







# Embracing Change: Incremental Updates of Discovered Event Queries

Rebecca Sattler <sup>1</sup>, Sarah Kleest-Meißner <sup>1</sup>, Steven Lange <sup>1</sup>, Markus Schmid <sup>1</sup>, Nicole Schweikardt <sup>1</sup>, and Matthias Weidlich <sup>1</sup>

**Abstract:** In complex event processing (CEP), queries are evaluated continuously over streams of events to detect situations of interest, thereby facilitating reactive applications. However, users often lack insights into the precise event pattern that characterizes the situation, which renders the definition of the respective queries challenging. Once a database of finite, historic streams, each containing a materialization of the situation of interest, is available, query discovery supports users in the definition of the desired queries. It constructs the queries that match a certain share of the given streams, as determined by a support threshold. Yet, upon changes in the database or changes of the support threshold, existing algorithms need to construct the resulting queries from scratch, neglecting the queries obtained in previous runs. In this paper, we aim to avoid the resulting inefficiencies by techniques for incremental query discovery. We first provide a theoretical analysis of the problem context, before presenting algorithmic solutions to cope with changes in the stream database or the adopted support threshold. Our experiments using real-world data show that our incremental query discovery reduces the runtimes by up to three orders of magnitude compared to a baseline solution.







**Keywords:** Complex Event Processing, Event Streams, Query Discovery, Incremental Updates

## 1 Introduction

Complex event processing (CEP) is a computational paradigm to facilitate reactive and proactive applications [CA12, Gi20]. In CEP, a set of queries are evaluated continuously over a stream of events in order to detect situations of interest in domains, such as urban transportation [Ar14], computational finance [Ch10], and supply chain management [KL19]. Often, these situations of interest are deviating or abnormal behaviour, so that their timely detection facilitates the implementation of counter-measures.

In essence, CEP queries capture sequential patterns of events, i.e., they describe a sequence of events of specific types, additional conditions on the attribute values of the respective events, and a time window for their occurrence. As the exact characterization of the event patterns is challenging in practice, one may support users in the specification of queries with techniques for automated query discovery [MCT14, GCW16, K123]. Here, the assumption is

---

<sup>1</sup> Humboldt-Universität zu Berlin, Institut für Informatik, Unter den Linden 6, 10099 Berlin, Germany, [sattlere@hu-berlin.de](mailto:sattlere@hu-berlin.de),  <https://orcid.org/0000-0002-7342-7131>;  
[kleest-meissner.sarah@alumni.hu-berlin.de](mailto:kleest-meissner.sarah@alumni.hu-berlin.de),  <https://orcid.org/0000-0002-4133-7975>;  
[langestx@hu-berlin.de](mailto:langestx@hu-berlin.de),  <https://orcid.org/0009-0001-7217-8188>;  
[mlschmid@mlschmid.de](mailto:mlschmid@mlschmid.de),  <https://orcid.org/0000-0001-5137-1504>;  
[schweikn@informatik.hu-berlin.de](mailto:schweikn@informatik.hu-berlin.de),  <https://orcid.org/0000-0001-5705-1675>;  
[matthias.weidlich@hu-berlin.de](mailto:matthias.weidlich@hu-berlin.de),  <https://orcid.org/0000-0003-3325-7227>

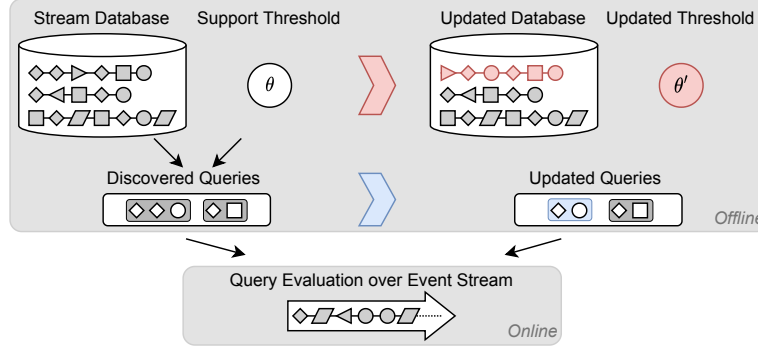


Fig. 1: The general setting of incremental event query discovery: An update of the stream database or support threshold shall be incorporated in an updated set of queries.

that a database of finite, historic (sub-)streams that cover the situation of interest is available. That is, if the situation of interest is some deviating or abnormal behaviour, the assumption is that the database contains only the respective streams, not the common behaviour. Then, queries that match a certain share of these streams, as determined by a support threshold, can be extracted and subsequently be assessed and validated by a user.

However, over time, the database of streams used as a basis for query discovery may be subject to change. Additional streams that capture a situation of interest may become available, or streams that are already in the database may be considered outdated. As part of that, the requirements for query discovery in terms of the adopted support threshold may also change, i.e., it may be relaxed to increase the number of discovered queries, or strengthened to avoid false positives in the result.

Yet, upon changes in the database or in the support threshold, existing discovery algorithms need to construct the resulting queries from scratch. This is problematic since solving the problem of event query discovery is computationally hard, as an exponentially growing space of candidate queries (in the query length and attribute domains) needs to be explored and the evaluation of a single query also shows an exponential runtime complexity (in the number of variables, i.e., conditions over multiple events) [K122].

In this paper, we set out to avoid inefficient recomputation with strategies for incremental event query discovery. Our idea is that the discovery result shall be reused and adapted once the databases or the support threshold change, as illustrated in Figure 1. To this end, after giving formal preliminaries (§2), we make the following contributions:

- (i) We present a theoretical analysis of the problem context (§3). Specifically, we show that under certain conditions, the result set of queries for an updated stream database or an updated support threshold denotes a subset of the existing result.
- (ii) We present algorithmic solutions for incremental query discovery (§4). They adapt a given set of queries, thereby aiming to avoid inefficient recomputation.

We assess the feasibility of our solution in several experiments (§5), showing that incremental discovery reduces the runtimes by up to 10x. Finally, we review related work (§6) and conclude the paper (§7).

## 2 Background on Query Discovery

Below, we introduce preliminary notions to study the problem of event query discovery.

**Event streams.** We based our work on the notion of a multivariate event stream. Such a stream has an event schema, captured by a tuple of attributes  $\mathcal{A} = (A_1, \dots, A_n)$ , where  $\text{dom}(A_i)$  is the finite domain of attribute  $A_i$ ,  $1 \leq i \leq n$ . All events have the same schema and we assume that attribute domains are disjoint,  $\cap_{i=1}^n \text{dom}(A_i) = \emptyset$ . As a short-hand, we write  $\Gamma = \bigcup_{i=1}^n \text{dom}(A_i)$  for the alphabet defined by all possible values of attributes. Moreover, an event is an instantiation of the event schema, captured as a tuple  $e = (a_1, \dots, a_n)$  with  $a_i \in \text{dom}(A_i)$  for  $1 \leq i \leq n$ . By  $e.A_i$ , we denote the value of attribute  $A_i$  of event  $e$ .

An event stream is a finite sequence of events  $S = \langle e_1, \dots, e_l \rangle$ , i.e., a finite subsequence of a potentially unbounded sequence of events. We write  $|S|$  for the length of the stream, and  $S[i]$  for its  $i$ -th event. A stream database is a finite set of streams  $D = \{S_1, \dots, S_d\}$ .

**Event queries.** We adopt a query model that captures sequences of events, which are correlated by predicates over their attribute values [ZDI14]. We neglect the definition of an explicit time window, as it may be derived from the maximal time difference between events in one of the streams. That is, taking the time between the start and end of each stream, the maximal time span over all streams is an upper bound for the time interval, in which a situation of interest is expected to occur. Moreover, we follow [K123] and use a linearized representation of event queries. Let  $\mathcal{A} = (A_1, \dots, A_n)$  be an event schema,  $\mathcal{X}$  be a finite set of variables such that  $\mathcal{X} \cap \Gamma = \emptyset$ , and  $\_ \notin \mathcal{X} \cup \Gamma$  be a placeholder symbol. Then, we define a query term  $t = (u_1, \dots, u_n)$  as an  $n$ -tuple comprising attribute values, variables, and placeholders:  $u_j \in \text{dom}(A_j) \cup \mathcal{X} \cup \{\_ \}$ ,  $1 \leq j \leq n$ , while we prohibit terms containing only placeholders, i.e.,  $u_j \in \text{dom}(A_j) \cup \mathcal{X}$  for at least one  $1 \leq j \leq n$ . By  $\varepsilon$ , we denote the empty query term that contains only placeholders, i.e.,  $\varepsilon = (\_, \dots, \_)$ . Based thereon, an event query is a finite sequence of query terms,  $q = \langle t_1, \dots, t_k \rangle$ , 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  $q[i] = (u_1, \dots, u_n)$  with  $u_j \in \mathcal{X}$ ,  $1 \leq i \leq k$  and  $1 \leq j \leq n$ , there exists another query term  $q[p] = (u'_1, \dots, u'_n)$  with  $u_j = u'_j$ ,  $1 \leq p \leq k$  and  $p \neq i$ . The empty query is  $\langle \varepsilon \rangle$ , i.e., the query containing the empty query term. By  $\mathcal{Q}$ , we denote the universe of all queries.

From an intuitive point of view, a query term describes a single event that shall be matched by defining a specific value, variable, or placeholder per attribute. While a placeholder denotes the absence of any constraint, the use of variables captures the correlation of events as matching events need to have equivalent values for the respective attributes.

Tab. 1: Overview of notations.

Notation	Explanation
$\mathcal{A} = (A_1, \dots, A_n)$	An event schema, a tuple of attributes
$\Gamma = \bigcup_{i=1}^n \text{dom}(A_i)$	The stream alphabet
$e = (a_1, \dots, a_n)$	An event, a tuple of attribute values
$e.A_i$	The value $a_i$ of attribute $A_i$ of event $e$
$S = \langle e_1, \dots, e_l \rangle$	An event stream, a finite sequence of events
$S[i]$	The $i$ -th event of the event stream $S$
$D = \{S_1, \dots, S_d\}$	A stream database, a set of event streams
$\mathcal{X}$	A finite set of query variables, $\mathcal{X} \cap \Gamma = \emptyset$
$\_$	A placeholder symbol, $\_ \notin \mathcal{X} \cup \Gamma$
$t = (u_1, \dots, u_n)$	A query term, an $n$ -tuple of attribute values, variables, and placeholders
$\varepsilon = (\_, \dots, \_)$	The empty query term, an $n$ -tuple of placeholders
$q = \langle t_1, \dots, t_k \rangle$	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
$M_+(q, D)$	The set of streams in $D$ that match the query $q$
$\text{supp}(q, D)$	The support of query $q$ in stream database $D$
$\theta$	The support threshold, a real number in the interval $(0, 1]$
$s( D , \theta)$	The stream threshold, the min number of streams to match to fulfill the support threshold
$\Gamma(D, \theta)$	The frequent stream alphabet
$\mathcal{Q}_D^\theta$	The set of all matching queries

Moreover, we distinguish three classes of queries: A type query contains only query terms defining attribute values (and placeholders). A pattern query contains only query terms defining variables (and placeholders). All other queries are referred to as mixed queries.

We define the semantics of a query  $q = \langle t_1, \dots, t_k \rangle$  over a stream  $S = \langle e_1, \dots, e_l \rangle$  as follows: A match of  $q$  in  $S$  is an injective mapping  $m : \{1, \dots, k\} \rightarrow \{1, \dots, l\}$ , such that:

- for each query term  $q[i] = (u_1, \dots, u_n)$ ,  $1 \leq i \leq k$ , the mapped event  $S[m(i)] = (a_1, \dots, a_n)$  has the required attribute values,  $u_j = a_j$  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  $q[i] = (u_1, \dots, u_n)$  and  $q[p] = (u'_1, \dots, u'_n)$ ,  $1 \leq i, p \leq k$ , 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 events in the stream is preserved, i.e.,  $m(i) < m(p)$  for  $1 \leq i < p \leq k$ .

Note that the set of matches as induced by the above definition corresponds to a greedy event selection policy (also known as unconstrained or skip-till-any-match). It does not limit how often an event can participate in matches. However, in the remainder, we do not explicitly refer to the set of matches, but consider only the existence of a match per stream. Specifically, we say that  $q$  matches  $S$  and write  $S \models q$ , if there exists a match for query  $q$  in stream  $S$ . Table 1 provides an overview of the introduced notations.

**Support.** To reason about the relation of a query  $q$  and a stream database  $D$ , we define  $M_+(q, D)$  as the set of matching streams, i.e.,  $M_+(q, D) = \{S \in D \mid S \models q\}$ . The support  $\text{supp}(q, D)$  of a query is then given as the fraction of streams in  $D$  that match the query,  $\text{supp}(q, D) = \frac{|M_+(q, D)|}{|D|}$ . A support threshold  $\theta$  is a real number in the interval  $(0, 1]$ . A query  $q$  is frequent in  $D$ , if  $\text{supp}(q, D) \geq \theta$ . From the aforementioned notions, we derive the stream threshold  $s(|D|, \theta)$  as the minimum number of streams that must match to a query in order for the query to be considered frequent, i.e.,  $s(|D|, \theta) = \lceil |D| \cdot \theta \rceil$ . Based thereon, we also derive the frequent stream alphabet, defined by all attribute values that appear in at least the number of streams as determined by the stream threshold, i.e.,  $\Gamma(D, \theta) = \{a \in \Gamma \mid |\{S \in D \mid \exists j \in \{1, \dots, |S|\}, i \in \{1, \dots, n\} : S[j].A_i = a\}| \geq s(|D|, \theta)\}$ . Furthermore, as a short-hand, we define the set of all matching queries as  $Q_D^\theta = \{q \mid \text{supp}(q, D) \geq \theta\}$ .

For a given stream database, it is clear that support thresholds induce a containment relation for the sets of matching queries. As we later rely on this property, we state it as follows:

**Property 1** *Let  $D$  be a stream database and  $\theta_1, \theta_2$  be support thresholds. Let  $Q_D^{\theta_1}$  and  $Q_D^{\theta_2}$  be the sets of all matching queries. If  $\theta_2 \geq \theta_1$ , then  $Q_D^{\theta_2} \subseteq Q_D^{\theta_1}$ .*

*Proof.* Let  $q \in Q_D^{\theta_2}$ . Then,  $\text{supp}(q, D) \geq \theta_2 \geq \theta_1$ , and thus,  $q \in Q_D^{\theta_1}$  and  $Q_D^{\theta_2} \subseteq Q_D^{\theta_1}$ .  $\square$

Table 2 shows a database of three streams from a cluster monitoring application [Re12]. Each event has three attributes, denoting the ID, status, and priority of a job that is executed on a compute cluster. Now, consider the query  $q = \langle (x_0, \text{schedule}, \text{low}), (\_, \text{schedule}, \_), (x_0, \text{evict}, \text{low}) \rangle$ . It matches streams  $S_2$  and  $S_3$ , as, e.g., the three query terms of  $q$  can be mapped to events  $e_{32}$ ,  $e_{34}$ , and  $e_{35}$  of  $S_3$  (the mapped events are highlighted in blue). Note that these are the only possible mappings for this scenario, since the order of the query needs to be preserved. Yet, the query does not match  $S_1$ , as the third query term cannot be mapped to any event ( $e_{14}$ , highlighted in red, does not qualify to be mapped). With a support threshold of 0.6, query  $q$  would be in the set of matching queries for the stream database.

### 3 Analysis of the Problem Context

In this section, we study the problem of event query discovery under updates, focusing especially on the relation of the sets of matching queries before and after an update. We first characterize the problem context. Let  $D_0 = \{S_1, \dots, S_d\}$  be a stream database,  $\theta_0$  be a support threshold, and  $Q_{D_0}^{\theta_0}$  be a matching query set. For this setting, we distinguish two possible updates, as follows, which may also be combined:

- The stream database  $D_0$  is updated resulting in a new database  $D_1$ . Here, we consider the addition of  $k$  new streams, i.e.,  $D_1 = D_0 \cup \{S_{d+1}, \dots, S_{d+k}\}$ , as well as the removal of  $k$  existing streams, i.e.,  $D_1 = D_0 \setminus D^-$  with  $D^- \subseteq D_0$  and  $|D^-| = k$ .
- The support threshold  $\theta_0$  is updated resulting in a new threshold  $\theta_1 \in (0, 1]$ .

Tab. 2: Example stream database.

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

These updates potentially have implications for the set of matching queries,  $Q_{D_0}^{\theta_0}$ , which, hence, needs to be updated to  $Q_{D_0}^{\theta_1}$ ,  $Q_{D_1}^{\theta_0}$ , or  $Q_{D_1}^{\theta_1}$ , respectively. Specifically, we seek to answer the following questions:

1. Which queries are still matching? That is, which queries  $q \in Q_{D_0}^{\theta_0}$  do still belong to  $Q_{D_0}^{\theta_1}$ ,  $Q_{D_1}^{\theta_0}$ , or  $Q_{D_1}^{\theta_1}$ , respectively?
2. Which new queries are matching? That is, which queries  $q \notin Q_{D_0}^{\theta_0}$  belong to  $Q_{D_0}^{\theta_1}$ ,  $Q_{D_1}^{\theta_0}$ , or  $Q_{D_1}^{\theta_1}$ , respectively?

Ideally, the above questions shall be answered incrementally. That is, we would like to avoid computing  $Q_{D_0}^{\theta_1}$ ,  $Q_{D_1}^{\theta_0}$ , or  $Q_{D_1}^{\theta_1}$  from scratch, but rather derive it directly from  $Q_{D_0}^{\theta_0}$ .

Before turning to algorithmic solutions to the above questions, we study the three relations between the matching query sets as illustrated in Figure 2.

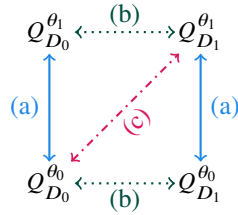


Fig. 2: Illustration of matching query sets and the relations between them.

In this paper, we focus on the case that  $D_1$  is derived from  $D_0$  by the addition of  $k$  streams, i.e.,  $D_1 = D_0 \cup \{S_{d+1}, \dots, S_{d+k}\}$ , and that the support threshold is strengthened, i.e.,  $0 < \theta_0 \leq \theta_1 \leq 1$ . The reason being that the properties discussed below hold analogously for

the mirrored updates, i.e., the deletion of existing streams and a relaxed support threshold. In the remainder, we first study the relation of the sets of matching queries when only the support threshold is updated (a), and when only the stream database is updated (b). Afterwards, we turn to the case that both are updated at the same time (c).

#### (a): Updating only the Support Threshold

In case only the support threshold is changed, i.e., it is strengthened, we note that there exists a containment relation between the sets of matching queries. Based on Property 1, we conclude that the set of matching queries for the increased support threshold is a subset of the one for the lower support threshold, i.e.,  $Q_{D_i}^{\theta_i} \subseteq Q_{D_i}^{\theta_0}$ ,  $i \in \{0, 1\}$ .

#### (b): Updating only the Stream Database

Regarding updates of the stream database, i.e., the addition of streams, we make the following observation based on the stream thresholds, i.e., the minimal numbers of streams that need to match to fulfill the support threshold. Consider the case that  $s(|D_1|, \theta_i) = s(|D_0|, \theta_i) + k$ ,  $i \in \{0, 1\}$ , where  $k$  represents the number of added streams. Intuitively, this means that all newly added streams need to be matched for a query to remain in the set of matching queries, under a worst-case assumption. That is, all queries that only match the minimal numbers of streams to fulfill the support threshold need to match all the new streams. We note though that, with a support threshold of  $\theta_i = 1$ , this worst case applies to all queries in the set of matching queries. If  $s(|D_1|, \theta_i) = s(|D_0|, \theta_i) + k$ , then the set of matching queries for the updated stream database can only become smaller, i.e., it is a subset of the original set. This observation on a containment relation between the matching sets of queries under updates to the stream database is captured by the following property:

**Property 2**  $s(|D_1|, \theta_i) = s(|D_0|, \theta_i) + k, i \in \{0, 1\} \implies Q_{D_1}^{\theta_i} \subseteq Q_{D_0}^{\theta_i}$ .

The proof of this property is directly covered by Lemma 1, which captures a more general case as detailed below. For now, we illustrate what happens if the respective relation between the stream thresholds does not hold true with an example.

Consider the streams  $S_1, S_2, S_3$ , as given in Table 2. Let  $D_0 = \{S_1, S_2\}$  be the initial stream database and let the support threshold be  $\theta_0 = 0.6$ . Then, the stream threshold is  $s(D_0, \theta_0) = 2$ , i.e., a query needs to match both streams to be frequent. Now, let  $D_1 = \{S_1, S_2, S_3\}$  be the stream database after adding one stream,  $S_3$ , while leaving the support threshold unchanged. Then, the stream threshold is still  $s(D_1, \theta_0) = 2$  and, hence,  $s(D_1, \theta_0) \neq s(D_0, \theta_0) + k$  as  $k = 1$  in our example.

As mentioned above, the query  $q = \langle (x_0, \text{schedule}, \text{low}), (\_, \text{schedule}, \_), (x_0, \text{evict}, \text{low}) \rangle$  matches  $S_2$  and  $S_3$ , but not  $S_1$ . This means that  $q \in Q_{D_1}^{\theta_1}$  and  $q \notin Q_{D_0}^{\theta_1}$ . As such, query  $q$

illustrates that the containment relation between the sets of matching queries does not hold and we conclude that, if  $s(D_1, \theta_0) \neq s(D_0, \theta_0) + k$ , then the relation between  $\mathcal{Q}_{D_1}^{\theta_1}$  and  $\mathcal{Q}_{D_0}^{\theta_1}$  depends on the added stream(s).

The above property and our example illustrate that there is a close relation between the amount of change applied to a stream database, i.e., the number  $k$  of added streams, the stream threshold, and the support threshold. In fact, the support threshold can be bounded based on the other two parameters. These bounds, in turn, enable us to derive insights on how often we can expect Property 2 to be applicable and, thereby, conclude that the set of matching queries of the updated database is a subset of the original set of matching queries.

As above, let  $D_0 = \{S_1, \dots, S_d\}$  and  $D_1 = D_0 \cup \{S_{d+1}, \dots, S_{d+k}\}$  be the original and updated databases and let  $\theta_0$  be the support threshold for both of them. To simplify the presentation, we use the short-hands  $s_0 = s(|D_0|, \theta_0)$  and  $s_1 = s(|D_1|, \theta_0)$  for the stream thresholds  $D_0$  and  $D_1$ , respectively. Then, it holds that:

$$s_1 = \lceil (d+k) \cdot \theta_0 \rceil \leq \lceil d \cdot \theta_0 \rceil + \lceil k \cdot \theta_0 \rceil \leq \lceil d \cdot \theta_0 \rceil + k = s_0 + k.$$

Based thereon, and considering the fact that  $s_0 \leq s_1 \leq s_0 + k$ , we conclude that  $s_1 \in \{s_0, \dots, s_0 + k\}$ .

Moreover, knowing that  $s_0 - 1 < \theta_0 \cdot d \leq s_0$ , we derive the following bounds for the support threshold based on the stream threshold  $s_0$  of  $D_0$ :

$$\frac{s_0 - 1}{d} < \theta_0 \leq \frac{s_0}{d}.$$

Analogously, we derive bounds for the support threshold  $\theta_0$  based on the stream threshold  $s_1$  of the updated database  $D_1$ :

$$\frac{s_1 - 1}{d+k} < \theta_0 \leq \frac{s_1}{d+k}.$$

Combining the above statements, we derive bounds for the support threshold based on the stream threshold of the original database. However, we need to distinguish two cases, i.e.,  $s_1 = s_0 + k$ , which means that all added streams need to be matched for a query to remain frequent under the worst-case assumption that it only matches exactly  $s_0$  streams; and  $s_1 < s_0 + k$ , which means that only some of the new streams need to be matched for any query to remain frequent:

**Property 3**  $s_1 < s_0 + k \implies \frac{s_0 - 1}{d} < \theta_0 \leq \frac{s_0 + k - 1}{d+k}$  and  $s_1 = s_0 + k \implies \frac{s_0 + k - 1}{d+k} < \theta_0 \leq \frac{s_0}{d}.$

The bounds as stated in Property 3 are useful to derive insights in the applicability of Property 2, i.e., they enable us to understand when  $s_1 = s_0 + k$  holds after  $k$  streams have been added to a stream database. Intuitively, the bounds on  $\theta_0$  characterize the change points for the stream threshold, depending on the size of a stream database  $d$  and the size of the



change  $k$ . We visualize the change points as determined by Property 3 in Figure 3. Here, we consider databases of different sizes,  $d \in \{10, 50, 100, 500, 1000\}$ , to which changes of size  $k \in \{1, 2, 5\}$  are applied. The orange ranges characterize the intervals of the support threshold  $\theta_0$  for which it holds that  $s_1 = s_0 + k$ , i.e., for which we know that the set of matching queries after the change is a subset of the original set of matching queries. The lower bound of the orange range is determined by the size of the database and the size of the change, and can be derived from Property 3 as follows:

$$\frac{s_0}{d} > \frac{s_0 + k - 1}{d + k} \implies \frac{s_0}{d} > \frac{k - 1}{k} \implies \theta_0 > \frac{k - 1}{k}.$$

From the visualization, we draw two main conclusions:

- The higher the support threshold, the more likely it is that Property 2 is applicable, so that the containment relation between the sets of matching queries holds true.
- The larger the size of the change, the less likely the property is applicable.

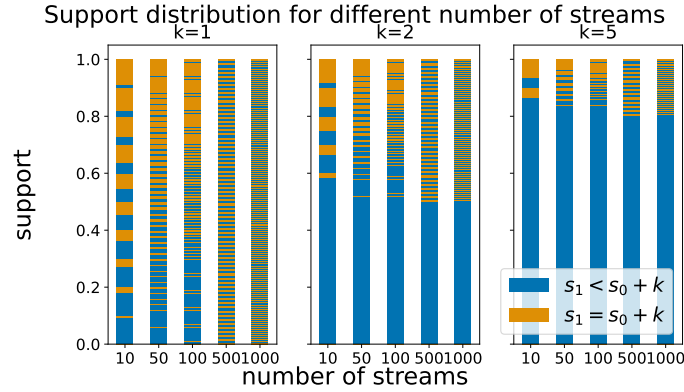


Fig. 3: Visualization of the change points for the stream threshold in relation to the support threshold, for different scenarios of database sizes and change sizes.

### (c): Updating both, the Stream Database and the Support Threshold

Finally, we consider the most general case of a change, where streams are added to the database and the support threshold is increased. Again, we note that the change in the stream thresholds that is induced by the change determines whether the sets of matching queries are in a containment relation, i.e., whether the change leads only to frequent queries being no longer frequent, but not to additional frequent queries. We capture this property as follows:

**Lemma 1** *Let  $D_0 = \{S_1, \dots, S_d\}$  and  $D_1 = D_0 \cup \{S_{d+1}, \dots, S_{d+k}\}$  be stream databases, and  $\theta_0$  and  $\theta_1$  with  $\theta_0 \leq \theta_1$  be support thresholds for  $D_0$  and  $D_1$ , respectively. Then, it holds that  $s(|D_1|, \theta_1) \geq s(|D_0|, \theta_0) + k \implies Q_{D_1}^{\theta_1} \subseteq Q_{D_0}^{\theta_0}$ .*

*Proof.* Let  $q_1$  be a query in  $Q_{D_1}^{\theta_1}$ . Recall that  $M_+(q_1, D_1)$  is the set of streams in  $D_1$  that match  $q_1$ ,  $M_+(q_1, D_0)$  is the set of streams in  $D_0$  that match  $q_1$ , and  $M_+(q_1, D_1 \setminus D_0)$  is the set of newly added streams that match  $q_1$ . Let  $|M_+(q_1, D_1 \setminus D_0)| = j$  be the number of these streams, for which it holds that  $j \in [0, k]$ . Then, it suffices to show that  $|M_+(q_1, D_0)| > s(|D_0|, \theta_0)$ . As before, we define  $s_0 = s(|D_0|, \theta_0)$  and  $s_1 = s(|D_1|, \theta_1)$ . Then, we derive the following bounds for the cardinality of the aforementioned sets:

$$|M_+(q_1, D_0)| = |M_+(q_1, D_1)| - |M_+(q_1, D_1 \setminus D_0)| \geq s_1 - j \geq s_0 + k - j \geq s_0.$$

Hence, indeed, it holds that  $Q_{D_1}^{\theta_1} \subseteq Q_{D_0}^{\theta_0}$ .  $\square$

We note that Lemma 1 provides a generalization of Property 2. That is, if  $\theta_1 = \theta_0$ , i.e., streams are added to the database, but the support threshold remains unchanged, due to the definition of the stream threshold,  $s_0 + k$  is an upper bound for  $s_1$ . As such, the condition of the implication on the containment relation becomes  $s_1 = s_0 + k$ .

The above lemma gives a condition, i.e.,  $s_1 \geq s_0 + k$ , to conclude on a containment relation for the sets of matching queries under an update of the stream database, regardless of the streams that are added.

Considering the general problem context as visualized in Figure 2, one might assume that  $Q_{D_1}^{\theta_1} \subseteq Q_{D_0}^{\theta_0} \implies Q_{D_1}^{\theta_0} \subseteq Q_{D_0}^{\theta_0}$  or  $Q_{D_1}^{\theta_1} \subseteq Q_{D_0}^{\theta_1}$ . That is, if the set of matching queries is only reduced by changes of potentially both, the stream database and the support threshold, the containment relation is expected to hold also if only one of the changes is considered. However, this is actually not the case, which we illustrate with a counter example in Table 3.

$d$	$d + 1$	$\theta_0$	$\theta_1$	$s(d, \theta_0)$	$s(d + 1, \theta_0)$	$s(d, \theta_1)$	$s(d + 1, \theta_1)$
99	100	0.8	0.9	80	80	90	90

Tab. 3: Stream thresholds under an example change in the stream database and the support threshold.

Here, we consider the setting of a stream database of size  $d = 99$ , for which one stream is added to obtain a database of size  $d + 1 = 100$ . Moreover, we consider a change of the support threshold from  $\theta_0 = 0.8$  to  $\theta_1 = 0.9$ . For this setting, it holds that  $s(d + 1, \theta_0) = s(d, \theta_0)$  as well as  $s(d + 1, \theta_1) = s(d, \theta_1)$ , i.e., the stream threshold does not change, if though a stream was added to the database. Hence, without information on this stream, we cannot conclude that  $Q_{D_1}^{\theta_0} \subseteq Q_{D_0}^{\theta_0}$  or  $Q_{D_1}^{\theta_1} \subseteq Q_{D_0}^{\theta_1}$ . However, the change of both, the stream database and the support threshold, actually leads to a change in the stream threshold, i.e.,  $s(d + 1, \theta_1) \geq s(d, \theta_0)$ . Based thereon, using Lemma 1, we can conclude on the containment relation for the sets of matching queries,  $Q_{D_1}^{\theta_1} \subseteq Q_{D_0}^{\theta_0}$ .

## 4 Algorithms for Incremental Discovery of Event Queries

In this section, we show how event query discovery is realized incrementally, under the changes discussed in the previous section. To this end, we first summarize a basic discovery

---

**Algorithm 1: DISCOVERY**

---

**Input:** Stream database  $D$ , support threshold  $\theta$ , frequent stream alphabet  $\Gamma(D, \theta)$ , set of queries  $Q$ , parent dictionary  $P$ , query stack  $O$ .

**Output:** Updated set of queries  $Q$ , updated parent dictionary  $P$ .

```
/* Explore queries that are pushed to the stack */
1 while  $O$  not empty do
2    $q \leftarrow O.\text{POP}()$ ;
3   if  $\text{MATCH}(q, D, \theta)$  then
4     /* For frequent queries, explore their children */
5      $Q \leftarrow Q \cup \{q\}$ ;
6      $P \leftarrow \text{CHILDQUERIES}(q, \Gamma(D, \theta), P)$ ;
7      $C \leftarrow \{q' \in \text{dom}(P) \mid P(q') = q\}$ ;
8     if  $C \neq \emptyset$  then
9       foreach  $q' \in C$  do  $O.\text{PUSH}(q')$ ;
9 return  $Q, P$ 
```

---

procedure that serves as a baseline (§4.1). Then, we discuss how to update the set of matching queries (i) once streams are added and/or the support threshold is increased (§4.2), and (ii) once existing streams are deleted and/or the support threshold is decreased (§4.3).

#### 4.1 Basic Algorithm for Event Query Discovery

As a basic algorithm for event query discovery, we consider an approach that searches through the space of candidate queries to collect all matching queries. To this end, it follows an iterative generate-and-test scheme. For each query that matches the stream database, stricter child queries are generated through the addition of query terms, or the insertion of variables and attribute values for placeholders in existing query terms. The child queries are, in turn, evaluated over the stream database. If they match, they become part of the result set and their child queries are considered further.

By starting the approach with an empty query, the approach iteratively explores the search space of candidate queries. It terminates once all child queries of all matching queries have been assessed.

The essence of this approach is captured by the function `DISCOVERY`, formalized in Alg. 1. It takes as input a stream database  $D$ ; a support threshold  $\theta$ ; the frequent stream alphabet  $\Gamma(D, \theta)$ ; a set of queries  $Q$  that are frequent in  $D$ ; a parent dictionary  $P$  that is modelled as a function  $P : Q \rightarrow Q$ , mapping child queries to their parent; and a stack  $O$  of queries that are frequent in  $D$  and still shall be explored by a depth-first search through the space of candidate queries.

Function `DISCOVERY` then explores the queries on the given stack. Each of them is evaluated against the stream database using the Boolean function `MATCH`, which returns true if query

$q$  is frequent in the stream database  $D$  as defined by the support threshold  $\theta$ ; otherwise, it returns false. If the query is frequent, it is added to the result set  $Q$  and its child queries are generated using function `CHILDQUERIES` and recorded in the aforementioned parent dictionary  $P$ . If such child queries have indeed been generated, i.e., set  $C$  is not empty, these queries are pushed to the stack for further exploration. When the stack is empty, the function returns the set of matching queries as well as the updated parent dictionary.

Moreover, we summarize the functionality of `CHILDQUERIES` to generate child queries of a query  $q$  as follows. It generates child queries by the insertion of a single symbol, i.e., a variable or an attribute value from the frequent stream alphabet, for a placeholder in one of the existing query terms. In addition, it also generates further child queries by appending a new query term to the query that is empty (i.e., comprises placeholders) except for one position that is filled with a symbol (variable or attribute value). In general, these insertions consider all possible attribute values, as well as a set of possible variables (for which the size is bounded by half of the length of the shortest stream in the database).

While we use function `DISCOVERY` also in our incremental approaches for event query discovery, a basic discovery algorithm simply instantiates by adding the empty query to the query stack and creating its entry in the parent dictionary. We formalize this instantiation in Alg. 2. Given a stream database  $D$  and support threshold  $\theta$ , it initializes the result set  $Q$ , derives the frequent stream alphabet  $\Gamma(D, \theta)$ , constructs the empty query  $\langle \varepsilon \rangle$  and pushes it to the stack, before starting the exploration with function `DISCOVERY`.

---

**Algorithm 2:** BASIC DISCOVERY ALGORITHM

---

**Input:** Stream database  $D$ , support threshold  $\theta$ .

**Output:** Set of matching queries  $Q$ , updated parent dictionary  $P$ .

---

```

1  $Q \leftarrow \emptyset$ ;
2  $\Gamma(D, \theta) = \{a \in \Gamma \mid |\{S \in D \mid \exists j \in \{1, \dots, |S|\}, i \in \{1, \dots, n\} : S[j].A_i = a\}| \geq s(|D|, \theta)\}$ ;
3  $q \leftarrow \langle \varepsilon \rangle$ ;
4  $P(q) \leftarrow \langle \varepsilon \rangle$ ;
5  $O \leftarrow \text{empty stack}$ ;
6  $O.\text{PUSH}(q)$ ;
7  $Q, P \leftarrow \text{DISCOVERY}(D, \theta, \Gamma(D, \theta), Q, P, O)$ ;
8 return  $Q, P$ 

```

---

## 4.2 Stream Insertion and/or Increased Support Threshold

Having discussed a basic discovery algorithm, we now present a first algorithmic solution to incremental processing under changes. Specifically, we consider the case that streams are added and/or that the support threshold is increased.

We summarize the idea behind the algorithm as follows: From Lemma 1, we know that the set of matching queries after the update will be a subset of the original set of matching queries. As such, no new queries need to be generated; we only need to check whether the

---

**Algorithm 3: UPDATEPLUS**

---

**Input:** Stream database  $D_0$ , support threshold  $\theta_0$ , set of matching queries  $Q_0$ , parent dictionary  $P$ , updated stream database  $D_1$ , updated support threshold  $\theta_1$ .  
**Output:** Set of matching queries  $Q_1$ , parent dictionary  $P$ .

```
1  $Q_1 \leftarrow \emptyset$ ;  
2 if  $\theta_1 > \theta_0$  then  $D \leftarrow D_1$ ;  
3 else  $D \leftarrow D_1 \setminus D_0$ ;  
4  $O \leftarrow$  empty stack;  
5  $O.\text{PUSH}(\langle \varepsilon \rangle)$ ;  
  /* Explore queries that are pushed to the stack */  
6 while  $O$  not empty do  
7    $q \leftarrow O.\text{POP}()$ ;  
8   if  $\text{MATCH}(q, D, \theta)$  then  
9     /* For frequent queries, add the query to the result set and push their  
10      child queries onto the stack */  
11      $Q_1 \leftarrow Q_1 \cup \{q\}$ ;  
12      $C \leftarrow \{q' \in \text{dom}(P) \mid P(q') = q\}$ ;  
13     foreach  $q' \in C$  do  $O.\text{PUSH}(q')$ ;  
14 return  $Q_1, P$ 
```

---

queries in the given set are still frequent. Moreover, the scope of this verification depends on the support threshold. If it remains unchanged and only the database changed, it suffices to evaluate the existing queries over the newly added streams. Otherwise, if the threshold is increased as well, the entire updated stream database needs to be considered.

We formalize this approach in Alg. 3. It takes as input the original stream database  $D_0$ ; the original support threshold  $\theta_0$ ; the set of matching queries  $Q_0$  obtained for  $D_0$  and  $\theta_0$ ; the parent dictionary  $P$  for these queries; the updated stream database  $D_1$ ; and the updated support threshold  $\theta_1$ . It returns the set of matching queries  $Q_1$  for  $D_1$  and  $\theta_1$  along with the updated parent dictionary  $P$ .

After initializing the result set  $Q_1$ , the algorithm determines the scope for the verification of the existing queries, as the entire updated stream database  $D_1$  or the added streams  $D_1 \setminus D_0$ . Then, the verification is operationalized with stack onto which we push the queries that shall be checked. We start with the empty query and then go through  $Q_0$  by means of the parent-child-relation between queries, as stored in the parent dictionary.

### 4.3 Stream Deletion and/or Decreased Support Threshold

Next, we turn to incremental event query discovery once streams are deleted from the database and/or the support threshold is decreased. Again, our approach relies on Lemma 1, and exploits that all queries in the existing set of matching queries will also match the updated stream database.

In general, the idea of handling stream deletion and/or a decreased support threshold in an incremental manner can be summarized as follows. We consider the queries, for which the child queries are not frequent. Intuitively, these queries denote the frontier of frequent queries in the space of all query candidates. For these queries, we derive the child queries and add them to the stack of queries to be explored further using the basic discovery procedure. However, this exploration first considers solely the attribute values that have been frequent in the original database. In order to ensure completeness of the set of matching queries, in a second step, we therefore also check whether the frequent alphabet in the updated stream database contains new attribute values. If so, we generate the child queries of all frequent queries found so far, but only incorporate the new frequent attribute values (and no insertion of variables) in the respective generation. All queries generated in this way, had not been considered previously. If one of these queries is frequent, we iteratively continue the exploration with their child queries, this time inserting all frequent attribute values as well as variables. As such, the algorithm is similar to the basic discovery procedure introduced in §4.1, but shows an important difference: It only assesses queries that have been not been checked as part of the previous discovery run over the database before the change.

We formalize the respective algorithm in Alg. 4. The input is the same as discussed for Alg. 3, in terms of the stream databases, support thresholds, parent dictionary and the set of matching queries. The algorithm then initializes the frequent alphabets for the stream databases before and after the change, i.e.,  $\Gamma(D_0, \theta_0)$  and  $\Gamma(D_1, \theta_1)$ . Then, we collect all parents of frequent queries, as a set  $R$ , to be able to iterate over all queries, for which the child queries are not frequent, given by  $Q_1 \setminus R$ . For these queries, the child queries are generated using function `CHILDQUERIES`, considering the frequent alphabet  $\Gamma(D_0, \theta_0)$ , and pushed to a query stack. The latter is used in function `DISCOVERY`, from Alg. 1, to explore the space of candidate queries, starting from the queries on the stack.

As mentioned, we need to consider that an update to the stream database changed the frequent alphabet, i.e., attribute values became frequent due to the removal of streams. If so, i.e., if  $\Gamma(D_1, \theta_1) \setminus \Gamma(D_0, \theta_0)$  is not empty, for each query obtained so far, we need to consider the child queries constructed only through the insertion of these new frequent attribute values (neglecting the insertion of additional variables), which is captured by function `CHILDQUERIESNOVARS`. For these queries, further exploration may be needed, so that they are pushed onto the query stack, which is then used in function `DISCOVERY`.

## 5 Experimental Evaluation

This section presents an evaluation of the algorithms proposed in §4. We first introduce our experimental setup (§5.1), before comparing our algorithms for incremental event query discovery against a baseline solution (§5.2).

---

**Algorithm 4: UPDATEMINUS**

---

**Input:** Stream database  $D_0$ , support threshold  $\theta_0$ , set of matching queries  $Q_0$ , parent dictionary  $P$ , updated stream database  $D_1$ , updated support threshold  $\theta_1$ .  
**Output:** Set of matching queries  $Q_1$ , updated parent dictionary  $P$ .

```
1  $Q_I \leftarrow Q_0$ ;  
2  $s_0 \leftarrow s(|D_0|, \theta_0)$ ;  
3  $\Gamma(D_0, \theta_0) \leftarrow \{a \in \Gamma \mid |\{S \in D_0 \mid \exists j \in \{1, \dots, |S|\}, i \in \{1, \dots, n\} : S[j].A_i = a\}| \geq s_0\}$ ;  
4  $s_1 \leftarrow s(|D_1|, \theta_1)$ ;  
5  $\Gamma(D_1, \theta_1) \leftarrow \{a \in \Gamma \mid |\{S \in D_1 \mid \exists j \in \{1, \dots, |S|\}, i \in \{1, \dots, n\} : S[j].A_i = a\}| \geq s_1\}$ ;  
6  $R \leftarrow \emptyset$ ;  
7  $O \leftarrow \text{empty stack}$ ;  
8 foreach  $q \in Q_I$  do  $R \leftarrow R \cup \{P(q)\}$ ;  
9 foreach  $q \in Q_I \setminus R$  do  
10    $P \leftarrow \text{CHILDQUERIES}(q, \Gamma(D_0, \theta_0), P)$ ;  
11    $C \leftarrow \{q' \in \text{dom}(P) \mid P(q') = q\}$ ;  
12   if  $C \neq \emptyset$  then  
13     foreach  $q' \in C$  do  $O.\text{PUSH}(q')$ ;  
14  $Q_1, P \leftarrow \text{DISCOVERY}(D_1, \theta_1, \Gamma(D_0, \theta_0), Q_I, P, O)$ ;  
15 if  $\Gamma(D_1, \theta_1) \setminus \Gamma(D_0, \theta_0) \neq \emptyset$  then  
16   foreach  $q \in Q_I$  do  
17      $P \leftarrow \text{CHILDQUERIESNOVARS}(q, \Gamma(D_1, \theta_1) \setminus \Gamma(D_0, \theta_0), P)$ ;  
18      $C \leftarrow \{q' \in \text{dom}(P) \mid P(q') = q\}$ ;  
19     if  $C \neq \emptyset$  then  
20       foreach  $q' \in C$  do  $O.\text{PUSH}(q')$ ;  
21    $Q_1, P \leftarrow \text{DISCOVERY}(D_1, \theta_1, \Gamma(D_1, \theta_1), Q_1, P, O)$ ;  
22 return  $Q_1, P$ 
```

---

## 5.1 Experimental Setup

**Datasets.** We use two real-world datasets to evaluate our algorithms, namely NASDAQ stock trade events [EO15] and the Google cluster traces [RWH11]. We adopted the methodology for obtaining stream databases from the [GCW16]. For each dataset, we define three queries to generate streams, as described below.

The attributes considered in the finance data are: 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).

Using these queries, we generated the stream databases summarized in Table 4 by evaluating the queries over the datasets. For each match (of the first at most 1000 matches), we constructed a stream by including the events preceding the match. The number of events taken into account for each stream is detailed in the column 'stream length'.

Tab. 4: Stream databases used in the experiments.

		stream length	number of streams
Finance	F1	50	69-79
	F2	40	990-1000
	F3	25	240-250
Google	G1-G3	7	990-1000

Note that the runtime is not determined by the size of the stream database,  $|D|$ , but rather by the structure of the streams. A small database with intricate and repetitive patterns can trigger a combinatorial explosion of candidate queries, leading to runtimes of minutes or hours. Conversely, a large database with simple, non-repetitive patterns may be processed in a matter of seconds.

**Baseline.** We compare the runtime of our algorithms making use of incremental updates against the baseline approach, in which no prior information is given to the 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).

**Environment and Implementation.** We implemented the whole experimental evaluation in Python. The code and the evaluation routine is publicly available on GitHub<sup>2</sup>. All experiments were conducted on a server with a E7-4880 CPU @2.5GHz and 1TB of RAM.

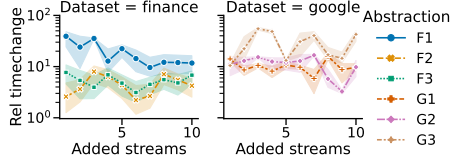
## 5.2 Baseline Comparison

We first conducted experiments on the datasets described in Table 4 by reducing the candidate query space—either by adding streams, increasing the support threshold, or both.

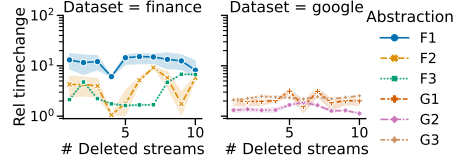
In Figure 4a, we plotted the relative time change of the incremental algorithm compared to the baseline, when adding up to 10 streams to the different stream databases. It can be seen that our incremental algorithm is at least an order of magnitude faster for all settings. For the google dataset G1, the improvement is even up to three orders of magnitude.

<sup>2</sup> <https://github.com/rebesatt/IncDISCES>

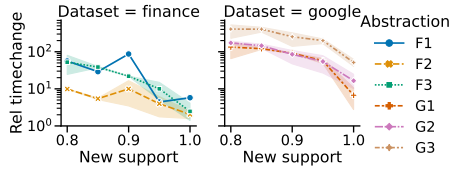




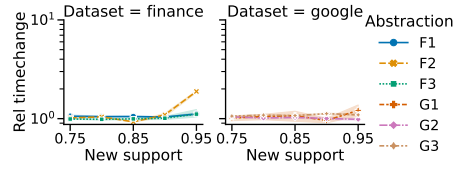
(a) Adding streams to the stream database



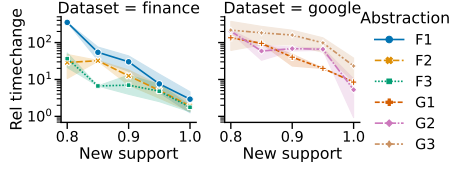
(a) Deleting streams to the stream database



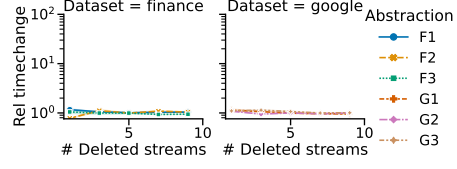
(b) Increasing the support threshold



(b) Decreasing support threshold



(c) Increasing sample size and the support threshold



(c) Decreasing sample size and support threshold

Fig. 4: Relative time change of updated algorithm compared to baseline for different stream databases while decreasing the search space.

Fig. 5: Relative time change of updated algorithm compared to baseline for different stream databases while increasing the search space.

In Figure 4b, we consider the change of increasing the support threshold. The original support threshold is 0.75, which is increased stepwise, up to 1.0. Here, we see a drop in the relative time change for increasing support values. This is especially true for the finance stream databases. One reason for this observation is the decreasing search space. For example, the number of queries for F1 and support threshold 1.0 is four. As consequence, both, the baseline and the incremental algorithm terminate quickly, in less than a second.

In Figure 4c, we examine the simultaneous effect of increasing the support threshold and adding streams. Like in the previous experiment, the original support threshold is 0.75 and is increased stepwise up to 1.0. At the same time, for each step, we add one stream to the stream database. Similar to Figure 4b, we see a drop in the relative time change for increasing support values, but still the runtime of the incremental algorithm is always lower than the baseline solution.

Next, we evaluated the behaviour of the algorithms when increasing the space of candidate queries that need to be searched, by deleting streams from the database and/or by decreasing the support threshold.

In Figure 5a, for each dataset, the incremental algorithm outperforms the baseline solution. For the google stream database G2, the incremental algorithm is twice as fast independently of the number of streams deleted from the original database. The highest benefit is observed for the google dataset G3, where the updated algorithm is up to 50 times faster.

In Figure 5b, we see that for decreasing support threshold for all datasets, the two algorithms need roughly the same runtime to discover the queries. This is because the original setting has a small set of matching queries. As a consequence, the benefit of the incremental discovery algorithms is also small, which yields a similar runtime as the baseline solution.

Similarly, in the experiment shown in Figure 5c, where both the support threshold and the number of streams are decreased, the results follow the same trend. The small set of matching queries in the original setting limits the advantage of the incremental discovery algorithms, leading to comparable runtimes with the baseline approach.

## 6 Related Work

Our work relates to existing approaches for automated query discovery, as well as incremental algorithms for sequential pattern mining and frequent itemset mining.

**Automated Query Discovery** So far research in the area of automated query discovery did not take into account updates in the database or the support threshold. We adopted the query model used in [K123]. But the proposed algorithm does not find all matching queries but only discovers one matching query for each run. The other approaches [GCW16, MCT14] further lack in the query expressiveness. While [MCT14] cannot discover queries with multiple occurrences of an attribute value, [GCW16] lacks the ability to find query terms which only contain variables.

**Incremental Sequential Pattern Mining** A related field in which incremental algorithms have been proposed is sequential pattern mining. Here, the input is a database of strings rather than a stream of events which can contain multiple attributes. Furthermore, the discovery is limited to type level without the ability to discover variables or patterns. In [Zh02, ZXM03], the authors propose two algorithms, one for adding and one for deleting data to the original database. The algorithms make use of negative border sequences which are those sequences whose sub-sequences are frequent to recalculate the results. [MGG13] presents a survey that summarizes the incremental approaches in this area. There the highlighted algorithms are based on common frequent sequence mining algorithms.

**Incremental Frequent Itemset Mining** In frequent itemset mining sets of items are discovered that occur frequently together in a database. This setting differs from our query discovery since the events in a discovered query are order sensitive whereas item sets can occur in arbitrary order. High-utility pattern mining, a sub-field of frequent itemset mining, discovers itemsets based on a user-defined utility threshold. The incremental algorithm introduced in [YR15] proposes a tree structure to update the item sets.

**Sequential Rule Mining** Sequential rule mining [FVNT11, Fo14] is a combination of sequential pattern mining and frequent itemset mining. The goal is to find rules that describe a relationship between itemsets in a sequence. The rules are of the form  $X \rightarrow Y$  where  $X$  and  $Y$  are itemsets. The rules are evaluated based on metrics like support and confidence, and may also be found incrementally [DBF20]. However, one limitation of such rules is that they are bounded to one implication whereas our queries can contain multiple events. Also, the rules are limited in their expressiveness as the itemset captures only type-level information, not allowing for variables.

## 7 Conclusions

In this paper, we focused on the problem of event query discovery in the presence of changes. That is, once all frequent queries have been discovered for a given stream database and a given support threshold, we studied how this set can be updated incrementally if the database changes and/or the support threshold changes, in order to avoid to recompute the respective queries from scratch. We first provided a theoretical analysis of the problem context. Here, we could establish conditions under which the set of matching queries after the change is a subset of the respective set before the change or vice versa. This information is valuable as it highlights when it is sufficient to verify whether the existing queries are still frequent, but there is no need for the generation of additional queries. Based on these insights, we also presented algorithmic solutions that adapt the set of queries incrementally. Our experiments using real-world data demonstrate that our approach to incremental query discovery is significantly faster than a baseline that recomputes the queries, reducing runtime by up to three orders of magnitude. In future work, we intend to increase the efficiency further, e.g., through the design of data structures that capture further details on where in the streams the queries match as well as through the use of statistics about the stream database (e.g., on the distribution of attribute values over streams) and we plan to extend our evaluations by adding alternative performance metrics, such as memory usage.

## Acknowledgments

This work was funded by the German Research Foundation (DFG), CRC 1404: "FONDA: Foundations of Workflows for Large-Scale Scientific Data Analysis".

## Bibliography

- [Ar14] Artikis, Alexander; Weidlich, Matthias; Schnitzler, François; Boutsis, Ioannis; Liebig, Thomas; Piatkowski, Nico; Bockermann, Christian; Morik, Katharina; Kalogeraki, Vana; Marecek, Jakub; Gal, Avigdor; Mannor, Shie; Gunopulos, Dimitrios; Kinane, Dermot: Heterogeneous Stream Processing and Crowdsourcing for Urban Traffic Management. In (Amer-Yahia, Sihem; Christophides, Vassilis; Kementsietsidis, Anastasios; Garofalakis,

Minos N.; Idreos, Stratos; Leroy, Vincent, eds): Proceedings of the 17th International Conference on Extending Database Technology, EDBT 2014, Athens, Greece, March 24-28, 2014. OpenProceedings.org, pp. 712–723, 2014.

- [CA12] Cugola, Gianpaolo; Alessandro: Processing flows of information: From data stream to complex event processing. *ACM Comput. Surv.*, 44(3):15:1–15:62, 2012.
- [Ch10] Chandramouli, Badrish; Ali, Mohamed H.; Goldstein, Jonathan; Sezgin, Beysim; Raman, Balan Sethu: Data Stream Management Systems for Computational Finance. *Computer*, 43(12):45–52, 2010.
- [DBF20] Drozdyuk, Andriy; Buffett, Scott; Fleming, Michael W.: Incremental Sequential Rule Mining with Streaming Input Traces. In (Goutte, Cyril; Zhu, Xiaodan, eds): *Advances in Artificial Intelligence*. Springer International Publishing, Cham, pp. 79–91, 2020.
- [EO15] EODData: NASDAQ Intra-Day Data. <https://eoddata.com/>, 2015. Accessed: 2015-09-27.
- [Fo14] Fournier-Viger, Philippe; Gueniche, Ted; Zida, Souleymane; Tseng, Vincent S.: ERMiner: Sequential Rule Mining Using Equivalence Classes. In (Blockeel, Hendrik; van Leeuwen, Matthijs; Vinciotti, Veronica, eds): *Advances in Intelligent Data Analysis XIII*. Springer International Publishing, Cham, pp. 108–119, 2014.
- [FVNT11] Fournier-Viger, Philippe; Nkambou, Roger; Tseng, Vincent Shin-Mu: RuleGrowth: mining sequential rules common to several sequences by pattern-growth. In: *Proceedings of the 2011 ACM symposium on applied computing*. pp. 956–961, 2011.
- [GCW16] George, Lars; Cadonna, Bruno; Weidlich, Matthias: IL-Miner: Instance-Level Discovery of Complex Event Patterns. *Proc. VLDB Endow.*, 10(1):25–36, September 2016.
- [Gi20] Giatrakos, Nikos; Alevizos, Elias; Artikis, Alexander; Deligiannakis, Antonios; Garofalakis, Minos N.: Complex event recognition in the Big Data era: a survey. *VLDB J.*, 29(1):313–352, 2020.
- [KL19] Konovalenko, Iurii; Ludwig, André: Event processing in supply chain management - The status quo and research outlook. *Comput. Ind.*, 105:229–249, 2019.
- [Kl22] Kleest-Meißner, Sarah; Sattler, Rebecca; Schmid, Markus L.; Schweikardt, Nicole; Weidlich, Matthias: Discovering Event Queries from Traces: Laying Foundations for Subsequence-Queries with Wildcards and Gap-Size Constraints. In (Olteanu, Dan; Vortmeier, Nils, eds): *25th International Conference on Database Theory, ICDT 2022, March 29 to April 1, 2022, Edinburgh, UK (Virtual Conference)*. volume 220 of LIPIcs. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, pp. 18:1–18:21, 2022.
- [Kl23] Kleest-Meißner, Sarah; Sattler, Rebecca; Schmid, Markus L.; Schweikardt, Nicole; Weidlich, Matthias: Discovering Multi-Dimensional Subsequence Queries from Traces - From Theory to Practice. In (König-Ries, Birgitta; Scherzinger, Stefanie; Lehner, Wolfgang; Vossen, Gottfried, eds): *Datenbanksysteme für Business, Technologie und Web (BTW 2023)*, 20. Fachtagung des GI-Fachbereichs „Datenbanken und Informationssysteme“ (DBIS), 06.-10. März 2023, Dresden, Germany, Proceedings. volume P-331 of LNI. Gesellschaft für Informatik e.V., pp. 511–533, 2023.

- [MCT14] Margara, Alessandro; Cugola, Gianpaolo; Tamburrelli, Giordano: Learning from the Past: Automated Rule Generation for Complex Event Processing. In: Proceedings of the 8th ACM International Conference on Distributed Event-Based Systems. DEBS '14, Association for Computing Machinery, New York, NY, USA, p. 47–58, 2014.
- [MGG13] Mallick, Bhawna; Garg, Deepak; Grover, P. S.: Incremental Mining of Sequential Patterns: Progress and Challenges. *Intelligent Data Analysis*, 17(3):507–530, January 2013.
- [Re12] Reiss, Charles; Tumanov, Alexey; Ganger, Gregory R; Katz, Randy H; Kozuch, Michael A: Towards understanding heterogeneous clouds at scale: Google trace analysis. Intel Science and Technology Center for Cloud Computing, Tech. Rep, 84:1–12, 2012.
- [RWH11] Reiss, Charles; Wilkes, John; Hellerstein, Joseph L: Google cluster-usage traces: format+ schema. Google Inc., White Paper, 1:1–14, 2011.
- [YR15] Yun, Unil; Ryang, Heungmo: Incremental High Utility Pattern Mining with Static and Dynamic Databases. *Applied Intelligence*, 42(2):323–352, March 2015.
- [ZDI14] Zhang, Haopeng; Diao, Yanlei; Immerman, Neil: On complexity and optimization of expensive queries in complex event processing. In (Dyreson, Curtis E.; Li, Feifei; Özsu, M. Tamer, eds): International Conference on Management of Data, SIGMOD 2014, Snowbird, UT, USA, June 22–27, 2014. ACM, pp. 217–228, 2014.
- [Zh02] Zheng, Qingguo; Xu, Ke; Ma, Shilong; Lv, Weifeng: The Algorithms of Updating Sequential Patterns, 2002.
- [ZXM03] Zheng, Qingguo; Xu, Ke; Ma, Shilong: When to Update the Sequential Patterns of Stream Data? In (Whang, Kyu-Young; Jeon, Jongwoo; Shim, Kyuseok; Srivastava, Jaideep, eds): Advances in Knowledge Discovery and Data Mining, 7th Pacific-Asia Conference, PAKDD 2003, Seoul, Korea, April 30 - May 2, 2003, Proceedings. volume 2637 of Lecture Notes in Computer Science. Springer, pp. 545–550, 2003.