Scalable Discovery of Queries over Event Streams

Rebecca Sattler

supervised by Matthias Weidlich,
Humboldt Universität zu Berlin, Unter den Linden 6, 10099 Berlin, Germany

Abstract
Complex event processing (CEP) detects situations of interest by matching patterns in a continuous stream of events. Queries for these patterns can be learned automatically in the presence of event data that is labelled with the situation to detect. However, existing event query discovery methods deal with scaling issues in a somewhat ad-hoc manner. In this paper, we therefore outline our plan to characterize the design space for event query discovery, instantiate it with concrete algorithms, and provide means to recommend algorithms based on properties of the event data used as input. Specifically, we consider four dimensions in the design of discovery algorithms that search the space of candidate queries. We also identify the core functionality underlying the respective algorithms and provide initial results on their realization.

1. Introduction

Complex Event Processing (CEP) systems evaluate a given set of queries (or rules) over large-scale streams of event data in order to identify situations of interest [5]. As the queries establish a (causal) relation between the observed events and the phenomena to reveal, CEP is the foundation for reactive and predictive applications in a wide range of domains.

The specification of event queries is not trivial, though. In addition to the inherent complexity of event query languages, there is often only partial knowledge available (i.e., by a domain expert) on how a situation of interest materializes in an event stream. This is particularly true for predictive applications [2], where event queries are employed to anticipate a situation of interest to implement mitigation strategies.

Against this background, it was suggested to discover queries automatically, by learning them from labeled historic event data [9, 4]. In practice, such an approach cannot be expected to yield queries that can directly be employed. Rather, the results support a user in the design of a suitable query workload, as the discovered queries can be inspected, evaluated, and eventually refined to avoid apparent overfitting of the event data used as a starting point. Unlike black-box models like for instance Machine Learning approaches to detect or predict a situation of interest, relying on the discovered queries fosters traceability and explainability of the results of the respective application.

However, the problem of event query discovery is computationally hard, due to the large space of candidate queries that needs to be searched. The size of the search space depends, among other aspects, on the length of the queries to be discovered (i.e., the number of events required to occur to indicate a match of a query) as well as the size of the stream alphabet (i.e., the size of the domain of possible event values). Since queries may characterize the events to be matched by any combination of their values, the query search space grows exponentially in the query length and domain size.

Existing algorithms to event query discovery [9, 4] cope with the scalability challenges in a rather ad-hoc manner. They guide the respective search by first identifying common event values and, based thereon, discover common sequences of these values. Yet, the approaches incorporate various design choices that are motivated by implicit assumptions on the event data, for instance, in terms of the frequency distribution of event values, the distinctness of the events that are part of query matches, as well as the selectivities of the queries to discover. As a consequence, it is not clear (i) in which application scenarios the existing discovery algorithms can be expected to work, and (ii) whether incorporating different design choices may facilitate event query discovery in scenarios in which existing algorithms are intractable.

Event query discovery has been applied to a range of fields such as cluster monitoring, finances and urban transportation [9, 4], which indicates that the approach is not limited to certain areas. This PhD project sets out to systematically explore the design space of algorithms for event query discovery. Specifically, it is devoted to the following research questions:

1) *How to characterize the design space for event query discovery?*
2) *How to instantiate this space in terms of concrete algorithms?*
3) *How to provide means to recommend particular discovery algorithms based on properties of the event data used as input?*

In the light of these research questions, §2 provides a

Proceedings of the VLDB 2022 PhD Workshop, September 5, 2022. Sydney, Australia.
rebecca.sattler@hu-berlin.de (R. Sattler)
0000-0002-7342-7131 (R. Sattler)

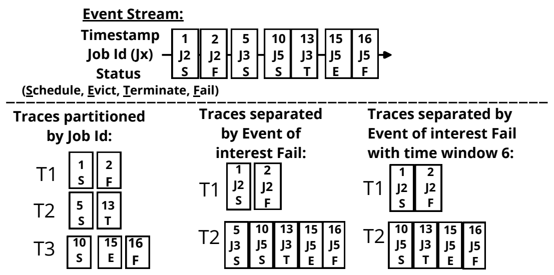
Event Stream:
Timestamp
Job Id (Jx)
Status
(Schedule, Evict, Terminate, Fail)
Traces partitioned by Job Id:
Traces separated by Event of interest Fail:
Traces separated by Event of interest Fail with time window 6:
T1
T2
T3

Figure 1: Construction of traces from an event stream.

formal characterization of the problem of event query discovery, before we review related work in §3. In §4, we present our initial research results, including a discussion of relevant design choices for discovery algorithms, a generic algorithmic template that incorporates these choices, an exemplary instantiation of this template, and initial experimental results obtained with the resulting discovery algorithm. Finally, we conclude and discuss next steps in §5.

2. Research context

2.1. Stream and Query Model

We define the model for our work as follows. An *event schema* is a finite sequence of attributes $A = (A_1, \dots, A_n)$ and each attribute can take on values from a given domain $\mathrm{dom}(A_i)$ for $i \in \{1, \dots, n\}$, which is a finite set of types. An *event* is an $n-$tuple and an instance of an event schema, i.e., $e = (a_1, \dots, a_n)$, $a_i \in dom(A_i)$ for $i \in \{1, \dots, n\}$, which can be represented as the string of the respective types. A *stream* S is an unbounded sequence of events, $S = e_1e_2 \dots$, which, again, can be represented as a string of the events. The stream events are ordered by ascending time of occurrence. Without loss of generality, we assume that all events in a stream have the same event schema. Also, the stream alphabet is defined as $\mathrm{dom}(S) := \bigcup_{i=1}^{n} \mathrm{dom}(A_i)$. Given a set of labels for a stream that indicate situation of interests, a set of traces can be extracted to serve as a training set to discover event queries. Formally, a *trace* t is a finite, non-empty subsequence of the stream S (or is string representation, respectively). Traces can be obtained in different ways. In Fig. 1, the different options are illustrated with an example. Assuming that only potentially relevant data is given, one may partition events by one of the domains A_i (i.e., traces partitioned by Job Id). Another option is to select the labeled events and then either cut traces after the occurrences of labeled events (i.e., traces separated by event of interest fail) or to only include all events that occur within a predefined time window before a labeled event (i.e., traces separated by event of interest fail with time window 6). In A *sequence database* D is a finite set of traces, $D = \{t_1, \dots, t_m\}$. To define queries over streams, and hence sequence databases, let Vars be a set of variables, such that the intersection of Vars and $\mathrm{dom}(S)$ is empty.

Definition 2.1 (Query). *Let* $\mathrm{dom}(S)$ *be a stream alphabet. Then, a query is a tuple* $q = (s, w)$, *where*
- $s \in ((\mathrm{dom}(S) \cup \mathrm{Vars})^k)^+$ *is the query string, where* k *is the number of attribute domains;*
- $w \in \mathbb{N}$ *the window size.*

In the remainder, a query that only contains types, or only variables, is called a *type query*, or *pattern query*, respectively. A query q matches a trace t, denoted by $q \models t$, if by replacing the variables with suitable types, the query string is a (not necessarily continuous) substring of the trace, while staying within the window size. A specific match is a respective substring of the trace, while the set of all matches of q in t is denoted by $\Omega(q, t)$.

Note that this query model is similar to SQL-like languages, such as SASE [16]. For instance, taking the event schema from Fig. 1, the query $q = ((x_0, S)(x_0, F), 5)$ with $S, F \in \mathrm{dom}(S), x_0 \in$ Vars could be written in the SASE language as follows:

```
PATTERN SEQ(Event e1, Event e2)
WHERE  e1.status=S AND e2.status=F AND e1.job=e2.job
WITHIN 5 minutes
```

We define the *support* of query q for the sequence database D as:

$$supp(q, D) := |\{t \in D \mid q \models t\}| .$$

Next, we consider a support threshold for the sequence database D, denoted by $supp_G(D) \in \{0, \dots, |D|\}$ which determines the number of traces that the query needs to match. Then, a query q matches the database D, if $supp(q, D) \geq supp_G(D)$.

Independent of any specific sequence database, we further define M as the set of all possible traces $t \in (\mathrm{dom}(S)^k)^+$ that match to a given query q:

$$M(q) := \left\{t \in (\mathrm{dom}(S)^k)^+ \mid q \models \tau\right\}.$$

2.2. Problem Statement

For a given sequence database D, more than one query might satisfy a given support threshold. In this case, the respective queries might be comparable and can potentially be ordered according to their strictness, as follows.

Definition 2.2 (Strictness). *Let* q_1, q_2 *be queries. We say that* q_1 *is stricter than* q_2*, denoted as* $q_1 \prec q_2$*, if* $M(q_1) \subseteq M(q_2)$.

In event query discovery, it is sufficient to focus on the strictest queries, assuming that they provide the most exact characterizations of how the situation of interest manifests in an event stream. As such, we summarize the respective problem as follows:

Problem 1. *Given a sequence database D and a support threshold $supp_G(D)$, find a set of queries Q such that Q is:*

$$\begin{aligned}
&\textit{correct: } q \in Q \implies supp(q,T) \geq supp_G(T),\\
&\textit{minimal: } q_1 \in Q, q_1 \prec q_2 \implies q_2 \notin Q,\\
&\textit{complete: } q \textit{ is correct and minimal} \implies q \in Q.
\end{aligned}$$

3. Related Work

Closest to our work are iCEP [9] and the IL-Miner [4], which both address the problem of event query discovery. Yet, both algorithms take ad-hoc design choices and are also limited in the considered types of queries. iCEP cannot discover queries, in which types occur multiple times. The IL-Miner, in turn, cannot discover pure pattern queries, as it constructs queries from frequent sequences of types.

Also, the use of machine learning algorithms to anticipate situations of interest received much attention recently [8, 12, 10]. For example, representation learning can help to construct probabilistic state machine patterns [8], which enable a prediction about the occurrence of a critical situation. The main drawback of such machine learning algorithms is the lack of traceability and explainability of the derived predictions.

Furthermore, event query discovery is linked to research on sequential pattern mining. In general, frequent sequence mining [1, 15, 14] limits the output sequences to the type-level. Pattern discovery has also been investigated for streams of time-series data [11, 13]. However, all of these methods were designed to work on only types or scalar values, so that they do not aim at finding correlation criteria as encoded with variables.

Another remotely related field is the correlation analysis in streaming data [6]. Here, the output are multivariate correlations rather than event queries.

4. Towards a Solution

This section gives an overview of our initial research results. We first elaborate on the design space for discovery algorithms, before introducing an algorithmic template and its exemplary instantiation. We close with initial experimental results.

4.1. Design Space for Algorithms

We have identified four dimensions to consider in the design of discovery algorithms that search the space of candidate queries:

1. Direction: bottom-up/top-down.
2. Strategy: depth-first-search/breadth-first-search.
3. Construction: type/pattern queries, unified or separated.
4. Domains: unified or separated.

Below, we explain these dimensions in detail.

Direction. When traversing the query search space, two directions may be considered: bottom-up approaches start with the most generic query and, by adding types or variables to the query string, stricter queries are generated. This traversal stops, when a query does not match the sequence database. On the other hand, top-down approaches start with the most specific query, i.e., the shortest trace of the sequence database. Here, we explore the search space by deleting types or exchanging them with variables; stopping whenever a matching query was found.

Strategy. Adopting a depth-first search strategy, the search space is explored by relaxing/tightening the query string until a query with a different matching behavior is reached (the desired matching behavior depends on the search direction). With breadth-first search, we start with matching all queries of the same length before continuing with queries of the next level (the query string becomes longer or shorter depending on the search direction).

Construction. Another algorithmic choice is whether to discover type queries and pattern queries within the same search space, or rather create two search spaces, explore them independently, and finally merge the results to obtain also the strictest mixed queries. Note that the isolated discovery of type queries corresponds to the common problem of frequent sequence mining.

Domains. Similar to the query construction, we can build the query search space over all domains simultaneously, or separate the attribute domains in the discovery process and merge the resulting queries per to obtain the final result.

4.2. Discovery Phases

For the design of a concrete algorithm, we propose a template that structures the discovery process into the following phases:

- Query candidate generation
- Redundancy check
- Matching
- Query set filtering

Algorithm 1: Exemplary discovery algorithm

```
Input: Sequence database $D$
Output: Complete set of correct and minimal queries $Q$
1  $Q \leftarrow \emptyset$
2  $A \leftarrow$ set of supported types occurring in $D$
3  $\mathbb{D} \leftarrow \emptyset$   // dictionary with query strings and their matching behavior
4  $O \leftarrow \{\}$   // stack for queries to be matched; initially the empty query
5  while $O \neq \emptyset$ do
6      $q \leftarrow O$.pop()
7      $R, M \leftarrow$ CheckRedundancy($q, \mathbb{D}$)
8      if $R = False$ then $M \leftarrow$ MatchQuery($q, D$)
9      if $M = True$ then
10         $Q \leftarrow Q \cup \{q\}$
11         $O \leftarrow$ NextQueries($q, A$)
12     $Q \leftarrow$ FilterQueries($Q$)
13 return $Q$
```

While the specific realization of these phases depends on the design choices taken along the above dimensions, the phases yield a template for a collection of discovery algorithms.

Query candidate generation. This phase generates queries to match against the sequence database. Since the query search space may grow exponentially, any realization of this phase shall meet the following constraints in order to avoid unnecessary computation:

1. Every matching query is generated.
2. Every query is generated at most once.

This way, it is ensure that all strictest queries can be discovered.

Redundancy check. Matching a query against the whole sequence database is time consuming. Hence, in this phase, we check if matching is actually necessary or whether the matching behavior can be deduced from previously matched queries. There are two possible scenarios that we can exploit. Given two queries q_1, q_2 and a sequence database D and let $q_1 \prec q_2$. If we check query q_2, but, during the discovery process, have already successfully matched a stricter query q_1 to the sequence database, then query q_2 is known to match. Similarly, if q_2 does not match the sequence database, then any stricter query will also not match.

Matching. The matching problem for our query model is known to be NP-complete [7]. Yet, since we often match queries that are similar, we might not always have to match the whole query against the entire sequence database.

Query set filtering. Finally, we have to remove matching queries for which stricter queries that also match have been found.

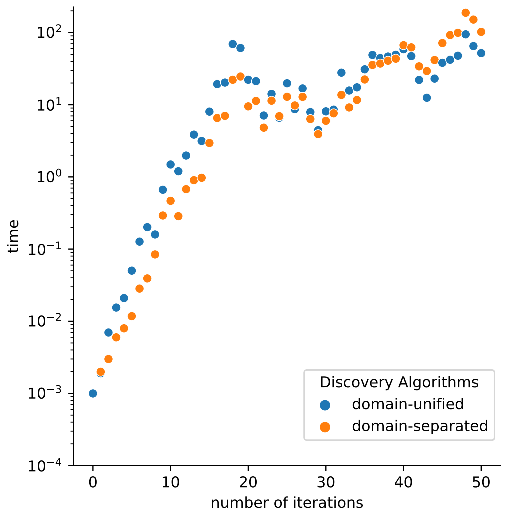

Figure 2: Runtime of discovery for increasing search spaces.

4.3. Exemplary Instantiation

In Alg. 1, we illustrate a specific discovery algorithm that adopts bottom-up, depth-first search, where pattern and type queries as well as all domains are considered in the same search space. It initially puts an empty query onto the stack O. The loop will continue until there are no further queries to be matched on the stack. In each loop, the query is checked for redundancy and, if necessary, the matching function will be called. Only if the query matches, possibly new queries will be added to the stack by adding the result of the function NextQueries. When the whole query search space has been explored, the query filtering function ensures that only the strictest queries are returned.

Matching algorithm. For each query that we match, we save for each trace the last matching position of the first match. For a stricter query, we can then rely on this information, so that only the newly added part of the query needs to be matched on the remaining part of the traces. For queries containing patterns, we first replace the newly added pattern by all possible types and then match the maintained type queries as described before. As such, we leverage the similarity of the considered queries, thereby avoiding the need to match the same substring over and over again.

4.4. Experimental Results

To achieve a controlled setup for evaluating the design choices in discovery algorithms, we proceed as follows. Since the size of the sequence database (number of traces and length of traces) is of minor importance, unlike the type distribution, we initialize a small, synthetic database with two traces of length ten with three domains. Each type of each domain occurs only once in the whole database, so that initially, there is no query to discover. We then adapt the database in 50 iterations, each time picking a trace position and domain randomly. Then, we either copy the type of one trace into all the other traces to generate matches for a type query, or we copy a type in each trace to another random position to generate matches for a pattern query. As such, in each iteration, we increase the result size and also the query search space.

Fig. 2 illustrates the runtime of a Python implementation of two variants of the discovery algorithms (bottom-up, depth-first, unified construction) that treat domains separately or unified. The runtime increases with the number of iterations. Also, domain-separated discovery performs better for smaller search spaces, whereas it is outperformed by domain-unified discovery for larger ones. This result underlines the need to explore the respective design choices in a systematic manner and highlights the potential of linking them to properties of the sequence database.

5. Conclusions and Next Steps

In this paper, we introduced the query discovery problem and pointed out different dimensions for the design of discovery algorithms. We also exemplified the design space with a specific algorithm and initial experimental results.

As a next step, we intend to explore in detail the relation between the outlined design choices for discovery algorithms and properties of sequence databases. This way, we hope to obtain a model that enables us to predict, which discovery algorithm shall be adopted in a given application scenario. Moreover, we strive for a characterization of the aspects of a sequence database (e.g., in terms of specific types or traces) that render discovery intractable and, hence, may be subject to some preprocessing of the database (e.g., by filtering particular types).

Acknowledgments. The work was supported by the German Research Foundation (DFG), project 414984028, CRC 1404 FONDA.

References

[1] Rakesh Agrawal and Ramakrishnan Srikant. 1995. Mining sequential patterns. In *ICDE*. IEEE, 3–14.

[2] Yagil Engel, Opher Etzion, and Zohar Feldman. 2012. A basic model for proactive event-driven computing. In *DEBS*. ACM, 107–118.

[3] Minos Garofalakis, Rajeev Rastogi, and Kyuseok Shim. 2002. Mining sequential patterns with regular expression constraints. *IEEE TKDE* 14, 3 (2002), 530–552.

[4] Lars George, Bruno Cadonna, and Matthias Weidlich. 2016. IL-Miner: Instance-Level Discovery of Complex Event Patterns. *Proc. VLDB End.* 10, 1 (2016), 25–36.

[5] Nikos Giatrakos, Elias Alevizos, Alexander Artikis, Antonios Deligiannakis, and Minos N. Garofalakis. 2020. Complex event recognition in the Big Data era: a survey. *VLDB J.* 29, 1 (2020), 313–352.

[6] Tian Guo, Saket Sathe, and Karl Aberer. 2015. Fast Distributed Correlation Discovery Over Streaming Time-Series Data. In *CIKM*, ACM, 1161–1170.

[7] Sarah Kleest-Meißner, Rebecca Sattler, Markus L. Schmid, Nicole Schweikardt, and Matthias Weidlich. 2022. Discovering Event Queries from Traces: Laying Foundations for Subsequence-Queries with Wildcards and Gap-Size Constraints. In *ICDT*, *LIPIcs*, Vol. 220. 18:1–18:21.

[8] Yan Li and Tingjian Ge. 2021. Imminence Monitoring of Critical Events: A Representation Learning Approach. In *SIGMOD*. ACM, 1103–1115.

[9] Alessandro Margara, Gianpaolo Cugola, and Giordano Tamburrelli. 2014. Learning from the Past: Automated Rule Generation for Complex Event Processing. In *DEBS*, ACM, 47–58.

[10] Raef Mousheimish, Yehia Taher, and Karine Zeitouni. 2017. Automatic Learning of Predictive CEP Rules: Bridging the Gap between Data Mining and Complex Event Processing. In *DEBS*, ACM, 158–169.

[11] Spiros Papadimitriou, Jimeng Sun, and Christos Faloutsos. 2005. Streaming pattern discovery in multiple time-series. (2005).

[12] Mehmet Ulvi Simsek, Feyza Yildirim Okay, and Suat Ozdemir. 2021. A deep learning-based CEP rule extraction framework for IoT data. *The Journal of Supercomputing*, 77, 8 1–30.

[13] Machiko Toyoda, Yasushi Sakurai, and Yoshiharu Ishikawa. 2013. Pattern discovery in data streams under the time warping distance. *VLDB J.* 22, 3, 295–318.

[14] Jianyong Wang and Jiawei Han. 2004. BIDE: Efficient mining of frequent closed sequences. In *ICDE*. IEEE, 79–90.

[15] Xifeng Yan, Jiawei Han, and Ramin Afshar. 2003. CloSpan: Mining: Closed sequential patterns in large datasets. In *Proc. of SIAM Int'l Conf. on Data Mining*. SIAM, 166–177.

[16] Haopeng Zhang, Yanlei Diao, and Neil Immerman. 2014. On complexity and optimization of expensive queries in complex event processing. In *SIGMOD*. ACM, 217–228.