DISCES: Systematic Discovery of Event Stream Queries

Technical Report

Rebecca Sattler
Humboldt-Universität zu Berlin
Berlin, Germany
rebecca.sattler@informatik.hu-berlin.de

Sarah Kleest-Meißner
Humboldt-Universität zu Berlin
Berlin, Germany
kleemeis@informatik.hu-berlin.de

Steven Lange
Humboldt-Universität zu Berlin
Berlin, Germany
langestx@hu-berlin.de

Markus L. Schmid
Humboldt-Universität zu Berlin
Berlin, Germany
MLSchmid@MLSchmid.de

Nicole Schweikardt
Humboldt-Universität zu Berlin
Berlin, Germany
schweikn@informatik.hu-berlin.de

Matthias Weidlich
Humboldt-Universität zu Berlin
Berlin, Germany
matthias.weidlich@hu-berlin.de

ABSTRACT

The continuous evaluation of queries over an event stream provides the foundation for reactive applications in various domains. Yet, the precise specification of a query that detects a situation of interest is challenging, since knowledge about the event patterns to match by a query is typically only partially available. However, once a database of finite, historic (sub-)streams that include a materialization of the situation of interest is available, one may aim at automatic discovery of the respective queries. Existing algorithms for event query discovery, however, incorporate ad-hoc design choices and it is unclear how their suitability for a given database shall be assessed.

In this paper, we address this gap with DISCES, an algorithmic framework for event query discovery. DISCES outlines a design space for discovery algorithms, thereby making the design choices explicit. We instantiate the framework to derive four specific algorithms, which all yield correct and complete results, but differ in their runtime sensitivity. We therefore also provide guidance on how to select one of the algorithms for a given database based on a few of its essential properties. Our experiments using simulated and real-world data illustrate that our algorithms are indeed tailored to databases showing certain properties and solve the query discovery problem several orders of magnitude faster than existing approaches.

Artifact Availability:
The source code, data, and/or other artifacts have been made available at https://github.com/rebesatt/disces.

1 INTRODUCTION

Systems for complex event processing (CEP) continuously evaluate a set of queries over streams of event data [7, 14]. As such, they facilitate the recognition of situations of interest that materialize as event patterns, thereby enabling reactive and proactive applications in various domains, including, for instance, supply chain management [17], urban transportation [4], and computational finance [6].

CEP systems provide a rich model for the specification of event queries [3]. Queries typically define conditions over the attribute values of events to establish their relevance for a pattern and to correlate them; and they impose constraints on the ordering of events and their occurrence within a certain window over the stream.

While event queries enable a comprehensive characterization of a situation of interest, their specification is challenging in practice. Domain experts may have a basic understanding of factors that contribute to a situation that shall be detected by a query, but typically lack full knowledge about the specific event patterns, in particular in predictive applications [9, 27]. Here, the occurrence of a situation (e.g., a delay in a supply chain or a fraudulent transaction) is anticipated in order to prevent or mitigate it.

To aid in the specification of event queries, approaches for automated query discovery have been proposed [13, 21]. They assume that a database of finite, historic (sub-)streams, each including at least one materialization of the situation, is available. Then, queries that match the streams and, hence, generalize the historic observations, may be inspected and refined by domain experts to construct a suitable set of queries. Unlike machine learning approaches for pattern detection, the discovery of event queries provides a traceable and explainable characterization of a situation of interest.

Solving the problem of event query discovery is computationally hard, though. The search space of candidate queries grows exponentially in the query length and the size of the domain of the payload data. Existing discovery approaches [13, 21] explore this space in *one* specific way, which may be suitable for one database, but leads to intractability for another one. Yet, the assumptions on the database that motivate the taken design choices are implicit, so that it is unclear for which database an algorithm can be expected to work.

In this paper, we argue for the systematic design of algorithms for event query discovery. We aim to answer the following question:

(i) *What are the design choices in query discovery?*
(ii) *How to choose a design for a given database?*
(iii) *How to provide feedback on the algorithmic performance?*

We address these questions through three contributions that we summarize, along with the paper structure following a problem formalization (§2), as follows:

(1) We present DISCES (§3) as a framework for the design of algorithms for event query discovery. It captures a set of fundamental design choices to define discovery algorithms.
(2) We instantiate the framework to derive four discovery algorithms (§4). They all provide correct and complete results, yet they differ in their exploration of the space of candidate queries.

(3) We provide means to guide event query discovery (§5). First, we show how to choose among our algorithms based on a few essential properties of a given stream database. Second, we provide hints on how to resolve intractability or ineffectiveness of discovery based on abstractions of the events' payload data.

We evaluated our techniques in experiments using simulated and real-world data (§6). Our results illustrate that the algorithms designed as part of the DISCES framework are indeed tailored to databases that show certain properties. They solve the query discovery problem several orders of magnitude faster than existing approaches, with runtimes that are comparable to strategies that approximate the result. We close with a review of related work (§7) and conclusions (§8).

2 PROBLEM SETTING

We first discuss scenarios for event query discovery (§2.1). Next, we introduce a model for queries over event streams (§2.2), before turning to the definition of the query discovery problem (§2.3).

2.1 Motivating Application Scenarios

We consider three scenarios for event query discovery, as follows:

(1) *Urban transportation:* Solutions to automated traffic management may exploit event data that captures spatial-temporal information on vehicles [4]. Here, event queries help to identify situations such as a delay on certain bus routes [32]. Similarly, queries may relate to event patterns that indicate surges in the pricing adopted by taxi drivers. Either way, event queries need to correlate multivariate events along several dimensions, such as bus routes, levels of accumulated delays, or taxi types. Yet, the exact materialization of the situations of interest is likely to be unknown, so that historic streams provide an opportunity to discover a first set of respective queries, which may be refined subsequently.

(2) *Finance:* Applications in computational finance rely on events that indicate stock price changes and fulfilled trades [6]. Here, event queries may detect situations that indicate high-risk developments for a specific portfolio, outline opportunities for beneficial trades, or event indicate fraudulent behaviour. Such queries typically span the joint occurrence of events in a particular order, with the need to correlate them based on their payload data, e.g., per stock symbol, level of trade volume, or risk class. Again, knowledge on the respective patterns is typically only partially available, so that the discovery of event queries from historic streams provides valuable insights.

(3) *Cluster monitoring:* The continuous supervision of technical infrastructures is often approached using CEP systems. For instance, consider the case of compute clusters, for which events denote state transitions of tasks, the availability of resources, or important changes in resource utilization [24]. Situations of interest include the build-up of massive resource demands or the presence of stragglers. Those situations may be characterized by patterns that include a specific order of events that indicate state changes of tasks as well as the presence of particular types of resources, potentially correlated by scheduling priorities, resource types, or user classes. Event query discovery based on event streams of past occurrences of the situations helps in the definition of the respective queries.

2.2 Event Streams and Queries

Event streams. We base our work on a model of multivariate event streams, similar to the one adopted in relational data stream processing [2]. It is based on the notion of an *event schema*, which is a tuple of attributes $\mathcal{A} = (A_1, \ldots, A_n)$. Each attribute A_i, $1 \leq i \leq n$, is of a primitive data type, for which the finite domain is denoted by $\text{dom}(A_i)$. Without loss of generality, we assume that there exists a single event schema and that all domains are distinct, i.e., $\cap_{i=1}^{n} \text{dom}(A_i) = \emptyset$. The schema includes an alphabet $\Gamma = \bigcup_{i=1}^{n} \text{dom}(A_i)$, i.e., the set of all attribute values. An *event* $e = (a_1, \ldots, a_n)$ is a tuple of attribute values that instantiates the event schema, i.e., $a_i \in \text{dom}(A_i)$ for $1 \leq i \leq n$, and we write $e.A_i$ to refer to the value of attribute A_i of event e. The starting point for query discovery is an *event stream*, a finite sequence of events $S = \langle e_1, \ldots, e_l \rangle$, with $|S|$ denoting its length, which is typically a subsequence of a potentially unbounded sequence of events. Here, we write $S[i]$ for the i-th event in the stream. A *stream database* is a set of streams $D = \{S_1, \ldots, S_d\}$, which are potentially overlapping in their contained events. A stream database D induces the supported stream alphabet $\Gamma_D \subseteq \Gamma$, containing all values that occur in every stream, i.e., $\Gamma_D = \{a \in \Gamma \mid \forall\, S \in D : \exists\, j \in \{1, \ldots, |S|\}, i \in \{1, \ldots, n\} : S[j].A_i = a\}$.

Event queries. Queries over event streams typically describe sequences of events that are potentially correlated by predicates over their attribute values [14, 33]. Many models also include a time window for the occurrence of the events. However, in our setting of event query discovery, the temporal context is given in terms of the separate streams of a stream database, which is why we do not model a time window explicitly. Moreover, most existing models formalize queries based on a sequence of variables and a set of predicates over these variables. However, we adopt a linearized representation, inspired by [16], that is better suited to describe discovery algorithms. Let $\mathcal{A} = (A_1, \ldots, A_n)$ be an event schema, $\mathcal{X}$ a finite set of variables with $\mathcal{X} \cap \Gamma = \emptyset$, and $\_ \notin \mathcal{X} \cup \Gamma$ a placeholder symbol. Then, a query term $t = (u_1, \ldots, u_n)$ is an n-tuple built of attribute values, variables, and placeholders, i.e., $u_j \in \text{dom}(A_j) \cup \mathcal{X} \cup \{\_\}$, $1 \leq j \leq n$, such that not all its components are placeholders, i.e., it holds that $u_j \in \text{dom}(A_j) \cup \mathcal{X}$ for at least one $1 \leq j \leq n$. The empty query term ε is the n-tuple comprising only placeholders, i.e., $\varepsilon = (\_, \ldots, \_)$. An event query is a finite sequence of query terms, $q = \langle t_1, \ldots, t_k \rangle$, with $q[i]$ denoting the i-th term. In a query, variables need to occur in at least two query terms for the same attribute, i.e., for any term $q[i] = (u_1, \ldots, u_n)$ with $u_j \in \mathcal{X}$, $1 \leq i \leq k$ and $1 \leq j \leq n$, there exists another query term $q[p] = (u'_1, \ldots, u'_n)$ with $u_j = u'_j$, $1 \leq p \leq k$ and $p \neq i$. The empty query is defined as $\langle \varepsilon \rangle$, i.e., it contains only the empty query term. The universe of all possible queries is $\mathcal{Q}$.

Intuitively, each query term characterizes an event that should be matched by the query. For each attribute, the term enforces a distinct value or permits any value of the respective domain. In the latter case, a placeholder models the absence of a constraint, whereas a variable that is used multiple times enforces equal attribute values. This way, equivalence predicates over attributes of events are modelled.

Queries built of terms that contain only attribute values (and placeholders) are called *type queries*; those built of terms of only variables (and placeholders) are called *pattern queries*; while those that combine attribute values and variables are called *mixed queries*.

Formally, we define the query semantics as follows. Let $q = \langle t_1, \ldots, t_k \rangle$ be an event query and let $S = \langle e_1, \ldots, e_l \rangle$ be an event stream. A match of q in S is an injective mapping $m : \{1, \ldots, k\} \rightarrow \{1, \ldots, l\}$, such that:

- for each query term $q[i] = (u_1, \ldots, u_n)$, $1 \leq i \leq k$, the mapped event $S[m(i)] = (a_1, \ldots, a_n)$ has the required attribute values, i.e., $u_j = a_j$ or $u_j \in \mathcal{X} \cup \{\_\}$ for $1 \leq j \leq n$;
- variables are bound to the same attribute values, i.e., for query terms $q[i] = (u_1, \ldots, u_n)$ and $q[p] = (u'_1, \ldots, u'_n)$, $1 \leq i, p \leq k$, it holds that $u_j \in \mathcal{X}$, $1 \leq j \leq n$, with $u_j = u'_j$ implies that $S[m(i)].A_j = S[m(p)].A_j$; and
- the order of the events in the stream is preserved, i.e., for $1 \leq i < p \leq k$, it holds $m(i) < m(p)$.

If there exists a match for query q in stream S, we say that S supports q (q matches S); and write $S \models q$. Table 1 lists our notations.

Example 1 - Stream Database

Consider the following stream database with the three streams S_1, S_2, S_3:

Stream	Event	Job Id	Status	Priority
S_1	e_{11}	1	schedule	low
	e_{12}	1	kill	low
	e_{13}	1	schedule	high
	e_{14}	1	finish	high
S_2	e_{21}	2	schedule	low
	e_{22}	3	schedule	high
	e_{23}	2	evict	low
S_3	e_{31}	4	schedule	high
	e_{32}	5	schedule	low
	e_{33}	4	finish	high
	e_{34}	6	schedule	low
	e_{35}	5	evict	low

The following query $q = \langle (x_0, \_, \_), (\_, \_, \text{high}), (x_0, \text{evict}, \_) \rangle$ describes three consecutive events (but not necessarily *immediately* consecutive) in which the first and the last event have the same job id, the priority of the second event is high and the status of the last event is evict. The query does not match S_1, as there is no event with an evict status. Query q matches S_2 and S_3, with e_{21}, e_{22}, e_{23} and e_{32}, e_{33}, e_{35}, respectively.

2.3 Query Discovery

Given a stream database, the problem of event query discovery relates to the identification of event queries that are supported by all streams, i.e., for which there exists at least a single match for each stream. However, there may exist queries that are supported by all streams, but which are comparable in the sense that one of them is stricter than another one. Formally, a query q is defined as stricter than a query q', denoted by $q \prec q'$, if (i) for any possible stream S (not necessarily contained in a given stream database), $S \models q$ implies $S \models q'$, and (ii) there exists a stream S, such that $S \models q'$, but $S \not\models q$.

In event query discovery, we are only interested in the strictest queries that are supported by all streams. The reasoning being that these queries denote the most precise characterization of the patterns of events that indicate a situation of interest. For a stream database D, we therefore consider a notion of descriptiveness [15]: A query q is descriptive for D, if it is supported by D and there does not exist a query q' that is stricter, $q' \prec q$, and that is also supported by the stream database D. Based thereon, we formulate the problem of event query discovery:

Table 1: Overview of notations.

Notation	Explanation
$\mathcal{A} = (A_1, \ldots, A_n)$	An event schema, a tuple of attributes
$\Gamma = \bigcup_{i=1}^{n} \text{dom}(A_i)$	The stream alphabet
$e = (a_1, \ldots, a_n)$	An event, a tuple of attribute values
$e.A_i$	The value a_i of attribute A_i of event e
$S = \langle e_1, \ldots, e_l \rangle$	An event stream, a finite sequence of events
$S[i]$	The i-th event of the event stream S
$D = \{S_1, \ldots, S_d\}$	A stream database, a set of event streams
Γ_D	The supported stream alphabet of stream database D
$\mathcal{X}$	A finite set of query variables, $\mathcal{X} \cap \Gamma = \emptyset$
$\_$	A placeholder symbol, $\_ \notin \mathcal{X} \cup \Gamma$
$t = (u_1, \ldots, u_n)$	A query term, an n-tuple of attribute values, variables, and placeholders with $u_j \in \text{dom}(A_j) \cup \mathcal{X} \cup \{\_\}$, $1 \leq j \leq n$, such that not all its components are placeholders
$\varepsilon = (\_, \ldots, \_)$	The empty query term, an n-tuple of placeholders
$q = \langle t_1, \ldots, t_k \rangle$	An event query, a finite sequence of query terms
$q[i]$	The i-th term of the query q
$\langle \varepsilon \rangle$	The empty query containing only the empty query term
$S \models q$	The stream S supports the query q, i.e., there exists a match

Problem 1 (Event Query Discovery). *Given a stream database $D = \{S_1, \ldots, S_d\}$, the problem of event query discovery is to construct the set of queries Q, such that:*

- *Q is correct: each $q \in Q$ creates at least a single match for all streams, i.e., for all $q \in Q$ and $S \in D$ it holds that $S \models q$;*
- *Q is descriptive: only the strictest queries are considered, i.e., for all $q, q' \in Q$ it holds that neither $q \prec q'$ nor $q' \prec q$;*
- *Q is complete: if a query q is both correct and descriptive, then it holds that $q \in Q$.*

Example 2 - Descriptive Query

Let us consider the following two queries:

$$q_1 = \langle (x_0, \text{schedule}, \text{low}), (\_, \_, \text{high}), (x_0, \_, \_) \rangle$$
$$q_2 = \langle (x_0, \text{schedule}, \text{low}), (x_0, \_, \_) \rangle.$$

Both queries match the stream database of Example 1. Also, q_1 is stricter than q_2, as any stream that matches q_1 will also match q_2, but not vice versa. Query q_1 is descriptive, since there is no other query which is stricter than q_1 and matches the stream database D from Example 1.

3 THE DISCES FRAMEWORK

This section introduces DISCES as a framework for the systematic design of algorithms to address the problem of event query discovery. We first discuss dimensions along which design choices are captured (§3.1), before turning to their combination (§3.2).

3.1 Dimensions of Design Choices

The DISCES framework includes four dimensions for design choices that guide how the space of candidate queries is explored. These dimensions, illustrated in Table 2, refer to the following properties of the search through that space, which are detailed in the remainder:

Direction: The search proceeds bottom-up or top-down.

Strategy: The approach is depth-first or breadth-first.

Construction: Type and pattern queries are initially constructed separately, or mixed queries are immediately considered.

Attributes: Attributes are initially considered separately, or immediately incorporated comprehensively.

Table 2: Illustration of the dimensions that are incorporated in the design of discovery algorithms as part of the DISCES framework.

Direction	Strategy	Construction: Separated | Unified			Attributes: Separated | Comprehensive		
$\langle(a,5,11)\rangle$	$\boxed{\underline{\langle(a,5)\rangle}}$	$\langle(x,y),(x,y)\rangle$	$\langle(a,5)\rangle$	$\langle(x,5),(x,\_)\rangle$	$\langle(x),(a),(x)\rangle$	$\langle(5),(7)\rangle$	$\langle(x,5),(x,7)\rangle$
bottom up $\uparrow$: $\downarrow$ top down	$\uparrow$	$\uparrow$	$\uparrow$	$\uparrow$	$\uparrow$	$\uparrow$	$\uparrow$
	$\boxed{\underline{\langle(a,\_)\rangle}; \langle(\_,5)\rangle}$	$\langle(x,\_),(x,\_)\rangle$	$\langle(a,\_)\rangle$	$\langle(x,\_),(x,\_)\rangle$	$\langle(x),(x)\rangle$	$\langle(5)\rangle$	$\langle(x,5),(x,\_)\rangle$
$\langle(a,\_,\_)\rangle$	$\underline{\text{DFS}}$ | $\boxed{\text{BFS}}$	pattern queries	type queries	mixed queries	attribute 1	attribute 2	all attributes

Direction. When traversing the space of candidate queries, two directions may be considered: Bottom-up approaches start with the most generic query possible and, by adding attribute values or variables, generate stricter queries. The traversal stops, once the sequences in the database no longer support the explored queries. Top-down approaches start with a most specific query, i.e., a shortest sequences of the given database. Then, they explore the search space by deleting attribute values or exchanging them with variables; stopping whenever queries that are supported by all sequences of the database have been found. Considering the example in Table 2, a bottom-up approach evaluates the query $\langle(a,\_,\_)\rangle$ before the more specific query $\langle(a,5,11)\rangle$, and vice versa for a top-down approach.

Strategy. To explore candidate queries, one may adopt a depth-first search (DFS) strategy. Then, the space is traversed by generalizing or specializing the queries until a query with a different support behavior is reached, i.e., a non-supported query is generalized until it is supported (top-down direction) or a supported query is specialized until it is no longer supported (bottom-up direction). A different strategy is to adopt breadth-first search (BFS), i.e., to first explore all queries containing an equal number of non-placeholders, before continuing with those with less (top-down direction) or more (bottom-up direction) non-placeholders. In Table 2, assuming a bottom-up search direction, from query $\langle(a,\_)\rangle$, DFS would continue with the stricter query $\langle(a,5)\rangle$, whereas BFS would first consider queries with the same number of non-placeholders, such as $\langle(\_,5)\rangle$.

Construction. Another algorithmic choice is whether to construct type queries and pattern queries separately or with a unified approach. The former means that the space of candidate type queries and the space of candidate pattern queries are explored independently, before merging the results to also obtain the descriptive mixed queries. Note that the isolated discovery of type queries corresponds to the common problem of maximal frequent sequence mining [5, 11, 12]. In a unified approach, in turn, all query structures are explored as part of a single search space. In Table 2, a separated approach would explore pattern queries, e.g., $\langle(x,\_),(x,\_)\rangle$ and subsequently $\langle(x,y),(x,y)\rangle$, and type queries, e.g., $\langle(a,\_)\rangle$ and then $\langle(a,5)\rangle$ before merging them. A unified approach would directly construct mixed queries and explore, for instance, $\langle(x,5),(x,\_)\rangle$ after $\langle(x,\_),(x,\_)\rangle$.

Attributes. Similarly, the attributes of an event schema induce a design choice for the exploration of the space of candidate queries. We may first explore each attribute separately, before merging the results to obtain the final set of descriptive queries; or rely on a comprehensive approach that explores all attributes simultaneously. Again, Table 2 illustrates this design choice: We may consider the first attribute, exploring $\langle(x),(x)\rangle$ followed by $\langle(x),(a),(x)\rangle$, and the second attribute, exploring $\langle(5)\rangle$ followed by $\langle(5),(7)\rangle$ and merging them afterward; or immediately consider both attributes, by exploring $\langle(x,5),(x,\_)\rangle$ and then $\langle(x,5),(x,7)\rangle$.

3.2 Combination of Design Choices

Having described fundamental choices in the design of discovery algorithms, we review their interplay and underlying assumptions.

First, the choice regarding the search direction, top-down vs. bottom-up, relates to an important assumption on the application scenario. A top-down search strategy will commence with a shortest stream of the database as a query, and step-wise generalize it until queries supported by the whole database are found. Hence, such an approach can be expected to work efficiently, if descriptive queries are only slightly shorter than the shortest stream in the database. In the scenarios outlined in §2.1, however, queries commonly contain solely a few terms and are generally much shorter than the available streams. Hence, a top-down exploration of candidate queries will quickly become intractable, due to the sheer size of the respective search space. In the remainder, we therefore focus on the instantiation of algorithms that adopt a bottom-up direction for the search.

Second, we focus on the combination of search strategies (DFS vs. BFS) and the construction approach (type/pattern-separated vs. unified). As mentioned above, the separate construction of type queries and pattern queries enables us to incorporate existing results: The discovery of type queries in isolation corresponds to the maximal frequent sequence mining (MFSM) problem [1]. Since state-of-the-art algorithms for the MFSM problem rely on BFS, we adopt it as the search strategy in any approach that is based on the separate construction of type queries and pattern queries.

In contrast, for the unified construction of queries, a BFS search strategy is harmful. A bottom-up, unified construction of queries following BFS will explore mixed queries, for which it is known that they cannot be descriptive. For instance, the mixed queries $\langle(a),(a),(x_0),(x_0)\rangle$ and $\langle(x_0),(x_0),(b),(b)\rangle$ will be considered as candidates if the queries $\langle(a),(a),(b),(b)\rangle$ and $\langle(x_0),(x_0),(x_1),(x_1)\rangle$ are supported by the stream database, even though the mixed queries cannot be descriptive. A separated construction of queries avoids the issue, as does the combination of a unified construction with DFS.

Third, the question whether to consider the attributes separately or comprehensively is largely orthogonal to the above design choices. Hence, the separate or comprehensive treatment of attributes may be combined with either of the above design choices.

Based thereon, we derive the four combinations of design choices listed in Table 3, which can be deemed suitable to design efficient algorithms for the problem of event query discovery.

Table 3: Combination of design choices for discovery algorithms.

	Direction	Strategy	Construction	Attributes
B-S-S	bottom-up	$\underline{\text{B}}$FS	$\underline{\text{S}}$eparated	$\underline{\text{S}}$eparated
B-S-C	bottom-up	$\underline{\text{B}}$FS	$\underline{\text{S}}$eparated	$\underline{\text{C}}$omprehensive
D-U-S	bottom-up	$\underline{\text{D}}$FS	$\underline{\text{U}}$nified	$\underline{\text{S}}$eparated
D-U-C	bottom-up	$\underline{\text{D}}$FS	$\underline{\text{U}}$nified	$\underline{\text{C}}$omprehensive

4 DISCES ALGORITHMS

Having introduced the DISCES framework to capture important choices in the design of discovery algorithms, we now turn to its instantiation. We first introduce algorithms to generate query candidates (§4.1), to match query against a stream database (§4.2), and to merge queries (§4.3). Based thereon, we propose four specific discovery algorithms (§4.4) that realize the above design choices.

4.1 Query Candidate Generation

Our discovery algorithms rely on an iterative generation of query candidates, which are then matched against a stream database. As detailed in §3, we follow a bottom-up direction in the exploration of candidate queries and step-wise generate stricter queries for a given query q. That is, the child queries of q are those obtained by inserting a new variable, by inserting a variable that has been present already, or by inserting an attribute value; either way, considering an existing query term or a newly added query term.

Variables are inserted in a certain order, which is reflected in a total order $<$ over a set of variables $\mathcal{X}$. To simplify the notation, we write $\mathcal{X}^<$ for the sequence of variables induced by $<$ over $\mathcal{X}$. Also, our construction employs at most $s/2$ variables with $s = \min_{S \in D} |S|$ being the minimal stream length in the stream database. Hence, the size of $\mathcal{X}$ and the length of $\mathcal{X}^<$ are bounded.

Query candidate generation, formalized in Alg. 1, takes as input a query q; an event schema $\mathcal{A}$; a supported alphabet Γ_D; a parent dictionary P that is modelled as a function $P : Q \rightarrow Q$, mapping child queries to their parent; a sequence of variables $\mathcal{X}^<$; and a Boolean flag b to control whether variables shall be inserted. The algorithm returns the set of child queries for q.

Alg. 1 is divided into three parts, following some initialization (line 1 - 4), each of which captures one of the three aforementioned insertions. Those are designed such that *all* child queries that can be obtained by insertion of a single symbol (attribute value or variable) are constructed *exactly once*. The generation relies on the auxiliary function Next (line 20 - 31). Given a query, it generates query candidates by inserting a given symbol (variable or attribute value) in any query term following a given position in the query, if the query contains a placeholder for that attribute. In addition, it also generates candidates by appending a new query term to the query that is empty except for the given symbol for the given attribute. Based thereon, the three parts of the main algorithm include:

Insertion of a new variable (line 7 - 10). For each attribute, we consider all query terms after the last occurrence of any attribute value (g) and after the first occurrence of the last inserted variable (f). Using function Next, a single new variable $\mathcal{X}^<[v+1]$ is introduced by replacing a placeholder or adding a new query term.

Insertion of existing variable (line 13 - 15). To avoid redundant generation of child queries, this step is realized solely if the last insertion into the query had been the last inserted variable, i.e., $s = \mathcal{X}^<[v]$. Again, insertion of the variable is realized using Next.

Insertion of attribute value (line 16 - 18). For each query term after the last occurrence of any attribute value (g) and after the first occurrence of the last inserted variable (f), and each attribute value in the supported alphabet, we insert a supported attribute value of the respective attribute. Again, this is realized by replacing a placeholder or by appending a query term.

Algorithm 1: ChildQueries: Query Cand. Generation

```
Input:  Query $q$, event schema $\mathcal{A}$, supported alphabet $\Gamma_D$,
        parent dictionary $P$, sequence of query variables $\mathcal{X}^<$,
        Boolean flag $b \in \{True, False\}$ whether to insert variables.
Output: Updated parent dictionary $P$.
1  $g \leftarrow$ position of last inserted attribute value in $q$;
2  $f \leftarrow$ first position of last inserted variable in $q$;
3  $z \leftarrow \max(g, f)$;
4  $n \leftarrow |\mathcal{A}|$;
   /* Shall variables be inserted?                                   */
5  if $b = True$ then
6      $v \leftarrow$ number of distinct query variables in $q$;
       /* Insert a new variable $\mathcal{X}^<[v+1]$                   */
7      foreach $A \in \mathcal{A}$ do
8          foreach $q' \in \text{Next}(q, z, A, \mathcal{X}^<[v+1], n)$ do
9              foreach $q'' \in \text{Next}(q', z + |q'| - |q|, A, \mathcal{X}^<[v+1], n)$ do
10                 $P(q'') \leftarrow q$;
       /* Insert last inserted variable $\mathcal{X}^<[v]$             */
11     $l \leftarrow$ last position of last inserted variable in $q$;
12     $s \leftarrow$ last inserted symbol in $q$;
13     if $s = \mathcal{X}^<[v]$ then
14         $A \leftarrow$ attribute $A \in \mathcal{A}$ with $s \in \text{dom}(A)$;
15         foreach $q' \in \text{Next}(q, l, A, \mathcal{X}^<[v], n)$ do $P(q') \leftarrow q$ ;
   /* Insert new supported attribute value                            */
16 foreach $A \in \mathcal{A}$ do
17     foreach $a \in (\Gamma_D \cap \text{dom}(A))$ do
18         foreach $q' \in \text{Next}(q, z, A, a, n)$ do $P(q') \leftarrow q$ ;
19 return $P$
   /* Function to generate the next candidate queries                 */
20 Next (query $q$, start pos. $s$, attribute $A$, symbol $y$, number of attributes $n$)
21 $Q' \leftarrow \emptyset$;
22 foreach $i \in \{s, \ldots, |q|\}$ do
       /* Generate candidate by replacing placeholder with
          given variable/attribute value $y$                           */
23     if $q[i].A = \_$ then
24         $q' \leftarrow q$;
25         $q'[i].A \leftarrow y$;
26         $Q' \leftarrow Q' \cup \{q'\}$;
       /* Generate candidate by including new query term that
          only contains given variable/attribute value $y$             */
27     $e' \leftarrow \varepsilon$;
28     $e'.A \leftarrow y$;
29     $q' \leftarrow \langle q[1], \ldots, q[i], e', q[i+1], \ldots, q[|q|] \rangle$;
30     $Q' \leftarrow Q' \cup \{q'\}$;
31 return $Q'$
```

4.2 Query Matching

As our algorithms explore the space of candidate queries incrementally, the evaluation of a query may benefit from the match results obtained for less strict queries at an earlier stage. Therefore, for each stream and query, we track the last position of the first match. For a stricter query, we then rely on this information and consider only the additional parts of a query and the remaining parts of the stream in the search for a match, thereby avoiding redundant computation.

To realize the above idea, our matching algorithm maintains a stream matches dictionary $T : Q \rightarrow \{D \rightarrow \{\Gamma^* \rightarrow \mathbb{N}\}\}$, which is a multi-level data structure storing match positions and variable bindings. The first level is a function that maps a query q to a set of functions, which, at the second level of the structure, map a stream S to another set of functions that capture the information about matches. The latter set of functions maps a sequence of attribute values $\langle a_1, \ldots a_k \rangle \in \Gamma^*$ to a position $m \in \mathbb{N}$, where k corresponds to the number of distinct variables in query q. Intuitively,

Example 3 - Candidate Generation

Let us consider the stream database D and the query q from Example 1 and assume that 'evict' was the most recently added symbol:

$$q = \langle (x_0, \_, \_), (\_, \_, \text{high}), (x_0, \text{evict}, \_) \rangle.$$

To generate new query candidates, we first add a new variable:

$$q_1 = \langle (x_0, \_, \_), (\_, \_, \text{high}), (x_0, \text{evict}, x_1), (\_, \_, x_1) \rangle$$
$$q_2 = \langle (x_0, \_, \_), (\_, \_, \text{high}), (x_0, \text{evict}, \_), (x_1, \_, \_), (x_1, \_, \_) \rangle$$
$$q_3 = \langle (x_0, \_, \_), (\_, \_, \text{high}), (x_0, \text{evict}, \_), (\_, x_1, \_), (\_, x_1, \_) \rangle$$
$$q_4 = \langle (x_0, \_, \_), (\_, \_, \text{high}), (x_0, \text{evict}, \_), (\_, \_, x_1), (\_, \_, x_1) \rangle$$

We do not insert existing variables, as it would happen only if a variable was the last inserted symbol. However, we also add supported attribute values ('schedule' for the second attribute (Status); 'high' and 'low' for the third attribute (Priority)), after the last occurrence of an attribute value:

$$q_5 = \langle (x_0, \_, \_), (\_, \_, \text{high}), (x_0, \text{evict}, \text{high}) \rangle$$
$$q_6 = \langle (x_0, \_, \_), (\_, \_, \text{high}), (x_0, \text{evict}, \text{low}) \rangle$$
$$q_7 = \langle (x_0, \_, \_), (\_, \_, \text{high}), (x_0, \text{evict}, \_), (\_, \text{schedule}, \_) \rangle$$
$$q_8 = \langle (x_0, \_, \_), (\_, \_, \text{high}), (x_0, \text{evict}, \_), (\_, \_, \text{high}) \rangle$$
$$q_9 = \langle (x_0, \_, \_), (\_, \_, \text{high}), (x_0, \text{evict}, \_), (\_, \_, \text{low}) \rangle$$

Example 4 - Matching

Let us consider stream S_3 and query q from Example 1:

Stream	Event	Job Id	Status	Priority
S_3	e_{31}	4	schedule	high
	e_{32}	5	schedule	low
	e_{33}	4	finish	high
	e_{34}	6	schedule	low
	e_{35}	5	evict	low

$$q = \langle (x_0, \_, \_), (\_, \_, \text{high}), (x_0, \text{evict}, \_) \rangle.$$

The parent query p and its entry in the stream matches dictionary T are:

$p = \langle (x_0, \_, \_), (\_, \_, \text{high}), (x_0, \_, \_) \rangle$ and $T(p) = \{S_3 \mapsto \{\langle 5 \rangle \mapsto 5\}\}$.

To match q, we replace the variables with the according value:

$$q_{rep} = \langle (5, \_, \_), (\_, \_, \text{high}), (5, \text{evict}, \_) \rangle$$

and find the matching information of its parent query p_{rep}:

$p_{rep} = \langle (5, \_, \_), (\_, \_, \text{high}), (5, \_, \_) \rangle$ and $T(p_{rep}) = \{S_3 \mapsto \{\langle\rangle \mapsto 5\}\}$

For p_{rep}, a match is found in stream S_3 at the position 5. When searching for a match for q_{rep}, we only need to consider the stream positions starting at 5, where, in this instance, a match for q_{rep} is indeed found.

$T(q)(S)(\langle a_1, \ldots a_k \rangle)$ captures the last position of the first match of query q with query variables $\langle x_1, \ldots, x_k \rangle$ in stream S, when each variable x_i is assigned the value a_i, $1 \leq i \leq k$. If q does not contain any variables, the sequence of bound attribute values is empty, i.e., $T(q)(S)(\langle\rangle)$ points to the last position of the first match of q in S.

We formalize our approach to query matching in Alg. 2. The algorithm iterates over each stream S in the database D (line 3). For each stream, the variables within query q are replaced by suitable attribute values using the information about matches of the parent query q_p (line 5), as kept in the stream matches dictionary T. If the replacement yields a type query (line 7), then it can be evaluated directly. Otherwise (line 13), all possible bindings for the remaining variable (there can only be one, as q is a child of q_p) are considered.

Algorithm 2: Match: Matching a Query

```
Input:  Query $q$, event schema $\mathcal{A}$, stream database $D$, parent query $q_p$,
        stream matches dictionary $T$.
Output: Match result $M \in \{True, False\}$, updated stream matches dict. $T$.
1  $U \leftarrow \emptyset$;
2  foreach $S \in D$ do $U(S) \leftarrow \emptyset$;
3  foreach $S \in D$ do
4      $N \leftarrow \emptyset$;          /* Set of matching positions of query $q$ */
       /* Consider each assignment of attribute values to
          variables as obtained for the parent query $q_p$        */
5      foreach $\langle a_1, \ldots, a_k \rangle \in \text{dom}(T(q_p)(S))$ do
           /* Replace variables in $q$ with attribute values    */
6          $q_{rep} \leftarrow \text{REPLACE}(q, \langle a_1, \ldots, a_k \rangle)$;
           /* If replacement yielded type query, match it     */
7          if $q_{rep}$ is a type query then
8              $T \leftarrow \text{MatchStream}(q_{rep}, S, \mathcal{A}, T)$;
9              if $T(q_{rep})(S)(\langle\rangle) \neq \circ$ then
10                 $N \leftarrow N \cup \{T(q_{rep})(S)(\langle\rangle)\}$;
11                 $U(S)(\langle a_1, \ldots, a_k \rangle) \leftarrow T(q_{rep})(S)(\langle\rangle)$;
12         else
               /* Consider all bindings for the one variable
                  that is still contained in $q_{rep}$                */
13             $A \leftarrow$ attribute $A \in \mathcal{A}$ of the variable in $q_{rep}$;
14             foreach $a \in \text{dom}(A)$ do
15                 $q_{rep} \leftarrow \text{REPLACE}(q_{rep}, \langle a \rangle)$;
16                 $T \leftarrow \text{MatchStream}(q_{rep}, S, \mathcal{A}, T)$;
17                 if $T(q_{rep})(S)(\langle\rangle) \neq \circ$ then
18                     $N \leftarrow N \cup \{T(q_{rep})(S)(\langle\rangle)\}$;
19                     $U(S)(\langle a_1, \ldots, a_k, a \rangle) \leftarrow T(q_{rep})(S)(\langle\rangle)$;
20     if $N = \emptyset$ then return $False, T$ ;
21 $T(q) \leftarrow \{U(S) \mid S \in D\}$;
22 return $True, T$;
   /* Function to match a type query against a single stream */
23 MatchStream (query $q$, stream $S$, event schema $\mathcal{A}$, stream match. dict. $T$)
   /* Construct a parent query, dropping the last query term */
24 if $|q| = 1$ then $p \leftarrow \langle \varepsilon \rangle$;
25 else $p \leftarrow \langle q[1], \ldots, q[|q| - 1] \rangle$;
26 $s \leftarrow \langle q[|q|] \rangle$;
27 if $S \notin \text{dom}(T(p))$ then $T \leftarrow \text{MatchStream}(p, S, \mathcal{A}, T)$;
   /* Get last matching position of parent query $p$ in $S$       */
28 $m_p \leftarrow T(p)(S)(\langle\rangle)$;
   /* Try to extend the match of the parent; capture the
      obtained match position $m$ or $\circ$ for no match              */
29 if $\exists\, i \in \{m_p + 1, \ldots, |S|\} : s \models S[i]$ then
30     $T(q)(S)(\langle\rangle) \leftarrow m_p + \text{argmin}_{i \in \{m_p+1,\ldots,|S|\}}\, s \models S[i]$;
31 else $T(q)(S)(\langle\rangle) \leftarrow \circ$;
32 return $T$
```

The evaluation of a type query for a specific stream is then formalized in function MatchStream (line 23-32). It constructs the parent type query by dropping the last query term, and checks for the last matching position in stream S as a starting point for matching (line 27, obtaining it recursively, if it is not available). That is, for query q, we check the remaining part of the stream for matches of the added part, i.e., the last query term of q (line 29).

4.3 Query Merging

In the DISCES framework, some algorithms separate the discovery of (i) type queries and pattern queries, and (ii) queries per attribute. In either case, queries need to be merged at a later stage. Here, we focus on the merging of type queries and pattern queries, while the algorithm to merge queries obtained for individual attributes can be found in the appendix of the accompanying technical report [26].

Algorithm 3: MERGEMIXEDQUERYSET: Query Merging

```
   Input:  Stream database $D$, event schema $\mathcal{A}$, type query set $Q_t$,
           pattern query set $Q_p$, parent dictionary $P$, stream matches dict. $T$.
   Output: Set of merged queries $Q$.
1  $Q \leftarrow \emptyset$; $O \leftarrow$ empty queue;
   /* Initialize the queue of merge options, incorporating the
      roots of the hierarchies of type/pattern queries        */
2  foreach $q_t \in (Q_t \setminus \text{dom}(P))$, $q_p \in (Q_p \setminus \text{dom}(P))$ do
3      foreach $q \in \{q' \in \text{dom}(P) \mid P(q') \in \{q_t, q_p\}\}$ do
4          $O$.ENQUEUE$(q_t, q_p, \langle\varepsilon\rangle, q)$;
5  while $O$ not empty do
6      $(q_t, q_p, q_m, q_e) \leftarrow O$.DEQUEUE$()$;
       /* Construct queries by merging $q_e$ into $q_m$        */
7      $C, P \leftarrow$ MERGEMIXEDQUERIES$(q_t, q_p, q_m, q_e, P)$;
8      for $q_n \in C$ do
9          $M, T \leftarrow$MATCH$(q_n, \mathcal{A}, D, P(q_n), T)$;
           /* For matching queries, derive new merge options */
10         if $M = True$ then
11             $Q \leftarrow \{q_n\} \cup Q \setminus \{q_m, q_e\}$;
12             if $q_e$ is a type query then
13                 foreach $q \in \{q' \in P \mid P(q') = q_n\}$ do
14                     $O$.ENQUEUE$(q_e, q_p, q_m, q)$;
15                 foreach $q \in \{q' \in P \mid P(q') = q_p\}$ do
16                     $O$.ENQUEUE$(q_e, q_p, q_m, q)$;
17             else
18                 foreach $q \in \{q' \in P \mid P(q') = q_n\}$ do
19                     $O$.ENQUEUE$(q_t, q_e, q_m, q)$;
20                 foreach $q \in \{q' \in P \mid P(q') = q_t\}$ do
21                     $O$.ENQUEUE$(q_t, q_e, q_m, q)$;
22 return $Q$;
```

Alg. 3 defines our approach to merge a set of type queries Q_t and a set of pattern queries Q_p. Our idea is to exploit the hierarchy among the type/pattern queries, and step-wise create mixed queries that become slightly more specific by incorporating exactly one additional attribute value or variable, respectively, in each step.

The algorithm maintains a queue O of merge options. Each element in O comprises a type query q_t, a pattern query q_p, a mixed query q_m constructed from q_t and q_p, and an evolved query q_e that is derived from q_t or q_p by adding one attribute value or variable, respectively. The queue is initialized with the root queries of the hierarchies (line 2-4), i.e., the type and pattern queries without a parent, while the evolved query corresponds to one of their children.

While the queue O is not empty, each tuple (q_t, q_p, q_m, q_e) is used to create more specific queries by merging q_e into q_m through function MERGEMIXEDQUERIES (line 7), described below. Then, each of the resulting queries $q_n \in C$ is matched against the stream database (line 9). If all streams support the query (line 10), the set of merged queries is adjusted (line 11) and further merge options are inserted into the queue (line 12-21), which realizes a traversal of the hierarchy of type queries and pattern queries.

The merging of the evolved query p_e into the mixed query p_m is formalized in Alg. 4, which also takes the associated type query q_t and pattern query q_p, and the parent dictionary P as input. It first determines an index i_m (line 4-8). It indicates until which query term the query q_m cannot change anymore without creating a conflict with queries q_t and q_p that form the basis for its construction. Here, the function MERGETERMS merges query terms (line 27-32), per attribute, returning the empty term (line 30), in case of a conflict.

Algorithm 4: MERGEMIXEDQUERIES: Merging of Queries

```
   Input:  Type query $q_t$, pattern query $q_p$, mixed query $q_m$, evolved query $q_e$,
           parent dictionary $P$.
   Output: Set of merged queries $Q$, updated parent dictionary $P$.
1  $i_t, i_p, i_m \leftarrow 0, 0, 0$;
2  $Q \leftarrow \emptyset$;
3  $j \leftarrow$ position of last inserted symbol in query $q_m$;
   /* Determine the index after which $q_m$ may still change    */
4  while $i_m < j$ do
5      if MERGETERMS $(q_t[i_t], q_m[i_m]) \neq \varepsilon$ then $i_t \leftarrow i_t + 1$;
6      if MERGETERMS $(q_p[i_p], q_m[i_m]) \neq \varepsilon$ then $i_p \leftarrow i_p + 1$;
7      $i_m \leftarrow i_m + 1$;
8      if MERGETERMS $(q_t[i_t], q_m[i_m]) = \varepsilon \wedge$
         MERGETERMS $(q_p[i_p], q_m[i_m]) = \varepsilon$ then break ;
9  if $q_e$ is a type query then
       /* Merge by incorporating an add. attribute value      */
10     for $k \in \{i_m, \ldots, |q_m|\}$ do
11         $q' \leftarrow \langle q_m[1], ..., q_m[k-1], q_t[|q_t|], q_m[k], ..., q_m[|q_m|]\rangle$;
12         $Q \leftarrow Q \cup \{q'\}$; $P(q') \leftarrow q_m$;
13     for $k \in \{i_m, \ldots, |q_m|\}$ do
14         $t \leftarrow$ MERGETERMS $(q_t[|q_t|], q_m[k])$;
15         if $t \neq \varepsilon$ then
16             $q' \leftarrow \langle q_m[1], ..., q_m[k-1], t, q_m[k+1], ..., q_m[|q_m|]\rangle$;
17             $Q \leftarrow Q \cup \{q'\}$; $P(q') \leftarrow q_m$;
18 else
       /* Merge by incorporating query terms of $q_e$             */
19     if $i_p = |q_p| \vee i_t = |q_t|$ then
20         $q' \leftarrow \langle q_m[1], ..., q_m[i_m], q_e[i_p], ..., q_e[|q_e|]\rangle$;
21         $Q \leftarrow Q \cup \{q'\}$; $P(q') \leftarrow q_m$;
22     $t \leftarrow$ MERGETERMS $(q_m[i_m], q_e[i_p])$;
23     if $t \neq \varepsilon$ then
24         $q' \leftarrow \langle q_m[1], ..., q_m[i_m - 1], t, q_e[i_p + 1], ..., q_e[|q_e|]\rangle$;
25         $Q \leftarrow Q \cup \{q'\}$; $P(q') \leftarrow q_m$;
26 return $Q, P$

   /* Function to merge two query terms                         */
27 MERGETERMS (query term $t_1$, query term $t_2$, event schema $\mathcal{A}$)
28 $t_m \leftarrow t_1$;
29 foreach $A \in \mathcal{A}$ do
30     if $t_1.A \neq t_2.A \wedge t_1.A \neq \_ \wedge t_2.A \neq \_$ then return $\varepsilon$;
31     if $t_1.A \neq t_2.A$ then $t_m.A = t_2.A$;
32 return $t_m$
```

Based on index i_m, mixed queries are constructed by incorporating an additional attribute value (line 9-17), either in one of the existing query terms or as a new query term, for all possible positions. Subsequently, if the evolved query is not a type query, further mixed queries are constructed (line 18-25), by replacing the query terms after i_m of the original query with those of the evolved query, potentially merging the last term of the original query and the first of the evolved query. All these mixed queries are collected in the result set Q, while maintaining the parent dictionary P for the queries.

4.4 Realizations of Query Discovery

Using the above auxiliary algorithms, we can now instantiate four discovery algorithms, as outlined already in Table 3. Below, we provide formalizations for two of the algorithms, D-U-C and B-S-C, and summarize the other two algorithms (for which a formalization can be found in [26]). We close with a discussion of their properties.

D-U-C: DFS, pattern-type unified, attribute comprehensive. The algorithm, given in Alg. 5, employs a DFS strategy, constructs mixed queries directly, and incorporates all attributes right away.

Algorithm 5: D-U-C

```
Input:  Stream database $D$, event schema $\mathcal{A}$, sequence of query var. $\mathcal{X}^<$,
        Boolean flag $b \in \{True, False\}$ whether to output only desc. queries.
Output: Set of descriptive queries $Q$, updated parent dictionary $P$.
1  $Q \leftarrow \emptyset$;
2  $\Gamma_D \leftarrow \{a \in \Gamma \mid \forall\, S \in D : \exists\, j \in \{1,...,|S|\}, i \in \{1,...,n\} : S[j].A_i = a\}$;
3  $q \leftarrow \langle\varepsilon\rangle$; $P(q) \leftarrow \langle\varepsilon\rangle$;
4  foreach $S \in D$ do $T(q)(S)(\langle\rangle) \leftarrow 0$ ;
   /* Explore queries that are pushed to the stack */
5  $O \leftarrow$ empty stack; $O$.PUSH$(q)$;
6  while $O$ not empty do
7      $q \leftarrow O$.POP$()$;
8      $M, T \leftarrow$ MATCH$(q, \mathcal{A}, D, P(q), T)$ ;
9      if $M = True$ then
           /* For queries supported by the stream database,
              explore their children */
10         $P \leftarrow$ CHILDQUERIES$(q, \mathcal{A}, \Gamma_D, P, \mathcal{X}^<, True)$;
11         $C \leftarrow \{q' \in \mathrm{dom}(P) \mid P(q') = q\}$;
12         if $C \neq \emptyset \wedge b = True$ then
13             foreach $q' \in C$ do $O$.PUSH$(q')$;
14         else $Q \leftarrow Q \cup \{q\}$;
15     else if $b = False$ then $Q \leftarrow Q \cup \{P(q)\}$;
16 if $b = True$ then $Q \leftarrow$ DESCRIPTIVEQUERIES$(Q, \Gamma_D, \mathcal{A}, \mathcal{X}^<)$ ;
17 return $Q, P$
```

For a stream database D, an event schema $\mathcal{A}$, and a sequence of query variables $\mathcal{X}^<$ (see §4.1), it returns a set of descriptive queries. It starts with the empty query $\langle\varepsilon\rangle$, which is pushed to a stack of queries to explore. While this stack is not empty, candidate queries are assessed and generated (line 6-15). That is, a query is popped from the stack and matched. Note that the empty query will match any stream. Following a successful match, child queries are generated using function CHILDQUERIES and extracted from the parent dictionary P into a set C. If child queries exist, they are added to the stack; otherwise, the current query is added to the preliminary result set. In a post-processing step, a function DESCRIPTIVEQUERIES narrows down the set of queries supported by the database to those that are descriptive, which is achieved by a syntactic comparison of query terms (formalized in our technical report [26]).

B-S-C: BFS, pattern-type separated, attribute comprehensive. In Alg. 6, we show how discovery is realized with BFS, considering type queries and pattern queries separately, while incorporating all attributes at once. In general, the algorithm follows a similar structure compared to the previous one. However, the exploration is based on candidate queries that are maintained in queues to realize a BFS strategy. Also, the assessment of queries and, if they are supported by the stream database, subsequent generation of child queries is conducted separately for type queries (line 6-14) and pattern type queries (line 16-24). Only once this exploration ended, mixed queries are considered based on the function MERGEMIXEDQUERYSET.

D-U-S: DFS, pattern-type unified, attribute separated. This algorithm is a variant of the D-U-C algorithm. However, it first discovers queries separately per attribute, before merging them to obtain the final result. That is, the algorithm runs the D-U-C algorithm (Alg. 5) per attribute, i.e., on a projection of the streams and of the event schema on one attribute. However, setting flag b to *False*, all queries supported by the database are collected, not only those that are descriptive. Finally, queries discovered per attribute are merged.

B-S-S: BFS, pattern-type separated, attribute separated. This is a variant of B-S-C, adopting BFS and the separated discovery of type queries and pattern queries. Yet, it first considers queries per attribute, before merging them. As such, it adopts the strategy explained above for D-U-S, just with B-S-C as the base algorithm.

Algorithm 6: B-S-C

```
Input:  Stream database $D$, event schema $\mathcal{A}$, sequence of query var. $\mathcal{X}^<$,
        Boolean flag $b \in \{True, False\}$ whether to output only desc. queries.
Output: Set of descriptive queries $Q$, updated parent dictionary $P$.
1  $Q \leftarrow \emptyset$;
2  $\Gamma_D \leftarrow \{a \in \Gamma \mid \forall\, S \in D : \exists\, j \in \{1,...,|S|\}, i \in \{1,...,n\} : S[j].A_i = a\}$;
3  $q \leftarrow \langle\varepsilon\rangle$; $P(q) \leftarrow \langle\varepsilon\rangle$;
4  foreach $S \in D$ do $T(q)(S)(\langle\rangle) \leftarrow 0$;
   /* Explore type queries based on a queue */
5  $O_{type} \leftarrow$ empty queue; $O_{type}$.ENQUEUE$(q)$;
6  while $O_{type}$ not empty do
7      $q \leftarrow O_{type}$.DEQUEUE$()$;
8      $M, T \leftarrow$ MATCH$(q, \mathcal{A}, D, P(q), T)$;
9      if $M = True$ then
10         $Q_{type} \leftarrow Q_{type} \cup \{q\}$;
11         $P \leftarrow$ CHILDQUERIES$(q, \mathcal{A}, \Gamma_D, P, \mathcal{X}^<, False)$;
12         $C \leftarrow \{q' \in \mathrm{dom}(P) \mid P(q') = q\}$;
13         if $C \neq \emptyset$ then
14             foreach $q' \in C$ do $O_{type}$.ENQUEUE$(q')$;
   /* Explore pattern queries based on a queue */
15 $O_{pattern} \leftarrow$ empty queue; $O_{pattern}$.ENQUEUE$(q)$;
16 while $O_{pattern}$ not empty do
17     $q \leftarrow O_{pattern}$.DEQUEUE$()$;
18     $M, T \leftarrow$ MATCH$(q, \mathcal{A}, D, P(q), T)$;
19     if $M = True$ then
20         $Q_{pattern} \leftarrow Q_{pattern} \cup \{q\}$;
21         $P \leftarrow$ CHILDQUERIES$(q, \mathcal{A}, \emptyset, P, \mathcal{X}^<, True)$;
22         $C \leftarrow \{q' \in \mathrm{dom}(P) \mid P(q') = q\}$;
23         if $C \neq \emptyset$ then
24             foreach $q' \in C$ do $O_{pattern}$.ENQUEUE$(q')$;
   /* Merge the found type queries and pattern queries */
25 $Q, P \leftarrow$ MERGEMIXEDQUERYSET$(D, \mathcal{A}, Q_{type}, Q_{pattern}, P, T)$;
26 if $b = True$ then $Q \leftarrow$ DESCRIPTIVEQUERIES$(Q, \Gamma_D, \mathcal{A}, \mathcal{X}^<)$;
27 return $Q, P$
```

Correctness, descriptiveness, completeness. The four presented algorithms solve the problem of event query discovery (Problem 1). Given a stream database D, all four algorithms, D-U-C, B-S-C, D-U-S, and B-S-S, construct a set of queries Q that is correct, descriptive, and complete, which is established as follows:

Correctness: Algorithm MATCH returns the match result *True*, only if for each stream, at least one match is found for some binding of attribute values to the variables. Here, MATCHSTREAM exploits that the index of a match of a type query in a stream can only be larger or equal to the minimal index of the matches of a query obtained by removing the last query term. Now, consider the individual discovery algorithms: D-U-C (with flag $b = True$) collects in Q only matching queries. Similarly, B-S-C enqueues in Q_{type} and $Q_{pattern}$ only matching queries, while the queries obtained by merging these sets in MERGEMIXEDQUERYSET are checked by MATCH to be part of the result set. For D-U-S and B-S-S, correctness follows from D-U-C and B-S-C returning only matching queries, and MERGEATTRIBUTEQUERIES only adding matching merged queries to Q_m.

Descriptiveness: The result set Q contains only descriptive queries, as all four algorithms filter non-descriptive queries in a post-processing step (function DESCRIPTIVEQUERIES in Alg. 5 and Alg. 6).

Completeness: First, CHILDQUERIES may inductively generate any query according to our model starting with the empty query for a given schema and sequence of query variables (the number of variables is bound by half of the length of a shortest stream). That is, any placeholder of an existing query term may be replaced by any attribute value of the respective domain (supported by all streams) or a variable, and a new query term may also be inserted for any of these values or variables. For D-U-C, completeness then follows from the fact that any possible child of a matching query, derived by inserting one additional attribute value or variable, is explored. For B-S-C, in addition to the generation of all child queries of matching type/pattern queries, MergeMixedQuerySet and MergeMixedQueries generate any possible interleaving of query terms, as well as elements of query terms from these queries. For D-U-S and B-S-S, our argument rests further on the completeness of the merging of queries per attribute, which unfolds all combinations.

Runtime complexity. We list the worst-case time complexity of our individual algorithms in Table 4 (with MergeMixedQuerySet being dominated by Match). Based thereon, we conclude that the presented algorithms do not differ in their overall runtime complexity. In addition to the large search space, the source of computational complexity is the exponential running time of our algorithm for solving the matching problem. This, however, cannot be avoided, since the matching problem is NP-hard in general [15]. For a more efficient implementation of the matching problem, it would be desirable to avoid the term $|\Gamma|^{|\mathcal{X}|}$, e.g., by obtaining a running time of $O(\text{poly}(|D|, |\Gamma|) \cdot 2^{|\mathcal{X}|})$ (note that for constant queries, this would be a polynomial running time). Unfortunately, due to the $W[1]$-hardness of the matching problem parameterized by $|\mathcal{X}|$ [15], such running times are excluded as well (under common complexity theoretical assumptions). These complexity bounds justify our approach where, instead of trying to optimize an algorithm for the matching problem, we exploit the fact that the matching problem is solved repeatedly, for instances that are structurally similar (see §4.2).

5 GUIDANCE ON QUERY DISCOVERY

To guide the use of our algorithms in application scenarios, we now discuss how to choose among the algorithms for a given stream database (§5.1), before outlining how to give feedback if their application turns out to be intractable or ineffective (§5.2).

5.1 Algorithm Selection

Our idea is to guide the selection of a discovery algorithm based on characteristic properties of a stream database. The properties may hint at which of the algorithms can be expected to be particularly efficient, as it caters for the structure of the database. To realize this idea, we may leverage database properties that (i) can be computed efficiently (i.e., in linear time in the size of the streams), and (ii) enable us to separate the performance characteristics of the algorithms.

In general, one may assume that the size of the stream database D in terms of the number of streams, $|D|$, and their lengths, e.g., given by the minimum length $\min_{S \in D} |S|$, are important properties for event query discovery. Yet, based on our complexity analysis, we observe a minor influence of these properties on the runtime performance of discovery algorithms. The reason being that the size of a database does not characterize the size of the space of candidate queries that needs to be explored by the discovery algorithms. We therefore consider the following four properties:

Table 4: Worst-case complexity of algorithms.

Function	Complexity
ChildQueries	$\mathcal{O}\left(\lvert q\rvert^2 \cdot n^2 + \lvert q\rvert \cdot \lvert\Gamma_D\rvert \cdot n\right)$
Match	$\mathcal{O}\left(\lvert D\rvert \cdot \lvert\Gamma\rvert^{\lvert\mathcal{X}\rvert}\right)$
MergeMixedQueries	$\mathcal{O}\left(\lvert q_t\rvert \cdot \lvert q_p\rvert \cdot n\right)$
MergeAttributeQueries (see [26])	$\mathcal{O}\left(\max_{i \in \{1,\dots,n\}} (\lvert q_i\rvert)^n\right)$
DescriptiveQueries (see [26])	$\mathcal{O}\left(\lvert q\rvert \cdot n\right)$ per $q \in Q$

By $|q|$, we denote the length of the query q, i.e., the number of query terms. By n, we denote the number of attributes in the event schema $\mathcal{A}$.

Number of attributes $|\mathcal{A}|$. The property captures the number of attributes in the event schema. It can be expected to indicate whether the separate or unified handling of type queries and pattern queries is more beneficial. The overhead induced by the merging of separately discovered queries can be expected to increase significantly with the number of attributes. Hence, a higher value of $|\mathcal{A}|$ shall render D-U-S and D-U-C more efficient compared to B-S-S and B-S-C.

Number of supported attribute values $|\Gamma_D|$. We consider the size of the supported alphabet, i.e, the number of supported attribute values that appear in the database. With a large alphabet, the separated discovery of type queries and pattern queries can be expected to be beneficial, i.e., B-S-S and B-S-C shall be most efficient. The reason being that we expect many pattern queries to be present, which, under a unified construction, would always be extended with a large number of attribute values during the exploration.

Max of sum of supported attribute values ρ_S. The property is the maximum number over all streams and all attributes, of the summed up occurrences of all supported attribute values:

$$\rho_S = \max_{\langle e_1,\dots,e_l\rangle \in D} \left(\sum_{i=1}^{l} \sum_{A \in \mathcal{A}} \mathbb{1}(e_i, A) \right) \text{ with } \mathbb{1}(e_i, A) = \begin{cases} 1, \text{if } e_i.A \in \Gamma_D \\ 0, \text{otherwise} \end{cases} .$$

If there is a large number of supported values for an attribute, approaches that handle attributes separately need to realize numerous merge operations for single-attribute queries. This overhead is avoided in D-U-C and B-S-C, which can be expected to run more efficiently than D-U-S and B-S-S, for higher values of this measure.

Min of sum of repeated attribute values ρ_R. The property captures the minimum number over all streams of the summed up repetitions of all attribute values within a stream S:

$$\rho_R = \min_{\langle e_1,\dots,e_l\rangle \in D} \left(\sum_{i=1}^{l} \sum_{A \in \mathcal{A}} \mathbb{1}(e_i, A) \right) \text{ with}$$

$$\mathbb{1}(e_i, A) = \begin{cases} 1, \text{if } \exists\ j \in \{1,\dots,l\}, i \neq j : e_i.A = e_j.A \in \Gamma \\ 0, \text{otherwise} \end{cases} .$$

Intuitively, the repetition of attribute values in a stream increases the size of the search space for pattern queries. As the repetitions may generally be spread over several attributes, approaches that separate discovery per attribute, i.e., D-U-S and B-S-S, can be expected to be more efficient than those with comprehensive handling of attributes.

The above properties can be expected to be correlated with differences in the runtime performance of the discovery algorithms. However, to guide the selection of an algorithm for a given stream database, absolute thresholds need to be identified to formulate decision criteria. We later explore this aspect empirically, using a controlled experimental setup.

5.2 Feedback based on Algorithmic Performance

In practice, a stream database is typically not fixed, but subject to transformations as part of the extraction and preparation of the data. Various abstractions, such as event selection, projection of attributes, or discretization of attribute values, are commonly employed. Therefore, once a discovery algorithm does not terminate within reasonable time or yields an empty result set, feedback may be derived on how these abstractions influence the algorithmic performance.

Abstractions to reduce runtime. If event query discovery turns out to be intractable, the space of candidate queries may be reduced. Specifically, an attribute may be removed entirely or highly frequent values of an attribute may be made unique. In either case, the respective information will be ignored in the discovery procedure, which, assuming it is justified in the considered application scenario, can be expected to reduce the overall runtime of the discovery algorithms. Practically, the attributes with the smallest domains are suitable candidates for removal and a specialization of their attribute values.

Abstractions to increase the result size. In case no or only a very few descriptive queries are discovered, abstractions of the respective data may be adopted to obtain a larger result set. In particular, attribute values may be filtered entirely or generalized through clustering, e.g., as part of a discretization of continuous attributes. Then, the domain of the attribute becomes smaller and the number of events carrying the same attribute values grows. Hence, it is more likely that regularities in the streams can be identified through the discovery of queries. The attributes with the largest domains are suitable candidates for the filtering and generalization of their values.

6 EXPERIMENTAL EVALUATION

To evaluate the DISCES framework, we first introduce an experimental setup (§6.1). Then, we compare our algorithms against the state of the art for event query discovery (§6.2). Finally, we evaluate the sensitivity of our algorithms with respect to the characteristics of a stream database (§6.3), before studying the effectiveness of our strategies to guide the application of discovery algorithms (§6.4).

6.1 Experimental Setup

Datasets. We leverage both real-world and synthetic data to evaluate our algorithms. Specifically, we employ streams of NASDAQ stock trade events [10] and the Google cluster traces [25]. The methodology for obtaining stream databases is adapted from [13]. For each dataset, we define three queries to generate streams, as follows.

The event schema for the finance data comprises three attributes: stock name, volume, and closing value. The queries to construct streams capture two events with stock name 'GOOG' (F1); three events with stock names appearing in the sequence 'AAPL', 'GOOG', and 'MSFT' (F2); and four events for which the volume remains constant (F3). Events in the Google cluster traces contain four attributes: job ID, machine ID, task status, and priority. One query defines three events running on the same machine with identical job IDs, with the first and last ones of status 'submit', while the second one has status 'kill' (G1); one query encompasses four events executing on the same machine with status 'finish' (G2); and one query detects job eviction resulting from the scheduling of another job on the same machine (G3).

Table 5: Stream databases used in the experiments.

		stream length a ǀ b	number of streams	maximum query length
	F1	50 ǀ 51	79	4
Finance	F2	40 ǀ 41	1000	4
	F3	25 ǀ 26	250	5
Google	G1-G3	7 ǀ 8	1000	4

Table 6: Properties of the synthetic stream databases.

	$\lvert D\rvert$	$\lvert S\rvert$	$\lvert\mathcal{A}\rvert$	$\lvert\Gamma_D\rvert$	ρ_S	ρ_R
E_1	**2,102,...,902**	10	3	0	0	2
E_2	2	**6,16,...,96**	3	0	0	6
E_3	2	20	**1,6,...,101**	2	5	5
E_4	2	102	1	**1,11,...,91**	$\lvert\Gamma_D\rvert$	0
E_5	2	20	3	3	**9,24,...,294**	6
E_6	2	20	3	0	0	**2,3,...,13**

Using these queries, we generated the stream databases summarized in Table 5 by evaluating the queries over the datasets. For each match (of the first 1000 matches), we constructed a stream by including the a or $b = a + 1$ events preceding the match. The values of a and b, listed in Table 5, have been determined experimentally using the IL-Miner [13] (described below), i.e., the state of the art for event query discovery: For a database with streams of length a, the IL-Miner was still able to compute results, whereas for streams of length b, it failed to compute results within 12 hours. As such, the databases characterize the computational limit of the state of the art, thereby providing a suitable basis for the comparison.

Moreover, we employed synthetic data for controlled experiments, which assess the impact of specific properties of a stream database. For each evaluation scenario, the dataset was generated by first creating a database of two streams that do not support any query (i.e., there are neither supported nor repeated attribute values). Here, the number of events per stream and the number of attributes may vary for each scenario. Then, the database is adapted by adding or replacing attribute values, as listed in Table 6, such that one database property increases while the others remain constant. The only exception is that a higher number of supported attribute values $|\Gamma_D|$ also increases the max of sum of supported attribute values ρ_S.

Baselines. We compare the algorithms of the DISCES framework against the IL-Miner [13], the only existing algorithm to discover event queries with repeated attribute values (see §7). Since the IL-Miner, by default, cannot discover queries that comprise query terms that contain only variables, we consider two variants: one extended version that can discover these queries, thereby covering the full query model of the DISCES framework, and one lossy version that neglects queries with terms containing no attribute values.

To add another perspective, we include a baseline using a representation learning algorithm for event streams, as introduced in [19]. The approach learns a probabilistic automaton to capture events and dependencies that indicate a situation of interest. While the learned automaton is used for stream monitoring in [19], a query can be extracted from it by enumerating certain transition paths [20]. While this approach yields only type queries, no pattern queries or mixed queries, a comparison sheds light on the potential of learning-based techniques for event query discovery.

Measures. We primarily measure the efficiency of event query discovery in terms of the algorithms' runtime. We report averages over five experimental runs, including error bars (mostly negligible).

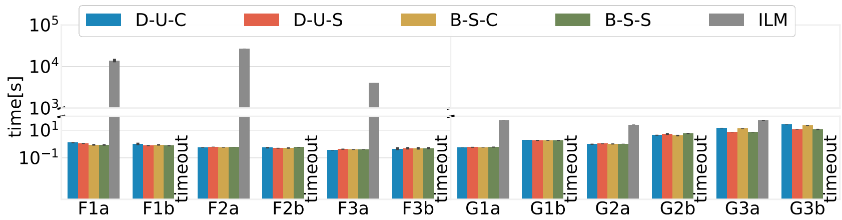


Figure 1: Runtime comparison with the state of the art.

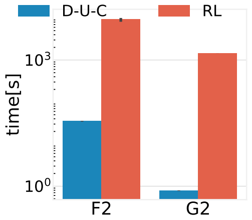


Figure 2: Runtime, RL approach.

Table 7: Comparison of our algorithms and the lossy IL-Miner.

Datasets	Avg $^{\text{Time Lossy ILM}}/_{\text{Time DISCES}}$	Desc(riptive)	Found	¬ Desc
F1b-F3b	0.42	8	0	4
G1b-G3b	2.9	19	1	2

Table 8: Comparison of our algorithms and the RL approach.

Dataset	Desc(riptive)	Found	¬ Desc	¬ Matching
F2b	1	0	1	4
G2b	1	0	1	3

Environment. We implemented the DISCES framework in Python. Due to performance issues of the Java-based version of the IL-Miner and the absence of a public implementation of the representation learning approach, we relied on Python-based implementations of the baseline algorithms. All experiments were conducted on a server with a Xeon 6354 CPU @3,6GHz and 1TB of RAM.

6.2 State-of-the-Art Comparison

First, we compare the runtimes of our algorithms and the extended IL-Miner, which supports our full query model. The results in Fig. 1 show that our algorithms are faster than the IL-Miner in all cases. For the scenarios based on the finance dataset, our algorithms are about five orders of magnitude faster. As discussed, we determined the stream lengths for the *a* and *b* scenarios as the computational limit of the state of the art, i.e., in all *b* scenarios, the IL-Miner cannot produce results within 12 hours. The algorithms of the DISCES framework enable us to get past this limit in all cases, without a notable increase in the runtime.

A comparison with the lossy IL-Miner reveals that its improved runtime is traded for result correctness. The lossy IL-Miner yields similar runtimes compared to our algorithms, see Table 7, yet it discovers only a fraction of the queries. That is, with the correct results containing 8 (finance) and 19 (Google) descriptive queries ('Desc'), the lossy IL-Miner discovers none or only one of them ('Found'), and additionally two and four queries that are not descriptive ('¬ Desc'). As such, it largely compromises the result quality not only in terms of descriptiveness, but also in terms of completeness.

Finally, we turn to the comparison with the approach based on representation learning. Since the latter can only discover type queries, we adapted one of our algorithms (D-U-C) to also return only these queries. For the datasets F2b and G2b, Fig. 2 shows the obtained runtime results, illustrating that our algorithm is at least two orders of magnitude faster than the baseline algorithm. The effectiveness of query discovery is summarized in Table 8. None of the descriptive queries is discovered by the baseline ('Found'), while only one matching, but not descriptive query is returned for either scenario ('¬ Desc'). Also, the construction based on representation learning yields several queries that are not supported by the stream database ('¬ Matching'). Hence, the learning-based approach compromises correctness, descriptiveness, and completeness and, due to high runtimes, is not suitable to address the problem of event query discovery.

6.3 Sensitivity Analysis

Our algorithms show the same worst-case complexity (dominated by the need to match query candidates), but incorporate different design choices. We explore the impact of these choices regarding properties of stream databases (in particular those introduced in §5.1) in a sensitivity analysis. For the systematically generated stream databases from Table 6, the results are shown in Fig. 3.

Varying the number of streams (E1) leads to generally higher runtimes. Yet, since the size of the database influences the algorithm to match query candidates, all of our algorithms are effected in the same manner. An increased length of the streams (E2), in turn, does not affect the discovery efficiency. As expected (see §5.1), the property cannot be used to guide algorithm selection, since it does not influence the size of the space of candidate queries to explore.

For the four properties introduced in §5.1 for algorithm selection, the results confirm our hypotheses. A large number of attributes (E3) favours the algorithms that avoid merging of type queries and pattern queries, i.e., D-U-S and D-U-C are more efficient than B-S-S and B-S-C. Increasing the set of supported attribute values (E4) has the converse effect, favouring the separate discovery of type queries and pattern queries. The properties related to the distribution of supported attribute values (E5) and repeated attribute values (E6) enable us to separate the approaches that handle attributes separately or with a unified approach. As such, our results underline that the design choices captured in the DISCES framework indeed effect the discovery efficiency, so that depending on the properties of a stream database, any of the four algorithms may be most efficient.

6.4 Guidance of Query Discovery

Algorithm selection. Next, we assess whether the runtime performance of our algorithms is indeed correlated with database properties as observed for the real-world data. From Table 9, we conclude that the differences in the number of attributes ($|\mathcal{A}|$) and supported attribute values ($|\Gamma_D|$) are too small to enable any differentiation, while the results for the repeated attribute values (ρ_R) relate to an inconclusive value range (see Fig. 3). Interestingly, the measure based on the supported attribute values (ρ_S) provides clues on the runtime performance. For $\rho_S = 5$, the algorithms handling attributes separately (D-U-S and B-S-S) perform much better than those handling them comprehensively. For $\rho_S > 10$, the runtimes of our algorithms show the opposite trend. Hence, the considered database properties,

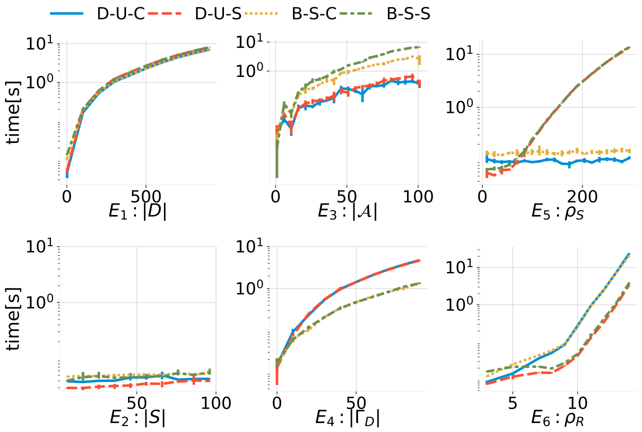


Figure 3: Sensitivity analysis using synthetic data.

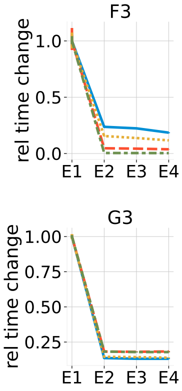
Table 9: Correlation of runtime and database properties.

Algorithm Runtime				Database Properties			
D-U-C	D-U-S	B-S-C	B-S-S	ρ_R	ρ_S	$\|\mathcal{A}\|$	$\|\Gamma_D\|$
18.28	26.43	17.65	26.21	6	11	3	2
13.50	14.95	10.52	13.34	5	17	3	2
0.65	1.39	0.88	1.68	9	15	3	6
259.11	364.07	226.32	366.51	10	12	4	1
55.43	83.84	49.32	84.51	8	10	4	1
533.90	172.54	488.45	172.37	8	5	4	1

assuming that their absolute values provide sufficiently strong indicators, indeed enable conclusions on the suitability of our algorithms.

Feedback mechanisms. Finally, we tested our mechanisms for feedback on the algorithmic performance. First, we assessed whether the exclusion of attribute values may indeed serve as an abstraction to reduce the discovery runtime. Fig. 4 demonstrates the relative change in the runtimes of our algorithms, when step-wise realizing such an exclusion. Here, E1 refers to the original database, while for E2-E4, we step-wise removed the most frequent attribute values of the attribute of the smallest domain ('volume' for finance, 'status' for Google). As expected, the runtime decreases. For both datasets, the attribute value distributions are highly skewed, so that the exclusion of the most frequent value (E2) has the largest effect.

Second, we study the abstraction of clustering of attribute values. C1 denotes the original dataset without clustering. Then, for the Google dataset, we clustered the 'priority' attribute, as adopted in [25] (C2) and [24] (C3). For the finance dataset, we clustered the 'close price' into 100 (C2) or 50 (C3) equally-sized clusters. Fig. 5 highlights that higher numbers of values within a cluster increase the runtimes. For the finance dataset, the effect is generally larger and especially affects the algorithms that handle attributes comprehensively. However, we motivated the abstraction with the desire to increase the result size. For the finance dataset, we obtain 18 (C1), 30 (C2), and 106 (C3) descriptive queries, while for the Google dataset, the result includes 202 (C1), 153 (C2), and 298 (C3) queries. We conclude that the expected trend materializes for our data, even though there is no monotonic relation. The latter is explained by the fact that clustering may also lead to similar descriptive queries being merged into one, which may dominate the effect of having more descriptive queries due to smaller attribute domains.



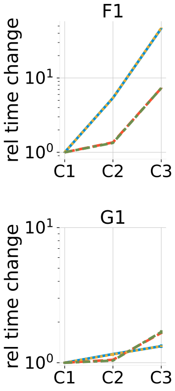


Figure 4: Attr. exclusion. **Figure 5: Attr. clustering.**

7 RELATED WORK

Closest to our work are iCEP [21] and the IL-Miner [13], which both address the problem of event query discovery. Yet, both algorithms take ad-hoc design choices and are limited in the considered query model. iCEP cannot discover queries, in which attribute values occur multiple times. This limitation is overcome by the IL-Miner, which, however, cannot discover queries with query terms that contain only variables. We showed empirically that this lossy version of the IL-Miner suffers from incomplete results. Also, our algorithms outperform an extended version of the IL-Miner and, in particular, enlarge the set of problem instances that may be addressed.

Our query model and the notion of descriptiveness is inspired by [15, 16]. Yet, the discovery algorithms proposed in [15, 16] are limited to finding *some* descriptive queries, not all of them.

Machine learning approaches to anticipate situations of interest provide an alternative angle to stream-based monitoring [19, 22, 28]. The main drawback of these methods, however, is the lack of traceability of the derived predictions. Using an approach to construct probabilistic state machines based on representation learning [19] in our evaluation, we observed correctness and completeness issues.

Event query discovery is linked to frequent sequence mining [1, 30, 31], which considers solely the level of attribute values, though. Our query model embeds a finite sequence into another sequence that satisfies certain constraints. This problem setting is ubiquitous in foundational algorithmic research, especially in combinatorial string matching [8, 18]. Pattern discovery has also been investigated for time-series data [23, 29], which, again, works only on attribute values and ignores criteria to correlate events by means of variables.

8 CONCLUSIONS

In this paper, we systematically explored the design space for algorithms that discover event queries for a stream database. We captured the design choices in the DISCES framework and instantiated it to derive four discovery algorithms, which all provide correct and complete results. Our experiments highlight that our algorithms outperform the state of the art in event query discovery, significantly extending the size of the problem instances that are still tractable. Moreover, we studied the influence of database properties on the algorithms' efficiency and the discovery result, thereby providing an angle to improve their applicability. In future work, we aim to develop methods for dynamic combination of our algorithms.

REFERENCES

[1] Rakesh Agrawal and Ramakrishnan Srikant. 1995. Mining sequential patterns. In *Proceedings of the eleventh international conference on data engineering*. IEEE, 3–14.

[2] Arvind Arasu, Shivnath Babu, and Jennifer Widom. 2006. The CQL continuous query language: semantic foundations and query execution. *VLDB J.* 15, 2 (2006), 121–142. https://doi.org/10.1007/s00778-004-0147-z

[3] Alexander Artikis, Alessandro Margara, Martín Ugarte, Stijn Vansummeren, and Matthias Weidlich. 2017. Complex Event Recognition Languages: Tutorial. In *Proceedings of the 11th ACM International Conference on Distributed and Event-based Systems, DEBS 2017, Barcelona, Spain, June 19-23, 2017*. ACM, 7–10. https://doi.org/10.1145/3093742.3095106

[4] Alexander Artikis, Matthias Weidlich, François Schnitzler, Ioannis Boutsis, Thomas Liebig, Nico Piatkowski, Christian Bockermann, Katharina Morik, Vana Kalogeraki, Jakub Marecek, Avigdor Gal, Shie Mannor, Dimitrios Gunopulos, and Dermot Kinane. 2014. Heterogeneous Stream Processing and Crowdsourcing for Urban Traffic Management. In *Proceedings of the 17th International Conference on Extending Database Technology, EDBT 2014, Athens, Greece, March 24-28, 2014*, Sihem Amer-Yahia, Vassilis Christophides, Anastasios Kementsietsidis, Minos N. Garofalakis, Stratos Idreos, and Vincent Leroy (Eds.). OpenProceedings.org, 712–723. https://doi.org/10.5441/002/edbt.2014.77

[5] Kaustubh Beedkar, Klaus Berberich, Rainer Gemulla, and Iris Miliaraki. 2015. Closing the Gap: Sequence Mining at Scale. *ACM Trans. Database Syst.* 40, 2, Article 8 (jun 2015), 44 pages. https://doi.org/10.1145/2757217

[6] Badrish Chandramouli, Mohamed H. Ali, Jonathan Goldstein, Beysim Sezgin, and Balan Sethu Raman. 2010. Data Stream Management Systems for Computational Finance. *Computer* 43, 12 (2010), 45–52. https://doi.org/10.1109/MC.2010.346

[7] Gianpaolo Cugola and Alessandro Margara. 2012. Processing flows of information: From data stream to complex event processing. *ACM Comput. Surv.* 44, 3 (2012), 15:1–15:62. https://doi.org/10.1145/2187671.2187677

[8] Joel D. Day, Maria Kosche, Florin Manea, and Markus L. Schmid. 2022. Subsequences with Gap Constraints: Complexity Bounds for Matching and Analysis Problems. In *33rd International Symposium on Algorithms and Computation, ISAAC 2022, December 19-21, 2022, Seoul, Korea*. 64:1–64:18. https://doi.org/10.4230/LIPICS.ISAAC.2022.64

[9] Yagil Engel, Opher Etzion, and Zohar Feldman. 2012. A basic model for proactive event-driven computing. In *Proceedings of the Sixth ACM International Conference on Distributed Event-Based Systems, DEBS 2012, Berlin, Germany, July 16-20, 2012*, François Bry, Adrian Paschke, Patrick Th. Eugster, Christof Fetzer, and Andreas Behrend (Eds.). ACM, 107–118. https://doi.org/10.1145/2335484.2335496

[10] EODData. 2015. NASDAQ Intra-Day Data. https://eoddata.com/. Accessed: 2015-09-27.

[11] Philippe Fournier-Viger, Cheng-Wei Wu, Antonio Gomariz, and Vincent S. Tseng. 2014. VMSP: Efficient Vertical Mining of Maximal Sequential Patterns. In *Advances in Artificial Intelligence*, Marina Sokolova and Peter van Beek (Eds.). Springer International Publishing, Cham, 83–94.

[12] Philippe Fournier-Viger, Cheng-Wei Wu, and Vincent S. Tseng. 2013. Mining Maximal Sequential Patterns without Candidate Maintenance. In *Advanced Data Mining and Applications*, Hiroshi Motoda, Zhaohui Wu, Longbing Cao, Osmar Zaiane, Min Yao, and Wei Wang (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 169–180.

[13] Lars George, Bruno Cadonna, and Matthias Weidlich. 2016. IL-Miner: Instance-Level Discovery of Complex Event Patterns. *Proc. VLDB Endow.* 10, 1 (Sept. 2016), 25–36. https://doi.org/10.14778/3015270.3015273

[14] 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. https://doi.org/10.1007/s00778-019-00557-w

[15] 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 *25th International Conference on Database Theory, ICDT 2022, March 29 to April 1, 2022, Edinburgh, UK (Virtual Conference) (LIPIcs)*, Dan Olteanu and Nils Vortmeier (Eds.), Vol. 220. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 18:1–18:21. https://doi.org/10.4230/LIPIcs.ICDT.2022.18

[16] Sarah Kleest-Meißner, Rebecca Sattler, Markus L. Schmid, Nicole Schweikardt, and Matthias Weidlich. 2023. Discovering Multi-Dimensional Subsequence Queries from Traces - From Theory to Practice. In *Datenbanksysteme für Business, Technologie und Web (BTW 2023), 20. Fachtagung des GI-Fachbereichs „Datenbanken und Informationssysteme" (DBIS), 06.-10, März 2023, Dresden, Germany, Proceedings (LNI)*, Birgitta König-Ries, Stefanie Scherzinger, Wolfgang Lehner, and Gottfried Vossen (Eds.), Vol. P-331. Gesellschaft für Informatik e.V., 511–533. https://doi.org/10.18420/BTW2023-24

[17] Iurii Konovalenko and André Ludwig. 2019. Event processing in supply chain management - The status quo and research outlook. *Comput. Ind.* 105 (2019), 229–249. https://doi.org/10.1016/j.compind.2018.12.009

[18] Maria Kosche, Tore Koß, Florin Manea, and Stefan Siemer. 2022. Combinatorial Algorithms for Subsequence Matching: A Survey. In *Proceedings 12th International Workshop on Non-Classical Models of Automata and Applications, NCMA 2022, Debrecen, Hungary, August 26-27, 2022*. 11–27. https://doi.org/10.4204/EPTCS.367.2

[19] Yan Li and Tingjian Ge. 2021. Imminence Monitoring of Critical Events: A Representation Learning Approach. In *Proceedings of the 2021 International Conference on Management of Data* (Virtual Event, China) *(SIGMOD/PODS '21)*. Association for Computing Machinery, New York, NY, USA, 1103–1115. https://doi.org/10.1145/3448016.3452804

[20] Kay Makowsky. 2022. *Discovery of Complex Event Queries via Representation Learning*. Bachelor's Thesis. Humboldt-Universität zu Berlin, Berlin, Germany.

[21] Alessandro Margara, Gianpaolo Cugola, and Giordano Tamburrelli. 2014. Learning from the Past: Automated Rule Generation for Complex Event Processing. In *Proceedings of the 8th ACM International Conference on Distributed Event-Based Systems* (Mumbai, India) *(DEBS '14)*. Association for Computing Machinery, New York, NY, USA, 47–58. https://doi.org/10.1145/2611286.2611289

[22] 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 *Proceedings of the 11th ACM International Conference on Distributed and Event-Based Systems* (Barcelona, Spain) *(DEBS '17)*. Association for Computing Machinery, New York, NY, USA, 158–169. https://doi.org/10.1145/3093742.3093917

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

[24] Charles Reiss, Alexey Tumanov, Gregory R Ganger, Randy H Katz, and Michael A Kozuch. 2012. Towards understanding heterogeneous clouds at scale: Google trace analysis. *Intel Science and Technology Center for Cloud Computing, Tech. Rep* 84 (2012), 1–12.

[25] Charles Reiss, John Wilkes, and Joseph L Hellerstein. 2011. Google cluster-usage traces: format+ schema. *Google Inc., White Paper* 1 (2011), 1–14.

[26] Rebecca Sattler, Sarah Kleest-Meißner, Steven Lange, Markus L. Schmid, Nicole Schweikardt, and Matthias Weidlich. 2024. *DISCES: Systematic Discovery of Event Stream Queries – Technical Report*. Technical Report. https://github.com/rebesatt/disces

[27] Suad Sejdovic, Yvonne Hegenbarth, Gerald H. Ristow, and Roland Schmidt. 2016. Proactive disruption management system: how not to be surprised by upcoming situations. In *Proceedings of the 10th ACM International Conference on Distributed and Event-based Systems, DEBS '16, Irvine, CA, USA, June 20 - 24, 2016*, Avigdor Gal, Matthias Weidlich, Vana Kalogeraki, and Nalini Venkasubramanian (Eds.). ACM, 281–288. https://doi.org/10.1145/2933267.2933271

[28] 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* (2021), 1–30.

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

[30] Jianyong Wang and Jiawei Han. 2004. BIDE: Efficient mining of frequent closed sequences. In *Proceedings. 20th international conference on data engineering*. IEEE, 79–90.

[31] Xifeng Yan, Jiawei Han, and Ramin Afshar. 2003. CloSpan: Mining: Closed sequential patterns in large datasets. In *Proceedings of the 2003 SIAM international conference on data mining*. SIAM, 166–177.

[32] Bin Yu, William HK Lam, and Mei Lam Tam. 2011. Bus arrival time prediction at bus stop with multiple routes. *Transportation Research Part C: Emerging Technologies* 19, 6 (2011), 1157–1170.

[33] Haopeng Zhang, Yanlei Diao, and Neil Immerman. 2014. 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*, Curtis E. Dyreson, Feifei Li, and M. Tamer Özsu (Eds.). ACM, 217–228. https://doi.org/10.1145/2588555.2593671

Algorithm 7: MergeAttributeQueries

```
Input:  Dictionary of matching queries for each attribute $Q_{\mathcal{A}}$,
        parent dictionary $P$, event schema $\mathcal{A}$, stream database $D$,
        sequence of query variables $\mathcal{X}^{<}$.
Output: Set of descriptive queries $Q_d$.
$Q_m \leftarrow \emptyset$;
$\Gamma_D \leftarrow \{a \in \Gamma \mid \forall S \in D : \exists j \in \{1,...,|S|\}, i \in \{1,...,n\} : S[j].A_i = a\}$;
/* All combinations of queries from different attributes */
$L \leftarrow \bigtimes_{A \in \mathcal{A}} Q_{\mathcal{A}}(A)$;
/* Stream matches dictionary */
$T \leftarrow \emptyset$;
foreach $(q_1, \ldots, q_n) \in L$ do
    $I \leftarrow \emptyset$;
    foreach $q \in \{q_1, \ldots, q_n\}$ do $I(q) \leftarrow \emptyset$;
    foreach $q \in (q_1, \ldots, q_n)$ do
        /* Instance positions of query $q$ in $D$ */
        $I(q) \leftarrow$ GetInstances$(q, D)$;
        if $\forall q' \in \mathrm{dom}(P) : P(q') \neq q$ then
            $P \leftarrow$ ChildQueries$(q, \mathcal{A}, \Gamma_D, P, \mathcal{X}^{<}, True)$;
    /* All combinations of instance positions of queries
       from different attributes */
    $B \leftarrow \bigtimes_{q_A \in \{q_1,...,q_n\}} I(q_A)$;
    foreach $(q'_1, \ldots, q'_n) \in B$ do
        /* Translate instance tuple to query */
        $q_m \leftarrow$ Instance2Query$((q'_1, \ldots, q'_n), (q_1, \ldots, q_n))$;
        $P \leftarrow$ ChildQueries$(q_m, \mathcal{A}, \Gamma_D, P, \mathcal{X}^{<}, True)$;
        $q_p \leftarrow P(q_m)$;
        $M, T \leftarrow$ Match$(q_m, \mathcal{A}, D, q_p, T)$;
        /* Add merged query to result set */
        if $M = True$ then $Q_m \leftarrow Q_m \cup \{q_m\}$ ;
return $Q_m$;
```

Algorithm 8: DescriptiveQueries

```
Input:  Set of queries $Q$, supported alphabet $\Gamma_D$, event schema $\mathcal{A}$,
        sequence of query variables $\mathcal{X}^{<}$ over the set of query variables $\mathcal{X}$.
Output: Set of descriptive queries $Q_d$.
1  $Q_n \leftarrow \emptyset$;
   /* For each query $q$ in the query set $Q$, create queries
      which are less strict and therefore not descriptive */
2  foreach $q \in Q$ do
3      $R \leftarrow \emptyset$;
4      foreach $s \in \Gamma_D \cup \mathcal{X}$ do $R(s) \leftarrow \emptyset$;
5      foreach $i \in \{1, \ldots, |q|\}$ do
6          foreach $A \in \mathcal{A}$ do
7              if $q[i].A \in \Gamma_D \cup \mathcal{X}$ then
8                  $q_n \leftarrow q$;
9                  $q_n[i].A \leftarrow \_$;
10                 $Q_n \leftarrow Q_n \cup$ NormalForm$(q_n)$;
11                 $R(q[i].A) \leftarrow R(q[i].A) \cup \{i\}$;
12     foreach $s \in \mathrm{dom}(R)$ do
13         if $s \in \mathcal{X} \wedge |R(s)| \geq 4$ then
               /* Find all combinations of partial replacements
                  of variable $s$ in query $q$ */
14             $G \leftarrow$ GenerateReplacements$(q, s)$;
15             $Q_n \leftarrow Q_n \cup G$;
16         if $s \in \Gamma_D \wedge |R(s)| \geq 2$ then
               /* Find all combinations of partial and complete
                  replacement of attribute type $s$ in query $q$ */
17             $G \leftarrow$ GenerateReplacements$(q, s)$;
18             $Q_n \leftarrow Q_n \cup G$;
19 $Q_d \leftarrow Q \backslash Q_n$;
20 return $Q_d$;
```

A APPENDIX

In this appendix, we formalize the algorithms that could not be included in the main part of the paper, due to space restrictions.

A.1 Merge Attribute Queries

Our approach to merge queries found per attribute is defined as MergeAttributeQueries in Alg. 7. It takes as input a dictionary of matching queries for each attribute $Q_{\mathcal{A}} : \mathcal{A} \rightarrow 2^{Q}$, a parent dictionary P, an event schema $\mathcal{A}$, a stream database D, and a sequence of query variables $\mathcal{X}^{<}$; and returns a set of merged queries.

In essence, the algorithm is based on an exhaustive combination of all combinations of queries sourced from distinct attributes. Each resulting combination then is handled in several steps:

- Instance Generation: Initially, for every query found for an attribute, we systematically generate all feasible instances of the query within a single stream of the database.
- Merge and Translation: Subsequently, we consolidate the instances of the queries per attribute, and translate them into a unified query.
- Query Matching: Following the merge, the obtained query is matched against the stream database. If a match is identified, the merged query is integrated into the result set.

A.2 Descriptive Queries

Our approach to select only the descriptive queries in a set of queries is given as DescriptiveQueries in Alg. 8. Given a set of queries Q, a supported alphabet Γ_D, an event schema $\mathcal{A}$, and a sequence of query variables $\mathcal{X}^{<}$ with $\mathcal{X}$ as the underlying set of query variables, it proceeds as follows. We iterate over each query q within the given set and, for each query, generate queries that are less specific than q. We generate less specific queries in three different ways:

- Replace each symbol (variable or attribute value) within the query by the placeholder (line 5-11). For every replacement, a new non-descriptive query is added to the set of non-descriptive queries Q_n.
- Partially replace variables that appear at least four times within a query by a new variable that is not used in query q (line 13-15).
- Partially and completely replace attribute types that appear at least twice within the given query (line 16-18).

At the end, we calculate the set of descriptive queries Q_d by subtracting all queries in Q_n from the given set of queries Q (line 19).

The above approach relies on function NormalForm, which checks, if a variable occurs at least twice, and otherwise replaces it with the placeholder _. Second, it restores the order of the variables, such that the first occurrences of the variables appear in the same order in the query q as they do in $\mathcal{X}^{<}$.

Moreover, function GenerateReplacements takes as input a query q and a symbol s, i.e., an attribute value or a variable that is to be replaced by a variable, which is not used in the query. The replacement routine first iterates through the number of symbols that can be replaced. In case of a variable (an attribute value) it is $[2, R(s) - 2]$ ($[2, R(s) - 1]$). Then, for each group size, it finds all combinations of possible replacements for the current query q.

Algorithm 9: D-U-S

```
Input:  Stream database $D$, event schema $\mathcal{A}$, sequence of query var. $\mathcal{X}^<$.
Output: Set of descriptive queries $Q$.
   /* Set of queries and parent dict. per attribute                  */
1  $Q_{\mathcal{A}}, P_{\mathcal{A}} \leftarrow \emptyset$;
2  foreach $A \in \mathcal{A}$ do
3      $Q_{\mathcal{A}}(A), P_{\mathcal{A}}(A) \leftarrow \emptyset, \emptyset$;
   /* Use D-U-C to discover queries per attribute                    */
4  foreach $A \in \mathcal{A}$ do
5      $D_A \leftarrow \{\langle (e_1.A), \ldots, (e_l.A) \rangle \mid \langle e_1, \ldots, e_l \rangle \in D\}$;
6      $Q_{\mathcal{A}}(A), P_{\mathcal{A}}(A) \leftarrow$ D-U-C$(D_A, (A), \mathcal{X}^<, False)$;
7  $P \leftarrow \cup_{A \in \mathcal{A}} P_{\mathcal{A}}(A)$;
   /* Merge queries found per attribute                              */
8  $Q \leftarrow$ MergeAttributeQueries$(Q_{\mathcal{A}}, P, \mathcal{A}, D, \mathcal{X}^<)$;
9  $Q \leftarrow$ DescriptiveQueries$(Q, \Gamma_D, \mathcal{A}, \mathcal{X}^<)$;
10 return $Q$;
```

Algorithm 10: B-S-S

```
Input:  Stream database $D$, event schema $\mathcal{A}$, sequence of query var. $\mathcal{X}^<$.
Output: Set of descriptive queries $Q$.
   /* Set of queries and parent dict. per attribute                  */
1  $Q_{\mathcal{A}}, P_{\mathcal{A}} \leftarrow \emptyset$;
2  foreach $A \in \mathcal{A}$ do
3      $Q_{\mathcal{A}}(A), P_{\mathcal{A}}(A) \leftarrow \emptyset, \emptyset$;
   /* Use B-S-C to discover queries per attribute                    */
4  foreach $A \in \mathcal{A}$ do
5      $D_A \leftarrow \{\langle (e_1.A), \ldots, (e_l.A) \rangle \mid \langle e_1, \ldots, e_l \rangle \in D\}$;
6      $Q_{\mathcal{A}}(A), P_A \leftarrow$ B-S-C$(D_A, (A), \mathcal{X}^<, False)$;
7  $P \leftarrow \cup_{A \in \mathcal{A}} P_{\mathcal{A}}(A)$;
   /* Merge queries found per attribute                              */
8  $Q \leftarrow$ MergeAttributeQueries$(Q_{\mathcal{A}}, P, \mathcal{A}, D, \mathcal{X}^<)$;
9  $Q \leftarrow$ DescriptiveQueries$(Q, \Gamma_D, \mathcal{A}, \mathcal{X}^<)$;
10 return $Q$;
```

Example - Merging Attribute Queries

Let us consider stream S_2 from Example 1, the following queries per attribute, and their matching instances within the stream:

Attribute	Query	Positions
Job Id	$\varepsilon, q_1 = \langle (2, \_, \_) \rangle$	$(e_{21}), (e_{23})$
Status	$\varepsilon, q_2 = \langle (\_, \text{schedule}, \_)\, (\_, \text{schedule}, \_) \rangle$	(e_{21}, e_{22})
Priority	$\varepsilon, q_3 = \langle (\_, \_, \text{low}) \rangle$	$(e_{21}), (e_{23})$

The combination of queries found for different attributes leads to the following set:

$$\{(q_1, q_2, q_3), (q_1, q_2, \varepsilon), (q_1, \varepsilon, q_3), (\varepsilon, q_2, q_3),$$
$$(q_1, \varepsilon, \varepsilon), (\varepsilon, \varepsilon, q_3), (\varepsilon, q_2, \varepsilon), (\varepsilon, \varepsilon, \varepsilon)\}.$$

The connection of the different position for the query tuple (q_1, q_2, q_3) leads to the following merged queries:

$$\langle (2, \text{schedule}, \text{low}), (\_, \text{schedule}, \_) \rangle$$
$$\langle (\_, \text{schedule}, \_), (\_, \text{schedule}, \_), (2, \_, \text{low}) \rangle$$
$$\langle (2, \text{schedule}, \_), (\_, \text{schedule}, \_), (\_, \_, \text{low}) \rangle$$
$$\langle (\_, \text{schedule}, \text{low}), (\_, \text{schedule}, \_), (2, \_, \_) \rangle$$

which then have to be matched against the complete stream database. This process would have to be repeated for all combinations of query tuples to get all possible merged queries.

For example, let s be a variable and $R(s)$=5. Let the positions of s within the query be $[1, 2, 3, 4, 5]$. Then the possible group sizes are 2 and 3. For group size 2, there are $\binom{5}{2} = 10$ combinations: $(1, 2), (1, 3), \ldots, (4, 5)$. For group size 3, there are $\binom{5}{3} = 10$ combinations: $(1, 2, 3), (1, 2, 4), \ldots, (3, 4, 5)$. For each of the calculated positions, s is replaced by the same variable x which is not used in q. The result is a set of queries which are less specific than query q and, therefore, not descriptive.

A.3 D-U-S & B-S-S

D-U-S: DFS, pattern-type unified, attribute separated. As mentioned, the D-U-S algorithm, presented in Alg. 9, is a variant of the D-U-C algorithm, which also employs DFS and constructs queries in a unified manner. It discovers queries separately per attribute, before merging them to obtain the final result. The sets of queries per attribute are maintained in a dictionary $Q_{\mathcal{A}} : \mathcal{A} \rightarrow 2^{\mathcal{Q}}$, i.e., $Q_{\mathcal{A}}(A)$ is the set of queries found for attribute $A \in \mathcal{A}$. The algorithm first constructs the projections of the streams on the attribute (line 4), which are then used to run the D-U-C algorithm (line 5). Note that setting flag b to *False* means that the D-U-C algorithm will return all found queries, not only those that are descriptive. The queries discovered per attribute are then merged, as detailed above, before the non-descriptive queries are removed from the result.

B-S-S: BFS, pattern-type separated, attribute separated. The B-S-S algorithm is defined in Alg. 10. It is a variant of B-S-C, adopting BFS and the separated discovery of type queries and pattern queries. Yet, it first considers queries per attribute, which are again collected in a dictionary $Q_{\mathcal{A}} : \mathcal{A} \rightarrow 2^{\mathcal{Q}}$, before merging them using the algorithm introduce above. As such, it adopts the strategy explained above for D-U-S, just with B-S-C as the base algorithm.