\ell}_{remote} = \boldsymbol{\ell}_{remote}(d)$. The overhead $\Delta_{+}\boldsymbol{\ell}_{match}$ 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, $\boldsymbol{\ell}_{pm}$, 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 (§5.1). Assume that the evaluation of a predicate of class j requires remote data, yet that this evaluation may be postponed until the creation of a partial match of a later class m, $j \prec m$. Then, we estimate the average time $t_{j\to m}$ for a partial match of class j to develop into one of class m, when ignoring all predicates that are based on 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 classes $\{r_1, \ldots, r_s\}$ 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 class 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 classes, 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 $\Delta_{-}\boldsymbol{\ell}_{remote}(j,m) = \min(E(j,m), \boldsymbol{\ell}_{remote}(d))$ (line 7 in Alg. 4). The estimation of $\Delta_{+}\boldsymbol{\ell}_{match}(j,m)$ is time-varying and incorporates $\#P^i(k)$, i.e., the expected number of partial matches of class i, as estimated at time point (or stream index) k, see §4.2. For each class 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 ($\boldsymbol{\ell}_{pm}$) yields the estimate for accumulated increase in evaluation latency (to simplify notation, we define $r_{s+1} = m$):

$$\Delta_{+}\boldsymbol{\ell}_{match}(j,k)(k) = \boldsymbol{\ell}_{pm} \prod_{1\leq i\leq s} (\#P^{r_i}(k) \cdot \lambda_{r_{i+1}} \cdot E(j,m)) \quad (8)$$

For each class j that requires remote data during query evaluation, we determine the set of succeeding classes for which lazy evaluation is beneficial. This set is defined as $\hat{succ}(j) = \{m \in \{1,\ldots,n\} \mid j \prec m \wedge \Delta_{-}\boldsymbol{\ell}_{remote} > \Delta_{+}\boldsymbol{\ell}_{match}\}$, 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 class 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, $\boldsymbol{\ell}_{remote}(d)$ (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.

(L2): Adapted evaluation procedure. Consider partial matches of a class j and $\hat{succ}(j)$ as the succeeding classes for which lazy evaluation is beneficial. Also, let $S(..k)$ be the time the stream prefix was processed and $\mathbb{C}(k) \subseteq \mathcal{D}$ as the cache content. Then, query evaluation to process event $S(k+1)$ is adapted for the partial matches of class 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 class j develops into one of a class $m \in \hat{succ}(j)$. Upon construction of a respective partial match, the cache is checked and, upon data availability, the predicate is evaluated.

Once a partial match of class j develops into a class $m' \notin \hat{succ}(j)$ 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 continues only once the data element d becomes available (line 15 in Alg. 4).

Preprint

6 CACHE MANAGEMENT

Cache management in EIRES shall retain the data elements that are most beneficial in the cache, thereby realizing operation (C1) as introduced in §3. Although the benefit of a data element strongly relates to its expected utility (see §4), certain query semantics suggest to base cache management on a relatively simple policy. Recall the greediness of CEP queries in event selection (§2.1). Either semantics motivates a different policy for the 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 materialize. This, in turn, will induce a large number of access requests to the cache for the respective data. In terms of 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 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 match, 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.

Once the cache capacity is reached, data elements (from T_2 before T_1) are evicted based on their utility. Let $\mathbb{C} \subseteq \mathcal{D}$ 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 $\mathcal{R} \subseteq \mathbb{C}$ to retain can be formulated as a knapsack problem:

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

The above is a standard, NP-hard, knapsack problem [22]. Yet, it may be approximated [23], e.g., by selecting data elements in the order of their utility and size ratios, until the capacity is reached.

7 EXPERIMENTAL EVALUATION

We evaluated the EIRES framework on various scenarios. §7.1 first outlines the setup. For controlled experiments, §7.2 reports on the overall efficiency and effectiveness, while §7.3 explores the sensitivity. Finally, §7.4 discusses two real-world scenarios. Note that an extended version of this paper including query specifications and additional experiments as well as the employed implementation and datasets are publicly available.[1]

[1] https://github.com/zbjob/EIRES

7.1 Experimental Setup

Dataset and queries. For controlled experiments, we generated synthetic data with a `Type` ($\mathcal{U}(\{A, B, C, D\})$), an `ID` ($\mathcal{U}(1, 100)$), and two numeric attributes V_1 ($\mathcal{U}(1, 100000)$), and V_2 ($\mathcal{U}(1, 100000)$). This dataset enables 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.

For the synthetic dataset, we evaluate the queries Q1 and Q2 of Listing 2. They differ in their structure (pure sequence vs. disjunction of sequences) and the integration of remote data in predicates.

```
Q1:
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
AND b.v1=REMOTE[h.v1]
WITHIN 8min
Q2:
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 2: Queries for the synthetic dataset.

Baselines. We compare our approaches for data fetching, PFetch, LzEval, and their combination (Hybrid) against three baselines. *BL1* denotes the naive integration of data fetching. It interrupts query evaluation when remote data is needed and continues once it has been fetched. *BL2* employs a cache of remote data to improve the efficiency of query evaluation. For this cache, we consider both policies discussed in §6, i.e., LRU and the cost-based policy. Finally, *BL3* first ignores all predicates based on remote data. Upon reaching a final state of the evaluation model, it fetches the remote data and conducts the respective event selection.

Measures. Our focus is the latency of query evaluation, 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 100k to 5 million matches. Specifically, we report the 5th, 25th, 50th, 75th, and 95th percentiles of the latency values. Moreover, we report the throughput, i.e., the number of events processed per second.

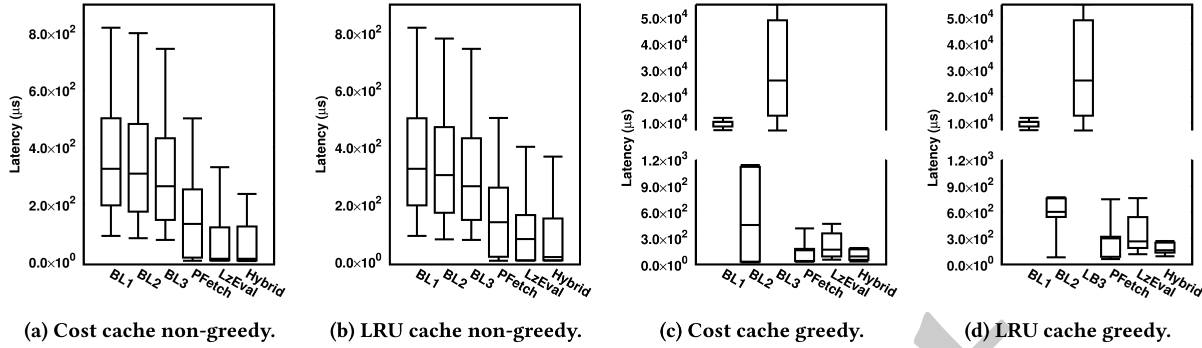
Implementation and environment. For the evaluation, we used an automata-based CEP engine, written in C++. All reported results are averaged over 20 runs on a NUMA node with 4 Intel Xeon E7-4880 CPUs (60 cores) and 1TB RAM, running openSUSE 15.0.

7.2 Overall Effectiveness and Efficiency

Figures 5 and 6, respectively, depict the overall performance obtained for Q1 and Q2 on the synthetic dataset. We compare the PFetch, LzEval, and Hybrid strategy against the three baselines under both greedy and non-greedy selection 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.

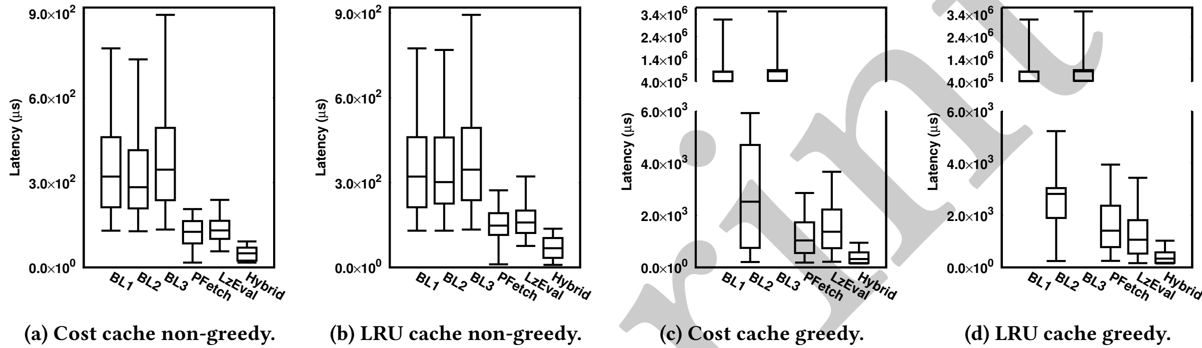
As shown in the figures, the Hybrid strategy consistently outperforms all other approaches. Furthermore, both PFetch and LzEval also always outperform all three baselines considerably, though it depends on the context which of these achieves better results.

Preprint



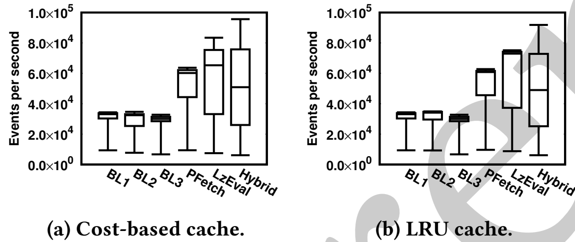
(a) Cost cache non-greedy. (b) LRU cache non-greedy. (c) Cost cache greedy. (d) LRU cache greedy.

Figure 5: Overall effectiveness and efficiency for Q1.



(a) Cost cache non-greedy. (b) LRU cache non-greedy. (c) Cost cache greedy. (d) LRU cache greedy.

Figure 6: Overall effectiveness and efficiency for Q2.



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

Figure 7: Throughput for Q1 under non-greedy selection.

Selection strategies. For non-greedy selection in Q1, 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 26×. For the 95th percentiles, the reduction is at least 4× (cost-based cache) and 2.5× (LRU). For Q2, we observe smaller gains in median latencies, yet bigger gains for the 95th percentile. Among the baselines, *BL3* outperforms both *BL1* and *BL2* for Q1, but this is the opposite for Q2. This is because Q1 has two states requiring different remote data in the CEP engine, whereas Q2 has one at each branch of disjunction sequences. Since *BL3* fetches different data items at once, its aggregated transmission latency is the maximal latency of all required remote data, instead of their sum (as for *BL1* and *BL2*). For Q1, this benefit outweighs the overhead caused by extra partial matches under non-greedy selection.

The results show greater variety for greedy selection policies. For Q1, Hybrid reduces the median latency by 111× (cost-based cache) and 63× (LRU) when compared to *BL1*. Furthermore, 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 95th 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 Q2, gains are even more extreme, i.e., Hybrid reduces median latencies by up to 2,726× and the 95th percentiles latency by up to 3,752×.

Caching policies. The impact of a local cache differs considerably per scenario. Since the reusability of data elements is low for non-greedy selection, a cache contributes little here, as illustrated by the comparable performance of *BL1*, *BL2*, and *BL3* in Figure 5a-5b and Figure 6a-6b. For greedy selection, in turn, adding a cache reduces latencies by at least two orders of magnitude, see *BL1*, *BL2*, and *BL3* in Figure 5c-5d and Figure 6c-6d. In such settings, LRU outperforms a cost cache for *BL2* due to its 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 5a) outperforms its counterpart with LRU (Figure 5b), 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.

Benefits of Hybrid. The Hybrid strategy always outperforms the individual PFetch and LzEval strategies, as their combination helps to overcome their individual weaknesses. For instance, the performance of PFetch is closely related to the quality of the made prefetching predictions. When PFetch fails to prefetch a data element, query evaluation is interrupted, resulting in high tail latencies (95th percentile), see Figure 5d. LzEval generally has lower median

Preprint

and 75th percentile latencies (Figure 5a-5b), because its fetching decision is always accurate. Yet, additional partial matches affect the latency under greedy selection, see Figure 5c-5d and Figure 6c-6d.

Hybrid combines alleviates the aforementioned issues. If inaccurate prediction lead to non-effective prefetching, Hybrid still (generally) performs lazy evaluation, thereby avoiding the interruption of stream processing. However, the overhead is smaller than using LzEval alone, since, due to prefetching, many matches are already handled by prefetching.

Throughput. We also investigate the throughput performance. Due to limited space, Figure 7 focuses on the throughput for Q1 under non-greedy selection with either policy for cache management. 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 LRU, the observed throughput is virtually equivalent.

7.3 Sensitivity Analysis

We assess the sensitivity of the proposed strategies with respect to: 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 Eq. 5. All results were obtained for query Q1, with a cost-based cache and greedy selection.

Utility estimation quality. The quality of the utility model 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 materialize. We compare the corresponding latency variations by injecting noise into 10% to 90% of the estimations.

PFetch is sensitive to such noise, see Figure 8a, since its median latency grows for higher noise levels. The reason is twofold: PFetch prefetches the wrong data elements, while 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 75th percentile and Hybrid's 95th percentile.

LzEval is less sensitive to noise, even obtaining stable latencies across noise levels for the 5th and 25th 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 decision 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 95th 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 75th 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 for Hybrid.

Cache size. The size of the employed cache naturally affects all strategies, where a larger cache size is beneficial, see Figure 8b. 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 size 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 delays. As shown in Figure 8c, the 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. Lastly, we consider the impact of ω, which balances the urgent and future utility in our model (§4.1, Eq. 5). We consider the impact of ω in the utility to guide the fetching of remote data (§5.1) and to manage the cache (§6). We refer to these, respectively, as ω_{fetch} and ω_{cache}.

Figure 9a shows the results obtained when varying ω_{fetch}, while fixing ω_{cache} at 0.5. Here, increasing the weight reduces the 75th and 95th 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 $\omega_{fetch} = 0.7$. However, the utility model 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 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 9b. 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.

7.4 Case Studies

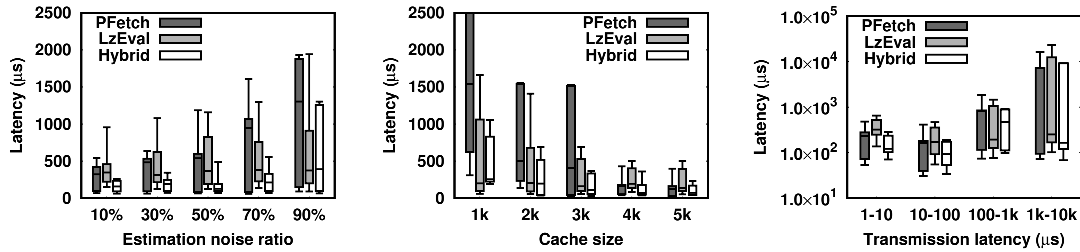
Finally, we applied our approach in two real-world scenarios, using a cost-based cache under greedy selection.

Bushfire detection. This case uses a real-world dataset of a bushfire detection system. The system's geostationary operational environmental satellite, GOES-16 [24], is capable of detecting heat signatures produced by fires [25]. While it tracks these in real-time, the obtained information is combined with data from a ground-based sensor network for fine-grained validation. For this dataset, we employ a query that detects bushfires during daytime [25]. In essence, the query identifies the repeated occurrence of a specific radiation pattern for a geographical area in the satellite data. However, to evaluate this query, the receiver handling the satellite data needs to fetch remote data from temperature and humidity sensors.

As shown in Figure 10a, 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*, *BL2* and *BL3* by 206×, 21× and 200×, respectively. For the 95th percentile latencies, the improvements are 18×, 13× and 14×. We also observe that PFetch has similar performance as Hybrid, except for the 95th percentile latency, which shows that PFetch accurately anticipates the need for remote data.

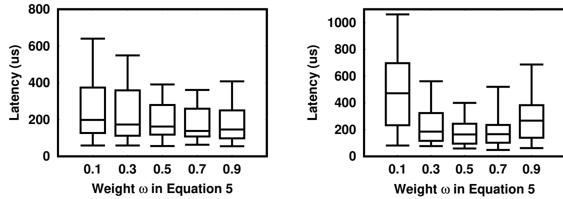
Cluster monitoring. This case uses data from Google Cluster Traces [26]. It contains events that indicate the lifecycle of tasks

Preprint



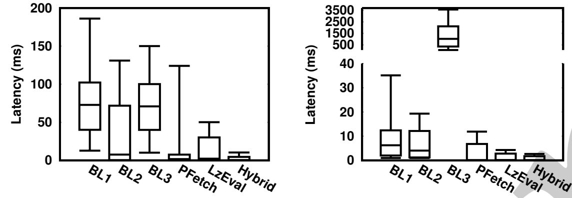
(a) Sensitivity of utility estimation. (b) Sensitivity of cache size. (c) Sensitivity of transmission latency.

Figure 8: Sensitivity analysis for utility estimation, cache size and remote data transmission latency.



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

Figure 9: Sensitivity analysis for utility weighting factor.



(a) Bushfire detection. (b) Cluster monitoring.

Figure 10: Case studies.

running in a large-scale cluster. For this dataset, we employ a query that detects the following pattern: A task is submitted, scheduled, and evicted on one machine; later, in a different region, it is rescheduled and evicted again; 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 10 highlight that Hybrid outperforms all other approaches. It reduces the median latencies by 73×, 47× and 11,879× for *BL1*, *BL2* and *BL3*. For the 95$^{\text{th}}$ percentile latencies, Hybrid improves 13×, 7× and 1,336×.

Discussion. We observe more than an order of magnitude difference in performance benefits achieved for synthetic and real-world datasets. This is because the queries for the real-world datasets contain many compute-intensive predicates (e.g., the bushfire detection query computes the spatial overlap of geographic areas). Also, these queries have larger time windows and, hence, tend to generate more partial matches.

8 RELATED WORK

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

Prefetching: Two surveys [27, 28] review general mechanisms for prefetching. Many of them are based on data access patterns derived by static and dynamic program analysis, including sequential patterns [29, 30] or patterns induced by specific operators like hash joins [31], call graphs [32], and irregular memory accesses [33, 33]. Under the umbrella of semantic prefetching, it was also suggested to derive patterns from data structure correlations [34] or from models that predict user behaviour [35].

Caching: Kossmann [36] surveyed optimization 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 synchronize cached data with the original database [37–39]. Similar questions have been addressed in the field of *web caching* [40, 41].

Compared to our work, the above mechanisms face similar trade-offs, e.g., between dataset sizes and their usage frequency. Naturally, though, the actual cost models and problem formulations to address these trade-offs look very different. Our work is the first systematic exploration of prefetching and caching strategies to incorporate data from remote sources in event stream processing.

9 CONCLUSIONS

We proposed EIRES, a framework for efficient integration of remote data in the evaluation of queries over event streams. Our core idea is to decouple fetching of remote data and its use in query evaluation. Data elements may be fetched before the need for them materializes 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.

In future work, we intend to explore the instantiation of EIRES for tree-based execution models [5] that define an order of operator evaluation and a hierarchy of buffers. Since our utility modelling is based on present and future partial matches, we expect to confirm our experimental results obtained for automata-based models.

ACKNOWLEDGMENTS

This research was funded by the Deutsche Forschungsgemeinschaft (DFG, German Research Foundation) with project-ID 246594964 and by the Federal Ministry of Education and Research (BMBF), Germany under the project LeibnizKILabor with grant No. 01DD20003.

Preprint

REFERENCES

[1] N. Giatrakos, E. Alevizos, A. Artikis, A. Deligiannakis, and M. N. Garofalakis, "Complex event recognition in the big data era: a survey," *VLDB J.*, vol. 29, no. 1, pp. 313–352, 2020. [Online]. Available: https://doi.org/10.1007/s00778-019-00557-w

[2] 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. [Online]. Available: http://doi.acm.org/10.1145/1142473.1142520

[3] 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. [Online]. Available: http://doi.acm.org/10.1145/1376616.1376634

[4] R. C. Fernandez, M. Weidlich, P. R. Pietzuch, and A. Gal, "Scalable stateful stream processing for smart grids," in *The 8th ACM International Conference on Distributed Event-Based Systems, DEBS '14, Mumbai, India, May 26-29, 2014*, U. Bellur and R. Kothari, Eds. ACM, 2014, pp. 276–281. [Online]. Available: http://doi.acm.org/10.1145/2611286.2611326

[5] 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. [Online]. Available: http://doi.acm.org/10.1145/1559845.1559867

[6] 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. [Online]. Available: http://doi.acm.org/10.1145/2588555.2593671

[7] M. Ray, C. Lei, and E. A. Rundensteiner, "Scalable pattern sharing on event streams," in *Proceedings of the 2016 International Conference on Management of Data, SIGMOD Conference 2016, San Francisco, CA, USA, June 26 - July 01, 2016*, F. Özcan, G. Koutrika, and S. Madden, Eds. ACM, 2016, pp. 495–510. [Online]. Available: https://doi.org/10.1145/2882903.2882947

[8] 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. [Online]. Available: https://doi.org/10.1109/ICDE.2008.4497476

[9] M. Weidlich, H. Ziekow, A. Gal, J. Mendling, and M. Weske, "Optimizing event pattern matching using business process models," *IEEE Trans. Knowl. Data Eng.*, vol. 26, no. 11, pp. 2759–2773, 2014. [Online]. Available: https://doi.org/10.1109/TKDE.2014.2302306

[10] B. Zhao, N. Q. V. Hung, 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. [Online]. Available: https://doi.org/10.1109/ICDE48307.2020.00099

[11] 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. [Online]. Available: https://doi.org/10.5441/002/icdt.2014.23

[12] 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*, vol. 6, no. 4, pp. 309–315, 2009.

[13] feedzai.com, "Modern Payment Fraud Prevention at Big Data Scale," https://feedzai.com/wp-content/uploads/2015/10/Feedzai-Whitepaper-Modern-Payment-Fraud-Prevention-at-Big-Data-Scale.pdf, 2013, last access: 01/02/21.

[14] A. Arasu, B. Babcock, S. Babu, J. Cieslewicz, M. Datar, K. Ito, R. Motwani, U. Srivastava, and J. Widom, "STREAM: the stanford data stream management system," in *Data Stream Management - Processing High-Speed Data Streams*, ser. Data-Centric Systems and Applications, M. N. Garofalakis, J. Gehrke, and R. Rastogi, Eds. Springer, 2016, pp. 317–336. [Online]. Available: https://doi.org/10.1007/978-3-540-28608-0_16

[15] G. Cugola and A. Margara, "Processing flows of information: From data stream to complex event processing," *ACM Comput. Surv.*, vol. 44, no. 3, pp. 15:1–15:62, 2012. [Online]. Available: http://doi.acm.org/10.1145/2187671.2187677

[16] 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*, 2007, pp. 1100–1102. [Online]. Available: http://doi.acm.org/10.1145/1247480.1247620

[17] G. Cugola and A. Margara, "Complex event processing with T-REX," *J. Syst. Softw.*, vol. 85, no. 8, pp. 1709–1728, 2012. [Online]. Available: https://doi.org/10.1016/j.jss.2012.03.056

[18] A. Adi and O. Etzion, "Amit - the situation manager," *VLDB J.*, vol. 13, no. 2, pp. 177–203, 2004. [Online]. Available: https://doi.org/10.1007/s00778-003-0108-y

[19] S. Chakravarthy and D. Mishra, "Snoop: An expressive event specification language for active databases," *Data Knowl. Eng.*, vol. 14, no. 1, pp. 1–26, 1994. [Online]. Available: https://doi.org/10.1016/0169-023X(94)90006-X

[20] J. Kingman, *Poisson Processes*, ser. Oxford Studies in Probability. Clarendon Press, 1992. [Online]. Available: https://books.google.de/books?id=VEiM-OtwDHkC

[21] M. Akdere, U. Çetintemel, and N. Tatbul, "Plan-based complex event detection across distributed sources," *Proc. VLDB Endow.*, vol. 1, no. 1, pp. 66–77, 2008. [Online]. Available: http://www.vldb.org/pvldb/vol1/1453869.pdf

[22] H. Kellerer, U. Pferschy, and D. Pisinger, *Knapsack problems.* Springer, 2004.

[23] C. Chekuri and S. Khanna, "A polynomial time approximation scheme for the multiple knapsack problem," *SIAM J. Comput.*, vol. 35, no. 3, pp. 713–728, 2005. [Online]. Available: https://doi.org/10.1137/S0097539700382820

[24] NOAA, "GOES-16: A GAME-CHANGER FOR FIGHTING DEADLY WILDFIRES," https://www.goes-r.gov/featureStories/goes16Wildfires.html, 2018, last access: 01/02/21.

[25] NOAA, "GOES-R Fire Detection and Characterization," https://www.goes-r.gov/education/docs/fs_fire.pdf, 2019, last access: 01/02/21.

[26] C. Reiss, J. Wilkes, and J. L. Hellerstein, "Google cluster-usage traces: format + schema," Google Inc., Mountain View, CA, USA, Technical Report, Nov. 2011, revised 2014-11-17 for version 2.1. Posted at https://github.com/google/cluster-data.

[27] S. P. Vanderwiel and D. J. Lilja, "Data prefetch mechanisms," *ACM Comput. Surv.*, vol. 32, no. 2, pp. 174–199, 2000. [Online]. Available: https://doi.org/10.1145/358923.358939

[28] S. Mittal, "A survey of recent prefetching techniques for processor caches," *ACM Comput. Surv.*, vol. 49, no. 2, pp. 35:1–35:35, 2016. [Online]. Available: https://doi.org/10.1145/2907071

[29] A. J. Smith, "Sequentiality and prefetching in database systems," *ACM Trans. Database Syst.*, vol. 3, no. 3, pp. 223–247, 1978. [Online]. Available: https://doi.org/10.1145/320263.320276

[30] 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. [Online]. Available: https://doi.org/10.1145/16894.16876

[31] 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. [Online]. Available: https://doi.org/10.1109/ICDE.2004.1319989

[32] M. Annavaram, J. M. Patel, and E. S. Davidson, "Call graph prefetching for database applications," *ACM Trans. Comput. Syst.*, vol. 21, no. 4, pp. 412–444, 2003. [Online]. Available: https://doi.org/10.1145/945506.945509

[33] Y. O. Koçberber, B. Falsafi, and B. Grot, "Asynchronous memory access chaining," *Proc. VLDB Endow.*, vol. 9, no. 4, pp. 252–263, 2015. [Online]. Available: http://www.vldb.org/pvldb/vol9/p252-kocberber.pdf

[34] 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. [Online]. Available: https://doi.org/10.1145/1007568.1007591

[35] 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. [Online]. Available: https://doi.org/10.1145/2882903.2882919

[36] D. Kossmann, "The state of the art in distributed query processing," *ACM Comput. Surv.*, vol. 32, no. 4, pp. 422–469, 2000. [Online]. Available: https://doi.org/10.1145/371578.371598

[37] 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. [Online]. Available: https://doi.org/10.1109/ICDE.2003.1260881

[38] C. Bornhövd, M. Altinel, C. Mohan, H. Pirahesh, and B. Reinwald, "Adaptive database caching with dbcache," *IEEE Data Eng. Bull.*, vol. 27, no. 2, pp. 11–18, 2004. [Online]. Available: ftp://ftp.research.microsoft.com/pub/debull/A04june/bornhoevd.ps

[39] 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. [Online]. Available: https://doi.org/10.1109/ICDE.2004.1319994

Preprint

[40] S. Podlipnig and L. Böszörményi, "A survey of web cache replacement strategies," *ACM Comput. Surv.*, vol. 35, no. 4, pp. 374–398, 2003. [Online]. Available: https://doi.org/10.1145/954339.954341

[41] J. Mertz and I. Nunes, "Understanding application-level caching in web applications: A comprehensive introduction and survey of state-of-the-art approaches," *ACM Comput. Surv.*, vol. 50, no. 6, pp. 98:1–98:34, 2018. [Online]. Available: https://doi.org/10.1145/3145813

Preprint