Discovering Multi-Dimensional Subsequence Queries from Traces – From Theory to Practice

Sarah Kleest-Meißner[1] Rebecca Sattler[2] Markus L. Schmid[3] Nicole Schweikardt[4]
Matthias Weidlich[5]

Abstract: Subsequence-queries with wildcards and gap-size constraints (swg-queries, for short) are an expressive model for sequence data, in which queries are described by patterns over an alphabet of variables and types, along with a global window size and a number of gap-size constraints. They are evaluated over a trace, i.e., a sequence of types, by replacing variables by single types, while satisfying the window and the gap-size constraints. Kleest-Meißner et al. (Proc. ICDT 2022) formalised the task of discovering an swg-query that describes best a given sample consisting of a finite number of traces, and developed a discovery algorithm solving this task. However, in practical application scenarios, traces are often multi-dimensional, i.e., a trace corresponds to a sequence of tuples of types, which renders the existing technique inapplicable.
In this paper, we lift the notion of swg-queries to such a multi-dimensional setting, thereby enlarging the applicability of the query model and the techniques for query discovery. We introduce a mapping between one-dimensional and multi-dimensional sequence data, such that a multi-dimensional trace matches a multi-dimensional query if and only if the corresponding one-dimensional trace matches the corresponding one-dimensional query. We complement our formal results with a description of our prototypical implementation of query discovery for multi-dimensional sequence data. Results from evaluation experiments with real-world data indicate the feasibility of our approach.

Keywords: multi-dimensional subsequence queries on traces, detecting descriptive multi-dimensional queries, subsequences, embeddings

1 Introduction

Models for sequence data define an order for a set of data items [Bab+02], which typically follows from the order in which these items have been created, observed, or received. They

[1] Humboldt-Universität zu Berlin, Unter den Linden 6, D-10099 Berlin, Germany, kleemeis@informatik.hu-berlin.de. Supported by the German Research Foundation (DFG), CRC 1404: "FONDA: Foundation of Workflows for Large-Scale Scientific Data Analysis"

[2] Humboldt-Universität zu Berlin, Unter den Linden 6, D-10099 Berlin, Germany, rebecca.sattler@informatik.hu-berlin.de. Supported by the German Research Foundation (DFG), CRC 1404: "FONDA: Foundation of Workflows for Large-Scale Scientific Data Analysis"

[3] Humboldt-Universität zu Berlin, Unter den Linden 6, D-10099 Berlin, Germany, MLSchmid@MLSchmid.de. Supported by the German Research Foundation (Deutsche Forschungsgemeinschaft, DFG) – project number 416776735 (gefördert durch die Deutsche Forschungsgemeinschaft (DFG) – Projektnummer 416776735)

[4] Humboldt-Universität zu Berlin, Unter den Linden 6, D-10099 Berlin, Germany, schweikn@informatik.hu-berlin.de

[5] Humboldt-Universität zu Berlin, Unter den Linden 6, D-10099 Berlin, Germany, matthias.weidlich@hu-berlin.de

doi:10.18420/BTW2023-24

facilitate the analysis of the evolution of some system over time and have been adopted in various domains. For instance, in cluster monitoring, sequence data describes the process of executing tasks on machines [Ver+15]; in urban transportation, sequence data captures the routes taken by vehicles [Art+14]; and in finance, sequence data represents a transaction history [TRP12].

Sequence data may be queried for relevant patterns by specifying which data items, in which order, and in which temporal context are of interest for a specific analysis question [Gia+20; CM12]. This way, *situations of interest* that happened in the past and materialized in historic data can be detected by evaluating a suitable subsequence query. Employing such a query over a stream of data items also enables the detection of such a situation immediately upon its occurrence, thereby enabling reactive and even pro-active applications.

As an example, consider a cluster monitoring scenario as illustrated in Figure 1. Here, data items indicate transitions in the lifecycle of a task, see Figure 1a. A query may then specify a subsequence of abnormal task execution in the cluster, e.g., as a sequence of data items that indicate that a task was scheduled, killed, and, after being treated in the same way twice (e.g., being updated twice), scheduled again for execution. Here, the respective subsequence is not necessarily continuous and certain lifecycle transitions that are not indicative may occur between the relevant ones. Figure 1b shows how such a query would be written following common languages for complex event recognition [Gia+20].

In practice, finding a suitable query that detects a particular situation is far from trivial, though. Users often know the time at which a situation occurred in the past, but lack insights into the exact materialization of the situation in the sequence data. To provide guidance in the formulation of an adequate query, it was therefore suggested to discover queries that describe patterns linked to the situation of interest [GCW16; MCT14]. These queries may then be interpreted and validated by a user in order to provide traceability and avoid overfitting.

Previously, we proposed a query language for describing subsequence queries with wildcards, a window size, and gap-size constraints [Kle+22], referred to as *swg-queries*. In essence, an swg-query defines a pattern over an alphabet of variables and types, a global window size, and gap-size constraints that bound the number of items that may occur between the queried types and variables. Taking up the query from Figure 1b, the respective swg-query includes: (i) a pattern SCH KIL x x SCH with x denoting a variable; (ii) a global window size of at most 15 items; and (iii) gap-size constraints, e.g., $(0, 10)$ to define that between zero and ten data items may occur between the type KIL and the first occurrence of variable x.

The general concept of subsequences has extensively been studied both in a purely combinatorial sense (in formal language theory, logic and combinatorics on words) and algorithmically (in string algorithms and bioinformatics); see the introductions of the recent papers [Gaw+21; Day+21] for a comprehensive list of relevant pointers. The problem of matching subsequences with gap-constraints (and analysis problems with respect to the set

⟨SUB, SCH, EVI, SCH, KIL, UPD, CHE, UPD, SCH, FIN⟩

(a) Sequence of data items recorded for a task and indicating the task's lifecycle: Submitted (SUB), scheduled (SCH), evicted (EVI), killed (KIL), updated (UPD), checked (CHE), finished (FIN).

```
PATTERN SEQ(Item a, Item b, Item c, Item d, Item e)
WHERE  a.status = e.status = SCH AND b.status = KIL AND c.status = d.status
WITHIN 15 data items
```

(b) A query over sequences of data items following common languages for complex event recognition [Gia+20]. It detects if a task was scheduled, killed, and, after treated in the same way twice (e.g. being updated twice), scheduled again for execution. It matches the sequence in Figure 1a.

Fig. 1: Illustration of a query over sequence data in the domain of cluster monitoring.

of all gap-constrained subsequences of given strings) has been investigated in the recent papers [Day+22; Kos+22a] (see also [Kos+22b] for a survey).

Patterns with variables were introduced by Angluin [Ang80]; they play a central role for inductive inference, in formal language theory and combinatorics on words (see [SA95; MS19; RS97]). Syntactically, our swg-queries are Angluin-style patterns, but adapted in a way that variables refer solely to single types (whereas in Angluin's semantics they refer to finite sequences of types) and matches are further constrained by a global window size and a number of gap-size constraints (such constraints are not available in Angluin's pattern queries).

Despite the fundamental semantic differences between swg-queries and Angluin-style patterns, it is possible to adapt concepts and algorithms from inductive inference of the so-called *pattern languages* that can be described by Angluin-style patterns. Most importantly, the classical concept of *descriptive patterns* (already introduced in [Ang80], see also [FR10; FR13]), can be adapted to swg-queries [Kle+22]: a query q is called *descriptive* for a given sample $\mathcal{S}$ (i.e., a finite set of sequences of data items) and a given support threshold sp if it matches in at least a fraction of sp sequences of $\mathcal{S}$ and there is no strictly more restrictive query q' that also matches in a fraction of at least sp sequences of $\mathcal{S}$.

For classical Angluin-style semantics, *Shinohara's algorithm* [Shi82] computes a descriptive pattern query upon input of a sample $\mathcal{S}$ and the support threshold $\mathsf{sp} = 1$ (see also [Fer+18] for a thorough analysis and extensions of Shinohara's algorithm). In [Kle+22] we presented an adaptation and extension of Shinohara's algorithm that is capable of discovering, upon input of a sample $\mathcal{S}$ and a support threshold $\mathsf{sp} \leqslant 1$, a descriptive swg-query — and different executions of this algorithm may be used to compute a number of different descriptive swg-queries.

However, a major drawback of swg-queries as well as related models of Angluin-style patterns is that they are based on a *one-dimensional* model of sequence data, i.e., data items refer to atomic types. As a consequence, they are not applicable in many practical scenarios in which sequence data comprise items that are instances of a *multi-dimensional* schema.

```
(job=1, task=2, machine=5, status=SCH, priority=low)
(job=4, task=3, machine=5, status=SCH, priority=high)
(job=2, task=1, machine=1, status=UPD, priority=high)
(job=1, task=2, machine=5, status=EVI, priority=low)
(job=1, task=2, machine=3, status=SCH, priority=low)
```

(a) A sequence of multi-dimensional data items. Each data item has five attributes that characterise the job to which a task belongs (job), the identifier of the task (task), the machine to which the task is assigned (machine), the task's lifecycle transition (status), and the execution priority of the task (priority).

```
PATTERN SEQ(Item a, Item b, Item c)
WHERE  a.status = b.status = SCH AND c.status = EVI
AND a.job = c.job AND a.task = c.task AND
AND a.machine = b.machine AND b.priority = high
WITHIN 10 data items
```

(b) A query over sequences of multi-dimensional data items (again, following common languages for complex event recognition [Gia+20]). The query is matched in the sequence depicted in Figure 2a by associating a, b, and c with the first, second, and fourth tuple of the sequence.

Fig. 2: Illustration of a query over multi-dimensional sequence data.

Considering the application domain of cluster monitoring, data items may capture not only the lifecycle transitions related to a task, but also the job to which the task belongs, the assigned machine, and the priority of task execution, see Figure 2a. These attributes enable the definition of more elaborate subsequence queries that incorporate predicates over the respective values of data items. For instance, the query depicted in Figure 2b detects the situation that a task is scheduled on a machine for which, subsequently, the scheduling of a task of a high-priority job on the same machine leads to the eviction of the first task.

In this paper, we address the above limitation by lifting swg-queries to multi-dimensional sequence data. This way, we extend the applicability of the query model and also enable the discovery of descriptive queries from a sample database of multi-dimensional sequences. Our approach is to formulate a suitable mapping between one-dimensional and multi-dimensional sequence data that facilitates query evaluation: A multi-dimensional sequence matches a multi-dimensional query if, and only if, the corresponding one-dimensional sequence matches the corresponding one-dimensional query. We complement these formal contributions with a description of our prototypical implementation of query discovery for multi-dimensional sequence data. We further report on experiments on applying this prototype to a real-world dataset in the domain of cluster monitoring, i.e., the Google Cluster Traces [RWH11]. Our results indicate the general feasibility of discovering swg-queries from multi-dimensional sequence data and also shed light on the sensitivity of the runtime of the approach with respect to structural characteristics of the sequence database.

The rest of this paper is structured as follows. Section 2 introduces the multi-dimensional query model and relates it to the previously studied one-dimensional case. Section 3 provides a solution for the query discovery problem and briefly describes our implementation. Section 4 presents our experimental evaluation. Section 5 concludes the paper.

2 Multi-Dimensional Subsequence-Queries

This section introduces the syntax and semantics of *multi-dimensional subsequence-queries with wildcards and gap-size constraints*, for short: mswg-queries (Section 2.1). After briefly discussing their relation to the (one-dimensional) swg-queries introduced in [Kle+22] (Section 2.2) we present a way to encode mswg-queries as swg-queries (Section 2.3). Prior to this, we fix some basic notation.

Let $\mathbb{N}$ and $\mathbb{N}_{\geqslant 1}$ be the set of non-negative integers and positive integers, respectively. For $\ell \in \mathbb{N}$ we let $[\ell] = \{i \in \mathbb{N} \,:\, 1 \leqslant i \leqslant \ell\}$.

Let A be a non-empty set. We write A^* (and A^+) for the set of all strings (and the set of all non-empty strings) over A. We denote the length of a string s by $|s|$. For a position $i \in [|s|]$ we write $s[i]$ to denote the letter at position i in s. A *factor* of a string $s \in A^*$ is a string $v \in A^*$ such that $s = uvu'$ for $u, u' \in A^*$. A *subsequence* of a string $t = t_1 t_2 \cdots t_n$, where $t_i \in A$ for all $i \in [n]$, is a string $s = s_1 \cdots s_m$ where $m \leqslant n$ and there exist integers $1 \leqslant i_1 < \cdots < i_m \leqslant n$ such that $s_j = t_{i_j}$ for all $j \in [m]$; the mapping $e : [m] \to [n]$ with $e(j) = i_j$ for all $j \in [m]$ is called an *embedding* of s in t. For example, the string `acc` is a subsequence of the string `abaccb` with embedding e where $e(1) \in \{1, 3\}$, $e(2) = 4$ and $e(3) = 5$. We write $s \preccurlyeq_e t$ to indicate that s is a subsequence of t with embedding e, and we suppress the subscript e if we only want to indicate that s is a subsequence of t.

For the rest of this paper we define Γ to be a (finite or infinite) alphabet with $|\Gamma| \geqslant 2$. The elements in Γ will be called *types*. Furthermore, we fix a countably infinite set Vars of *variables*, which is disjoint with the set Γ of types.

2.1 Syntax and Semantics of mswg-queries

We fix a number $k \in \mathbb{N}_{\geqslant 1}$ which we will henceforth call the *dimension*. We model a data item d with k attributes as an ordered k-tuple over Γ, i.e., an element in $\Sigma := (\Gamma^k)$.[6] A *k-dimensional trace over* Γ (for short: *k-trace*) is an element in Σ^+, i.e., a finite non-empty sequence of k-tuples over Γ. The *length* $|t|$ of a k-trace t is the number of k-tuples it comprises — i.e., $|t|$ is t's length as a string over alphabet Σ. We write $types(t)$ for the set of types in Γ that occur in t.

Example 1. *We consider 5-dimensional data items with attributes* job, task, machine, status *and* priority *(which, for a unique tuple representation, are ordered as indicated above), and let* $\Gamma := \{\mathsf{SCH}, \mathsf{UPD}, \mathsf{KIL}\} \cup \{0, 1, \ldots, 10\}$. *For example,* $(1, 2, 5, \mathsf{SCH}, 0)$ *and*

[6] When considering (Γ^k) as an alphabet, i.e., each k-tuple is viewed as a single letter of this alphabet, we use brackets to visualize this.

$(1, 1, 5, \mathsf{KIL}, 0)$ *are two* 5*-dimensional data items (i.e.,* 5*-tuples).*
The following are two examples of 5*-dimensional traces over* Γ*:*

$$s := (1,2,5,\mathsf{SCH},0)\ (1,1,5,\mathsf{KIL},0)\ (1,2,5,\mathsf{SCH},1)$$
$$t := (1,2,5,\mathsf{SCH},0)\ (1,2,4,\mathsf{UPD},0)\ (1,1,5,\mathsf{KIL},0)\ (1,2,5,\mathsf{SCH},1)\ (2,3,2,\mathsf{UPD},1)$$

These traces have length $|s| = 3$ *and* $|t| = 5$*. The mapping* $e : [3] \to [5]$ *with* $e(1) = 1$*,* $e(2) = 3$*, and* $e(3) = 4$ *is an embedding of* s *in* t *and hence witnesses that* s *is a subsequence of* t*. From now on, we will illustrate such an embedding in the following way:*

$$s = \quad (1,2,5,\mathsf{SCH},0) \qquad\qquad (1,1,5,\mathsf{KIL},0)\ (1,2,5,\mathsf{SCH},1)$$
$$t = \quad (1,2,5,\mathsf{SCH},0)\ (1,2,4,\mathsf{UPD},0)\ (1,1,5,\mathsf{KIL},0)\ (1,2,5,\mathsf{SCH},1)\ (2,3,2,\mathsf{UPD},1)$$

Definition 2. A *k-dimensional subsequence-query with wildcards and gap-size constraints* (*k-swg-query*, for short) $q = (s, w, c)$ (over Vars and Γ), consists of a query string $s \in ((\mathsf{Vars} \cup \Gamma)^k)^+$ (i.e., s is a non-empty string of k-tuples built from variables and types), a global window size $w \in \mathbb{N}_{\geqslant 1} \cup \{\infty\}$ with $w \geqslant |s|$, and a tuple of local gap-size constraints $c = (c_1, c_2, \ldots, c_{|s|-1})$, where $c_i = (c_i^-, c_i^+) \in \mathbb{N} \times (\mathbb{N} \cup \{\infty\})$, such that $c_i^- \leqslant c_i^+$ for every $i \in [|s|-1]$ and $|s| + \sum_{i=1}^{|s|-1} c_i^- \leqslant w$.

We speak of *multi-dimensional subsequence-queries* (for short: *mswg-queries*) to refer to k-swg-queries for arbitrary dimension $k \in \mathbb{N}_{\geqslant 1}$. For an mswg-query $q = (s, w, c)$ we write $types(q)$ (or $types(s)$) and $vars(q)$ (or $vars(s)$) to denote the set of types (from Γ) and the set of variables (from Vars), respectively, that occur in q's query string s. Such a query q is called an (ℓ, w, c)*-query* for $\ell := |s|$. We will refer to (ℓ, w, c) as *query parameters*.

The semantics of mswg-queries is defined as follows: Each variable in a query string s serves as a *wildcard* representing an arbitrary type from Γ. A k-swg-query $q = (s, w, c)$ *matches in* a k-trace t (in symbols: $t \models q$), if the wildcards in s can be replaced by types in Γ in such a way that the resulting k-trace s' satisfies the following: t contains a factor t' of length at most w such that s' occurs as a subsequence in t' and for each $i < \ell := |s|$ the gap between $s'[i]$ and $s'[i{+}1]$ in t' has length at least c_i^- and at most c_i^+. I.e., t' is of the form $s'[1]\, g_1\, s'[2]\, g_2 \cdots g_{\ell-1}\, s'[\ell]$ and $c_i^- \leqslant |g_i| \leqslant c_i^+$ for all $i \in [\ell{-}1]$.

An alternative, more formal description of these semantics relies on the following additional notation: An embedding $e : [\ell] \to [n]$ is said to *satisfy a global window size* w, if $e(\ell) - e(1) + 1 \leqslant w$; and we say e *satisfies a tuple* $c = (c_1, c_2, \ldots, c_{\ell-1})$ *of local gap-size constraints* (for ℓ and w), if $c_i^- \leqslant e(i{+}1){-}1 - e(i) \leqslant c_i^+$ for all $i < \ell$. Consider a mapping $\mu : (\mathsf{Vars} \cup \Gamma) \to \Gamma$ with $\mu(a) = a$ for all types $a \in \Gamma$ (such mappings will henceforth be called *substitutions*). We lift μ to a mapping from $(\mathsf{Vars} \cup \Gamma)^k$ to $\Sigma := (\Gamma^k)$ by letting $\mu((a_1, \ldots, a_k)) := (\mu(a_1), \ldots, \mu(a_k))$ for all $(a_1, \ldots, a_k) \in (\mathsf{Vars} \cup \Gamma)^k$; and we further lift μ to a mapping from query strings $s \in ((\mathsf{Vars} \cup \Gamma)^k)^+$ to k-traces of length $\ell := |s|$ by letting $\mu(s) = \mu(s[1]) \cdots \mu(s[\ell])$.

Using these notions, we obtain that a k-trace t *matches* a k-swg-query $q = (s, w, c)$ if, and only if, there exist a substitution $\mu : (\mathsf{Vars} \cup \Gamma) \to \Gamma$ and an embedding $e : [|s|] \to [|t|]$ such that $\mu(s) \preccurlyeq_e t$ and e satisfies w and c. We call (μ, e) *a witness for* $t \models q$.

Example 3. *We consider 5-dimensional data items with attributes* job, task, machine, status, *and* priority *(in this order) and types in* $\Gamma = \{\mathsf{SCH}, \mathsf{EVI}, \mathsf{UPD}, \mathsf{KIL}\} \cup \{0, 1, \ldots, 10\} \cup \{h, l\}$ *(with* h, l *being abbreviations for* `high` *and* `low`*). Consider the query from Figure 2b. This query searches for a subsequence of three 5-dimensional data items, the first two of which have status* SCH *(schedule) and the third of which has status* EVI *(evict) such that the following is true: the first and third data items are related to the same job and to the same task, the first and second data items are related to the same machine, and the second data item has a high priority; all within at most 10 data items.*
This can be expressed as a 5-swg-query $q = (s, w, c)$ *as follows: The query string length is* $\ell := 3$*. The window size is* $w := 10$*. As there are no particular constraints on the gap sizes between the data items, the gap size constraints* c *are chosen to be* $c = ((0, \infty), (0, \infty))$ *(meaning that each gap can be of arbitrary length). The query string* s *is*

$$s \;:=\; (x_j, x_t, x_m, \mathsf{SCH}, y_1)\,(y_2, y_3, x_m, \mathsf{SCH}, h)\,(x_j, x_t, y_4, \mathsf{EVI}, y_5)$$

where $x_j, x_t, x_m, y_1, \ldots, y_5$ *are pairwise distinct variables in* Vars*. The sequence of 5-dimensional data items depicted in Figure 2a corresponds to the 5-trace*

$$t \;:=\; (1, 2, 5, \mathsf{SCH}, l)\,(4, 3, 5, \mathsf{SCH}, h)\,(2, 1, 1, \mathsf{UPD}, h)\,(1, 2, 5, \mathsf{EVI}, l)\,(1, 2, 3, \mathsf{SCH}, l).$$

Observe that $t \models q$*, and a witness substitution* μ *and embedding* e *can be illustrated as:*

$$\begin{array}{ll} s = & (x_j, x_t, x_m, \mathsf{SCH}, y_1)\,(y_2, y_3, x_m, \mathsf{SCH}, h) \qquad\qquad (x_j, x_t, y_4, \mathsf{EVI}, y_5) \\ t = & (1, 2, 5, \mathsf{SCH}, l)\,(4, 3, 5, \mathsf{SCH}, h)\,(2, 1, 1, \mathsf{UPD}, h)\,(1, 2, 5, \mathsf{EVI}, l)\,(1, 2, 3, \mathsf{SCH}, l) \end{array}$$

2.2 One-dimensional swg-queries

For the special case of dimension $k = 1$ we identify k-tuples of elements in $\mathsf{Vars} \cup \Gamma$ with plain elements in $\mathsf{Vars} \cup \Gamma$; i.e., we simply write a instead of (a) for $(a) \in (\mathsf{Vars} \cup \Gamma)^1$. Using this identification, a 1-dimensional trace over Γ precisely corresponds to the notion of *trace over* Γ used in [Kle+22]; and the syntax and semantics of 1-swg-queries precisely coincides with the syntax and semantics of the *swg-queries over* Vars *and* Γ introduced and studied in [Kle+22]. Hence, the notions introduced in Section 2.1 are a natural generalisation of the notions of [Kle+22] from dimension 1 to arbitrary dimension $k \in \mathbb{N}_{\geqslant 1}$. Furthermore, all results achieved in [Kle+22] for swg-queries over Vars and Γ immediately carry over to the 1-swg-queries over Vars and Γ considered in the current paper. A brief survey of [Kle+22] can be found in [Sch22].

The following example illustrates the correspondence between the swg-queries of [Kle+22] and 1-swg-queries.

Example 4. *The sequence of data items depicted in Figure 1a corresponds the the 1-trace*

$$t := \quad (\mathsf{SUB})\ (\mathsf{SCH})\ (\mathsf{EVI})\ (\mathsf{SCH})\ (\mathsf{KIL})\ (\mathsf{UPD})\ (\mathsf{CHE})\ (\mathsf{UPD})\ (\mathsf{SCH})\ (\mathsf{FIN})$$

which, by omitting brackets around 1-tuples, we shortly write as

$$\mathsf{SUB}\ \ \mathsf{SCH}\ \ \mathsf{EVI}\ \ \mathsf{SCH}\ \ \mathsf{KIL}\ \ \mathsf{UPD}\ \ \mathsf{CHE}\ \ \mathsf{UPD}\ \ \mathsf{SCH}\ \ \mathsf{FIN}$$

and this is a trace in the sense of [Kle+22]. The query depicted in Figure 1b searches for a subsequence of data items that indicates that a task was scheduled, killed, and, after treated in the same way twice, scheduled again; all within a global window size of at most 15 data items. This can be expressed as a 1*-swg-query* $q = (s, w, c)$ *as follows: The query string length is* $\ell := 5$*. The window size is* $w := 15$*. As there are no particular constraints on the gap sizes between the data items, the gap size constraints are chosen as* $c := ((0,\infty),(0,\infty),(0,\infty),(0,\infty))$ *(meaning that each gap can be of arbitrary length). The query string is* $s := (\mathsf{SCH})\ (\mathsf{KIL})\ (x)\ (x)\ (\mathsf{SCH})$*, which, by omitting brackets around 1-tuples, is identified with* $\mathsf{SCH}\ \mathsf{KIL}\ x\ x\ \mathsf{SCH}$*, and this exactly yields a swg-query as considered in [Kle+22]. We observe that* $t \models q$*, and a witness substitution* μ *and embedding* e *can be illustrated as follows:*

$s =$		SCH			KIL	x		x	SCH	
$t =$	SUB	SCH	EVI	SCH	KIL	UPD	CHE	UPD	SCH	FIN

2.3 A one-dimensional representation of multi-dimensional traces and queries

This subsection fixes an encoding that allows to represent k-dimensional traces and k-swg-queries over Vars and Γ by corresponding 1-dimensional traces and 1-swg-queries over Vars and a slightly extended type set $\tilde{\Gamma}$. This will allow us to transfer the results obtained in [Kle+22] for the 1-dimensional case to the multi-dimensional setting.

We let $\tilde{\Gamma} := \Gamma \cup \{\#\}$ where # is a new symbol that belongs neither to Γ nor to Vars. We will use # as a separator to mark the beginning of the encoding of every k-dimensional data item. For each k-dimensional data item $d = (a_1, \ldots, a_k) \in \Gamma^k$ we let $enc(d)$ be the 1-dimensional trace over $\tilde{\Gamma}$ of length $k{+}1$ defined as $enc(d) := \# a_1 \cdots a_k$ (recall from Section 2.2 that we omit brackets around 1-dimensional data items, i.e., $enc(d)$ is $(\#)\ (a_1) \cdots (a_k)$).
We lift enc to be a mapping from k-traces over Γ to 1-traces over $\tilde{\Gamma}$ in the canonical way: for a k-trace $t = t_1 t_2 \cdots t_n$ with $t_i \in \Gamma^k$ for all $i \in [n]$ we let $enc(t) := enc(t_1)enc(t_2)\cdots enc(t_n)$. Note that the 1-trace $enc(t)$ has length $(k{+}1)\cdot|t|$.

The following example illustrates how an embedding e of a k-trace s in a k-trace t (witnessing that $s \preccurlyeq t$) can be transferred into an embedding $\tilde{e}$ of the 1-trace $enc(s)$ in the 1-trace $enc(t)$.

Example 5. *Let $k = 2$ and $\Gamma = \{\mathsf{a}, \mathsf{b}, \mathsf{c}\}$. Consider the following 2-traces s and t:*

$s =$		(a, b)		(a, b)	(b, c)		(a, c)
$t =$	(a, b)	(a, b)	(a, c)	(a, b)	(b, c)	(a, b)	(a, c)

Note that $s \preccurlyeq_e t$, witnessed by the embedding e illustrated above. I.e., $e : [4] \to [7]$ with $e(1) = 2$, $e(2) = 4$, $e(3) = 5$, and $e(4) = 7$. The 1-traces $\tilde{s} := enc(s)$ and $\tilde{t} := enc(t)$ are

$\tilde{s} =$				#	a	b				#	a	b	#	b	c				#	a	c
$\tilde{t} =$	#	a	b	#	a	b	#	a	c	#	a	b	#	b	c	#	a	b	#	a	c

Observe that $\tilde{s} \preccurlyeq_{\tilde{e}} \tilde{t}$ by the embedding $\tilde{e}$ illustrated above. This embedding is obtained from e by translating each position of a k-tuple in the k-trace into a block of $k{+}1$ consecutive positions in the corresponding 1-trace.

Note that there also exist other embeddings of $\tilde{s}$ in $\tilde{t}$ that do not correspond to embeddings of s in t; an example is the embedding $\hat{e}$ illustrated as follows:

$\tilde{s} =$	#	a				b				#	a	b	#	b	c				#	a	c
$\tilde{t} =$	#	a	b	#	a	b	#	a	c	#	a	b	#	b	c	#	a	b	#	a	c

The following notion is a straightforward generalization of the way the embedding $\tilde{e}$ was obtained from e in Example 5.

Let $m, n \in \mathbb{N}_{\geqslant 1}$ and let $e : [m] \to [n]$ such that $e(i) < e(j)$ for all $i, j \in [m]$ with $i < j$. Recall that $k \in \mathbb{N}_{\geqslant 1}$ is the fixed *dimension*. We let $rep_k(e)$ be the mapping from $[(k{+}1)m]$ to $[(k{+}1)n]$ defined as follows. We subdivide $[(k{+}1)m]$ into m consecutive blocks of length $(k{+}1)$ each — the i-th block starting at position $(i{-}1)(k{+}1) + 1$ and ending at position $i(k{+}1)$, for every $i \in [m]$. The i-th block of $[(k{+}1)m]$ is mapped by $rep_k(e)$ onto the $e(i)$-th block of $[(k{+}1)n]$. I.e., for all $i \in [m]$ and all $p \in \{1, \ldots, k{+}1\}$ we let

$$rep_k(e)((i{-}1)(k{+}1) + p) \;:=\; (e(i){-}1)(k{+}1) + p.$$

The following lemma provides the property intended by the choice of the definition of $rep_k(e)$; the proof is straightforward and therefore omitted in this paper.

Lemma 6. *Let s and t be two k-traces over Γ and let $\tilde{s} := enc(s)$ and $\tilde{t} := enc(t)$ be the corresponding 1-traces over $\tilde{\Gamma}$. If e is an embedding of s in t (witnessing that $s \preccurlyeq_e t$), then $\tilde{e} := rep_k(e)$ is an embedding of $\tilde{s}$ in $\tilde{t}$ (witnessing that $\tilde{s} \preccurlyeq_{\tilde{e}} \tilde{t}$).*

Next, we focus on how to translate a k-swg-query $q = (s, w, c)$ (over Vars and Γ) into a 1-swg-query $enc(q) = (\tilde{s}, \tilde{w}, \tilde{c})$ over Vars and $\tilde{\Gamma}$ in such a way that for all k-traces t we have: $t \models q \iff enc(t) \models enc(q)$.

The choices of $\tilde{w}$ and $\tilde{s}$ are obvious: We let

$$\tilde{w} := rep_k(w) := \begin{cases} \infty & \text{if } w = \infty \\ (k{+}1)w & \text{otherwise.} \end{cases}$$

The query string $\tilde{s}$ is obtained from s in the analogous way as $enc(t)$ is obtained from t. I.e., every k-tuple $d = (a_1, \ldots, a_k) \in (\mathsf{Vars} \cup \Gamma)^k$ is mapped to $enc(d) = \#a_1 \cdots a_k$, and $s = s_1 \cdots s_\ell$ with $s_i \in (\mathsf{Vars} \cup \Gamma)^k$ for all $i \in [\ell]$ is mapped to $\tilde{s} := enc(s) := enc(s_1) \cdots enc(s_\ell)$.

Note that each position i of s now corresponds to $k{+}1$ consecutive positions in $\tilde{s}$. The gap-size constraints in $\tilde{c}$ are chosen in such a way that they ensure that the k gaps between these positions are of size exactly 0. Consequently, we let

$$\tilde{c} := rep_k(c) = \big(\underbrace{(0,0),\ldots,(0,0)}_{k \text{ times } (0,0)}, \tilde{c}_1, \underbrace{(0,0),\ldots,(0,0)}_{k \text{ times } (0,0)}, \tilde{c}_2, \ldots, \tilde{c}_{\ell-1}, \underbrace{(0,0),\ldots,(0,0)}_{k \text{ times } (0,0)}\big)$$

where for each $i \in [\ell{-}1]$ the component $\tilde{c}_i = (\tilde{c}_i^-, \tilde{c}_i^+)$ is obtained from the i-th component $c_i = (c_i^-, c_i^+)$ of c by letting

$$\tilde{c}_i^- := (k{+}1)c_i^- \qquad \text{and} \qquad \tilde{c}_i^+ := \begin{cases} \infty & \text{if } c_i^+ = \infty \\ (k{+}1)c_i^+ & \text{otherwise.} \end{cases}$$

The following theorem states that the above definitions indeed have the intended functionality.

Theorem 7. *For every k-swg-query q over Vars and Γ and every k-trace t over Γ we have:* $t \models q \iff enc(t) \models enc(q)$.

Proof. We prove the direction "$\Longrightarrow$" (the proof of the opposite direction is analogous).
Let $q = (s, w, c)$ be a k-swg-query over Vars and Γ and let t be a k-trace over Γ. Let $\tilde{q} := enc(q) = (\tilde{s}, \tilde{w}, \tilde{c})$ and $\tilde{t} := enc(t)$ be the corresponding 1-swg-query and 1-trace. Let $\ell := |s|$ and $n := |t|$. Assume that $t \models q$. I.e., there exists a substitution $\mu : (\mathsf{Vars} \cup \Gamma) \to \Gamma$ and an embedding $e : [\ell] \to [n]$ that satisfies w and c, such that $\mu(s) \preccurlyeq_e t$. In other words: (μ, e) is a witness for $t \models q$.
We let $\tilde{e} := rep_k(e)$. And we define $\tilde{\mu} : (\mathsf{Vars} \cup \tilde{\Gamma}) \to \tilde{\Gamma}$ with $\tilde{\mu}(x) := \mu(x)$ for all $x \in \mathsf{Vars}$ and $\tilde{\mu}(a) := a$ for all $a \in \tilde{\Gamma} = \Gamma \cup \{\#\}$. We claim that $(\tilde{\mu}, \tilde{e})$ is a witness for $\tilde{t} \models \tilde{q}$. To prove this, we have to show that $\tilde{\mu}(\tilde{s}) \preccurlyeq_{\tilde{e}} \tilde{t}$ and that $\tilde{e}$ satisfies $\tilde{w}$ and $\tilde{c}$.
Let us start with the first task. By assumption we know that $\mu(s) \preccurlyeq_e t$. From Lemma 6 we obtain that $enc(\mu(s)) \preccurlyeq_{\tilde{e}} \tilde{t}$. Therefore, we are done by noting that $enc(\mu(s)) = \tilde{\mu}(\tilde{s})$.
Let $\tilde{\ell} := |\tilde{s}|$. Let us now verify that $\tilde{e}$ satisfies $\tilde{w}$. In case that $w = \infty$, this is obvious. Let us focus on the case where $w \neq \infty$. By assumption we know that e satisfies w. We have:

$$\begin{aligned} \tilde{e}(\tilde{\ell}) - \tilde{e}(1) + 1 &= e(\ell)(k{+}1) - ((e(1){-}1)(k{+}1) + 1) + 1 \\ &= (k+1) \cdot \big(e(\ell) - e(1) + 1\big) \\ &\leqslant (k+1) \cdot w \;=\; \tilde{w}. \end{aligned}$$

Finally, let us verify that $\tilde{e}$ satisfies $\tilde{c}$. Note that $\tilde{c}$ contains $\ell-1+\ell\cdot k = (k+1)\cdot\ell-1 = \tilde{\ell}-1$ gap-size constraints, where $\ell-1$ constraints correspond to the constraints c_i in $c = (c_1, \ldots, c_{\ell-1})$, which got multiplied by $(k+1)$. The remaining $\ell\cdot k$ constraints in $\tilde{c}$ are equal to $(0,0)$. Our definition of $\tilde{e}$ ensures that the $(0,0)$-constraints are satisfied.
By assumption we know that e satisfies c. Hence, for each $i \in [\ell-1]$ we have: $c_i^- \leqslant g_i \leqslant c_i^+$ for the actual size of the i-th gap $g_i := e(i+1) - 1 - e(i)$. Note that the size of the corresponding gap in the 1-dimensional representation is $(k+1)g_i$. This implies that the corresponding gap-size constraints in $\tilde{c}$ are satisfied.
This completes the proof of the "$\Longrightarrow$"-direction of Theorem 7. □

Theorem 7 serves as a tool to lift results known for the 1-dimensional case to the multi-dimensional setting. In the next section, we implement this for the results on the query discovery problem obtained in [Kle+22].

3 Query Discovery and Implementation

The following notions were introduced in [Kle+22] for the 1-dimensional case and straightforwardly carry over to the multi-dimensional setting.

The *model set* of a k-swg-query q over Vars and Γ is $\mathsf{Mod}_\Gamma(q) := \{\, t \in (\Gamma^k)^+ \,:\, t \models q \,\}$.
A query q' is said to be *strictly more restrictive than* q if $\mathsf{Mod}_\Gamma(q') \subsetneq \mathsf{Mod}_\Gamma(q)$.
A k-dimensional *sample* (over Γ) is a finite, non-empty set $\mathcal{S}$ of k-traces (over Γ). The *support* $\mathsf{supp}(q, \mathcal{S})$ of a k-swg-query query q in a sample $\mathcal{S}$ is defined as the fraction of k-traces in $\mathcal{S}$ that match q, i.e., $\mathsf{supp}(q, \mathcal{S}) := \frac{|\{t \in \mathcal{S} \,:\, t \models q\}|}{|\mathcal{S}|}$.
A *support threshold* is a rational number sp with $0 < \mathsf{sp} \leqslant 1$. A query q is said to *cover* a sample $\mathcal{S}$ *with support* sp if $\mathsf{supp}(q, \mathcal{S}) \geqslant \mathsf{sp}$.

Let us fix the query parameters (ℓ, w, c) and a support threshold sp. Let $\mathcal{S}$ be a sample. A k-swg-query q with parameters (ℓ, w, c) is said to be *descriptive for* $\mathcal{S}$ *w.r.t.* $(\mathsf{sp}, (\ell, w, c))$ if q covers $\mathcal{S}$ with support sp, and there is no k-swg-query q' with parameters (ℓ, w, c) that is strictly more restrictive than q and that still covers $\mathcal{S}$ with support sp. I.e., $\mathsf{supp}(q, \mathcal{S}) \geqslant \mathsf{sp}$ and there is no (ℓ, w, c)-query q' such that $\mathsf{supp}(q', \mathcal{S}) \geqslant \mathsf{sp}$ and $\mathsf{Mod}_\Gamma(q') \subsetneq \mathsf{Mod}_\Gamma(q)$.

The remainder of this section as well as the subsequent Section 4 are devoted to the following query discovery problem for arbitrary dimension $k \in \mathbb{N}_{\geqslant 1}$: The input consists of a support threshold sp, a k-dimensional sample $\mathcal{S}$, and query parameters (ℓ, w, c). The goal is to compute a k-swg-query q with parameters (ℓ, w, c) that is descriptive for $\mathcal{S}$ w.r.t. $(\mathsf{sp}, (\ell, w, c))$. An algorithm solving this discovery problem for the 1-dimensional case (i.e., where $k = 1$) was presented in [Kle+22]. In Section 3.1 we utilize Theorem 7 to lift this algorithm to arbitrary dimension $k \geqslant 1$; Section 3.2 gives a brief description of our implementation of this algorithm.

3.1 An algorithm solving the query discovery problem for arbitrary dimension k

Let $k \in \mathbb{N}_{\geqslant 1}$ be the given dimension and let (ℓ, w, c) be the given query parameters. Note that the most general k-swg-query with parameters (ℓ, w, c) is the query $q_{mg} = (s_{mg}, w, c)$ whose query string s_{mg} is of the form $((x_{1,1}, \ldots, x_{1,k})\,(x_{2,1}, \ldots, x_{2,k}) \cdots (x_{\ell,1}, \ldots, x_{\ell,k}))$ where the $x_{i,j}$ for $i \in [\ell]$ and $j \in [k]$ are $\ell \cdot k$ pairwise distinct variables in Vars. It is straightforward to see that $\mathsf{Mod}_\Gamma(q') \subseteq \mathsf{Mod}_\Gamma(q_{mg})$ for every k-swg-query q' with parameters (ℓ, w, c).

The discovery algorithm takes as input a k-dimensional sample $\mathcal{S}$, a support threshold sp, and the query parameters (ℓ, w, c).
If $\mathsf{supp}(q_{mg}, \mathcal{S}) < \mathsf{sp}$, then there does not exist any k-swg-query q with parameters (ℓ, w, c) with $\mathsf{supp}(q, \mathcal{S}) \geqslant \mathsf{sp}$, let alone a query that is descriptive for $\mathcal{S}$ w.r.t. $(\mathsf{sp}, (\ell, w, c))$. Therefore, the algorithm can safely abort with an error message indicating that the desired query does not exist.
If, on the other hand, $\mathsf{supp}(q_{mg}, \mathcal{S}) \geqslant \mathsf{sp}$, then the algorithm searches for an admissible *replacement operation* for each variable $x \in \mathit{vars}(q_{mg})$. Such an operation replaces x by a symbol y (which can be a type or an *available variable*). It is admissible if the resulting query q' satisfies $\mathsf{supp}(q', \mathcal{S}) \geqslant \mathsf{sp}$. If no replacement operation is possible, we keep x, i.e., the current query string remains unchanged, and x becomes available. After each variable $x \in \mathit{vars}(q_{mg})$ has been considered, the algorithm terminates and produces the current query as output.

Pseudocode implementing this is provided in Algorithm 1. We start by letting q be the most general k-swg-query with parameters (ℓ, w, c) and we let s be the query string of q. If q does not cover $\mathcal{S}$ with support sp, we abort and return the message $\perp$, indicating that there does not exist any k-swg-query with parameters (ℓ, w, c) that is descriptive for $\mathcal{S}$ w.r.t. $(\mathsf{sp}, (\ell, w, c))$. Otherwise, we proceed by letting Δ be the set of types that satisfy the support threshold (these will be the types available for replacement operations), and we initialise the set U of *unvisited* variables to be the set of all variables occurring in q. The set V of *available* variables is initialized to be the empty set. During the main loop in line 5, each variable $x \in U$ is considered exactly once, and for each such variable, the algorithm tests whether there exists a type or variable y from $\Omega := (\Delta \cup V)$ such that replacing all occurrences of variable x by y is an admissible replacement operation.[7] If such an $y \in \Omega$ exists, we perform the actual replacement in line 12. Otherwise, no replacement operation is possible, hence we do not change the current query string but add x to the set V of available variables (line 16).
For dimension $k = 1$, this algorithm was presented in [Kle+22].

Note that the lines 6 and 9 of Algorithm 1 allow to make an arbitrary choice. Different choices lead to different "runs" of the algorithm, and different runs might produce different output queries.

[7] We write $s\langle x \mapsto y \rangle$ to denote the query string obtained from s by replacing every occurrence of the variable x by the symbol y.

ALGORITHM 1: ComputeDescriptiveQuery($\mathcal{S}$,sp, (ℓ, w, c))

Input : k-dim. sample $\mathcal{S}$; support threshold sp with $0 < \mathsf{sp} \leqslant 1$; query parameters (ℓ, w, c)
Returns : descriptive k-swg-query q for $\mathcal{S}$ w.r.t. $(\mathsf{sp}, (\ell, w, c))$ or error message $\perp$

1 $s := s_{mg}$; $q := (s, w, c)$ // query string and query; start with the most general query
2 **if** $\mathsf{supp}(q, \mathcal{S}) < \mathsf{sp}$ **then stop and return** $\perp$
3 $\Delta := \{\gamma \in \Gamma : \frac{|\{t \in \mathcal{S} : \gamma \in types(t)\}|}{|\mathcal{S}|} \geqslant \mathsf{sp}\}$ // types to be considered
4 $U := vars(q)$; $V := \emptyset$ // unvisited variables and available variables
5 **while** $U \neq \emptyset$ **do**
6 **select** an arbitrary $x \in U$ and let $U := U \setminus \{x\}$
7 $\Omega := (\Delta \cup V)$; replace := False // available symbols
8 **while** $\Omega \neq \emptyset$ **do**
9 **select** an arbitrary $y \in \Omega$ and let $\Omega := \Omega \setminus \{y\}$
10 $q' := (s\langle x \mapsto y\rangle, w, c)$
11 **if** $\mathsf{supp}(q', \mathcal{S}) \geqslant \mathsf{sp}$ **then**
12 $s := s\langle x \mapsto y\rangle$ // ReplaceOp
13 replace := True
14 **break** inner loop
15 **if** replace *is* False **then**
16 $V := V \cup \{x\}$ // NoChangeOp
17 **stop and return** $q := (s, w, c)$

Theorem 8. *Let $k \in \mathbb{N}_{\geqslant 1}$, let $\mathcal{S}$ be a k-dimensional sample, let* sp *be a support threshold with $0 < \mathsf{sp} \leqslant 1$, and let (ℓ, w, c) be query parameters.*

(a) If there does not exist any k-swg-query with parameters (ℓ, w, c) that is descriptive for $\mathcal{S}$ w.r.t. $(\mathsf{sp}, (\ell, w, c))$, then there is only one run of Algorithm 1 upon input $\mathcal{S}, \mathsf{sp}, (\ell, w, c)$, and this run stops in line 2 with output $\perp$.

(b) Otherwise, every run of Algorithm 1 upon input $\mathcal{S}, \mathsf{sp}, (\ell, w, c)$ terminates and outputs a query that is descriptive for $\mathcal{S}$ w.r.t. $(\mathsf{sp}, (\ell, w, c))$.

Proof sketch. For the special case where $k = 1$, the theorem was proved in [Kle+22]. Moreover, [Kle+22] provided the following slightly stronger result — again, for the 1-dimensional case: The algorithm obtained from Algorithm 1 by omitting line 1 and, instead, starting with an arbitrary input query q, outputs either a query q' that is descriptive for $\mathcal{S}$ w.r.t. $(\mathsf{sp}, (\ell, w, c))$ and satisfies $\mathsf{Mod}_\Gamma(q') \subseteq \mathsf{Mod}_\Gamma(q)$ or, in case that no such q' exists, the message $\perp$. We use this for the particular 1-dimensional input query $q := enc(q_{mg})$, where q_{mg} is the most general k-swg-query with parameters (ℓ, w, c) for an arbitrary dimension $k \geqslant 2$. Utilizing the one-dimensional representation of k-dimensional traces and queries presented in Section 2.3 and, in particular, Theorem 7, this yields the theorem's statement for arbitrary dimension $k \geqslant 2$. □

The above theorem guarantees that the output produced by Algorithm 1 is correct in the sense that it is either a query that is descriptive for its input or the message $\perp$ indicating that no such query exists. Different runs of the algorithm may produce different queries (each with the guarantee that the delivered query is descriptive for its input). Let us mention, however, that (as shown in [Kle+22]) there exist inputs for which there exist some queries that cannot be delivered by any run of Algorithm 1 but that are descriptive for the input.

We close this subsection with two example runs of Algorithm 1.

Example 9. *For simplicity, the example deals with dimension $k = 1$. Let $\Gamma = \{\mathsf{a}, \mathsf{b}, \mathsf{c}\}$, $\mathcal{S} = \{\mathsf{a}\,\mathsf{b}\,\mathsf{b}, \mathsf{a}\,\mathsf{c}\,\mathsf{c}\}$, $\mathsf{sp} = 1$, $\ell = w = 3$, and $c = ((0,0),(0,0))$. Upon this input, each run of Algorithm 1 lets $s := s_{mg} = x_1x_2x_3$, where x_1, x_2, x_3 are three pairwise distinct variables in Vars. Since $\mathsf{supp}(q, \mathcal{S}) = 1$, the algorithm proceeds by computing $\Delta = \{\mathsf{a}\}$ and letting $U = \{x_1, x_2, x_3\}$ and $V = \emptyset$.*
Let us assume that in the first transition through the outer loop the algorithm selects $x := x_3$. Due to $\Omega = \{\mathsf{a}\}$, the only possible replacement is $s\langle x_3 \mapsto a\rangle$ — but it turns out that this replacement is not admissible as its support on $\mathcal{S}$ is < 1. Hence, x_3 remains unchanged in the query string and is inserted in the set V of available variables, i.e., $V = \{x_3\}$.
Let us assume that in the second transition through the outer loop the algorithm selects $x := x_1$, and in the inner loop it selects $y := \mathsf{a}$. It turns out that the replacement of x_1 by a is admissible (as it has support 1 on $\mathcal{S}$). Hence, s is replaced by the new query string $\mathsf{a}\,x_2x_3$ and V remains unchanged.
In its last iteration through the outer loop, it turns out that $s\langle x_2 \mapsto x_3\rangle$ is the only admissible replacement operation. The algorithm's run terminates after this iteration and outputs the query $q = (s, w, c)$ with $s = \mathsf{a}\,x_3x_3$.
We illustrate this entire run as follows:

$$x_1\ x_2\ \underline{x_3} \overset{\Delta=\{\mathsf{a}\},\ V=\emptyset}{\rightsquigarrow} \underline{x_1}\ x_2\ x_3 \overset{\Delta=\{\mathsf{a}\},\ V=\{x_3\}}{\rightsquigarrow} \mathsf{a}\ \underline{x_2}\ x_3 \overset{\Delta=\{\mathsf{a}\},\ V=\{x_3\}}{\rightsquigarrow} \mathsf{a}\ x_3\ x_3$$

Another run (that outputs a very similar query) is:

$$x_1\ \underline{x_2}\ x_3 \overset{\Delta=\{\mathsf{a}\},\ V=\emptyset}{\rightsquigarrow} \underline{x_1}\ x_2\ x_3 \overset{\Delta=\{\mathsf{a}\},\ V=\{x_2\}}{\rightsquigarrow} \mathsf{a}\ x_2\ \underline{x_3} \overset{\Delta=\{\mathsf{a}\},\ V=\{x_2\}}{\rightsquigarrow} \mathsf{a}\ x_2\ x_2$$

3.2 Implementation

In this section, we describe a prototypical implementation of our approach as a stand-alone Python tool, which is publicly available.[8] We explain how we express multi-dimensional samples and queries. We also briefly discuss how to decide whether $\mathsf{supp}(q, \mathcal{S}) \geqslant \mathsf{sp}$ for given $\mathcal{S}$, sp and q in Algorithm 1. Our implementation is designed for 1-dimensional as well as multi-dimensional queries and samples; in the context of this paper we focus on the multi-dimensional setting.

[8] https://gitlab.com/kleemeis95/sfb-1404-fonda-querydiscovery-prototype

Samples. We model traces as Python strings and use dedicated characters to separate data items as well as data item attributes. Moreover, a (multi-dimensional) sample $\mathcal{S}$ is described by an instance of a specific class (MultidimSample). It combines a list of traces (the actual sample) with meta information, e.g., the sample size (i.e., the number of traces), the data item dimension, and the set of all types occurring in $\mathcal{S}$ (optionally partitioned by the data item attributes and filtered by a given support threshold).

Queries. A query $q = (s, w, c)$ is implemented as an instance of a respective class (MultidimQuery). It encapsulates the query string (a Python string, using different separator symbols), the global window size (an integer), the local gap-size constraints (a list of tuples), and meta information, e.g. the set of types which occur within s, and the set of variables that occur more than once in s.

Matching. The matching routine is implemented as part of the class MultidimQuery. Here, a function match_sample takes as input a MultidimSample-instance $\mathcal{S}$ and a support threshold sp, and outputs true, if $\mathsf{supp}(q, \mathcal{S}) \geqslant \mathsf{sp}$, whereby q denotes the query which is represented by the current class instance. During a run of match_sample, we start by transforming the query string into a regular expression according to the Python *re* module, which takes into account the gap-size constraints c. We then test whether $t \models q$, for each $t \in \mathcal{S}$, by using the function search provided by Python for regular expressions.

Discovery Algorithm. Algorithm 1 is implemented according to the pseudocode presented in Section 3.1. However, we allow the user to influence the arbitrary choices in lines 6 and 9 of the algorithm: one may choose the next, not yet visited variable in the query string either (i) arbitrarily (as described in Section 3.1) or (ii) by scanning the query string from left to right, or vice versa. Furthermore, types (available variables) may be preferred over available variables (types), instead of making an arbitrary choice.

4 Experimental Evaluation

Using the prototype introduced above, we applied our approach to real-world data in the domain of cluster monitoring, a scenario that we already used as a running example. Our goal has been to assess the general feasibility of our approach to the discovery of descriptive queries for multi-dimensional sequence data. We summarize our results as follows:

- We have been able to discover queries from the real-world data, thereby providing evidence that query discovery is indeed feasible. We note though, that the number of discovered queries that cannot immediately be linked to situations of interest is very large, since the given traces consist of many regularities that materialize in the discovered queries.
- Comparing the runtime of our approach for different data samples, it turned out that, in addition to the dataset size, the traversal of the query string and the need to assess variable operations have a major impact.

Below, we first describe our experimental setup in Section 4.1, before we discuss the obtained results in Section 4.2.

4.1 Experimental Setup

Datasets. Our experiments used the Google Cluster Traces [RWH11], a dataset that contains cell information over multiple days. Cells are sets of machines that share a cluster-management system. The machines handle incoming jobs, which consist of at least one task. The dataset contains six types of tables, capturing information about machines, jobs, tasks, constraints, and resources. For a detailed description of the dataset, we refer the reader to [RWH11].

For our experiments, we considered the information on task executions. Specifically, each data item includes the following attributes:

(1) job: The job to which the task belongs. Each job is assigned a unique 64-bit identifier.
(2) task: The task index within a job, given as an integer value.
(3) machine: The machine on which the task shall run. Each machine is assigned a unique 64-bit identifier.
(4) status: The task's status in terms of its lifecycle. It is encoded an integer $i \in \{0, \ldots, 8\}$ that corresponds to one of the following states: SUBMIT (0), SCHEDULE (1), EVICT (2), FAIL (3), FINISH (4), KILL (5), LOST (6), UPDATE_PENDING (7), or UPDATE_RUNNING (8).
(5) priority: The execution priority of the task, modelled as an integer. The larger the number, the higher the priority.

To achieve a controlled setup for query discovery, we employed a setup that is based on three pre-defined queries q_i for $i \in [3]$. The idea being that based on the matches of such a query q_i, we derive a sample of traces $\mathcal{S}_i$, so that our discovery algorithm can be expected to discover query q_i', with $\mathsf{Mod}(q_i') \subseteq \mathsf{Mod}(q_i)$, when using a support value of $\mathsf{sp} = 1.0$. We realised this approach with the following three queries:

```
q1: PATTERN SEQ(Task a, Task b, Task c)
    WHERE   a.status = c.status = 1 AND b.status = 5
    AND     a.job = b.job = c.job AND a.machine = b.machine = c.machine
    WITHIN  1000 data items
q2: PATTERN SEQ(Task a, Task b, Task c, Task d)
    WHERE   a.status = b.status = c.status = d.status = 4
    AND   a.machine = b.machine = c.machine = d.machine
    WITHIN  100 data items
q3: PATTERN SEQ(Task a, Task b, Task c)
    WHERE   a.machine = b.machine AND a.job=c.job
    AND a.status = b.status = 1 AND c.status=2
    WITHIN  100 data items
```

Based on the matches of these queries, we constructed the samples $\mathcal{S}_i$, $i \in [3]$, as follows: For samples $\mathcal{S}_1$ and $\mathcal{S}_2$, the end of a trace was determined by the last item of a match. The

start of a trace was defined as the item following the last item of the *previous* match (or the first item of the whole dataset, respectively), so that traces are non-overlapping, i.e., any data item appears in at most one of the traces of a sample. Moreover, the traces for samples $\mathcal{S}_1$ and $\mathcal{S}_2$ were partitioned by the `machine` attribute, as the respective queries refer solely to items within such a partition. Finally, all traces with at most 100 items were included in the sample, which selected more than 98% ($\mathcal{S}_1$) or 91% ($\mathcal{S}_2$) of the constructed traces. For sample $\mathcal{S}_3$, again, the end of a trace was determined by the last item of a match. The start of a trace, however, was defined to be the first item of a match. The constructed samples had the following characteristics:

Sample	Size	Min. trace length	Max. trace length
$\mathcal{S}_1$	558	3	96
$\mathcal{S}_2$	679	22	99
$\mathcal{S}_3$	84	4	197

Experimental Procedure and Measures. For each sample, Algorithm 1 was executed using the most general query, setting its parameters (ℓ, w, c) as follows. For the length of the query string, we set $\ell = 3$ for $\mathcal{S}_1$ and $\mathcal{S}_3$, and $\ell = 4$ for $\mathcal{S}_2$. The global window size was set to $w = 100$ for $\mathcal{S}_1$ and $\mathcal{S}_2$, and to $w = 200$ for $\mathcal{S}_3$, as derived from the sample generation procedure. We defined the local gap-size constraints to be least restrictive by setting them to $c = ((0, 100), (0, 100))$ for $\ell = 3$, and to $c = ((0, 200), (0, 200), (0, 200))$ for $\ell = 4$, respectively. We considered several support thresholds (namely, 0.6, 0.8, and 1.0).

Concerning the choices in Algorithm 1 on the next unvisited variable and replacement preference, we consider all options outlined in Section 3.2: the next variable is derived left-to-right (l2r), right-to-left (r2l), or arbitrarily (a). This choice is combined with no preference (a), or preference given to types (t) or variables (v).

In addition to illustrating some of the discovered queries, we measure the runtime of our approach for different instantiations. We further break down the runtime by profiling the our implementation with Python's cProfile to shed light on the contribution of the various algorithmic steps.

Experimental Environment. All experiments have been executed on a 64 Bit Manjaro system with an AMD Ryzen 5 Pro 5650U processor running at 4.1 GHz and 16GB RAM.

4.2 Evaluation

Discovered queries. Applying our approach for each sample $\mathcal{S}_i$, $i \in [3]$, we successfully discovered multi-dimensional swg-queries. Moreover, the discovered queries turned out to

be more specific than the queries used to create the samples (see Section 4.1). For instance, we discovered the following query strings for sp = 1.0 for the three samples:

$$\mathcal{S}_1: \quad s_1 = (x_j, x_t, x_m, 1, x_p)(x_j, x_t, x_m, 5, x_p)(x_j, y_1, x_m, 1, y_2)$$
$$\mathcal{S}_2: \quad s_2 = (y_1, y_2, x_m, 4, x_p)(y_3, y_4, x_m, 4, x_p)(y_5, y_6, x_m, 4, x_p)(y_7, y_8, x_m, 4, x_p)$$
$$\mathcal{S}_3: \quad s_3 = (x_j, y_1, x_m, 1, x_p)(y_2, y_3, x_m, 1, y_4)(x_j, y_5, y_6, 2, x_p)$$

whereby $x_j, x_t, x_m, x_p, y_1, \ldots, y_8$ are pairwise distinct variables in Vars. We note that these query strings can be used to derive queries in common languages for complex event recognition. Taking s_1 as an example, we derive the following query q_1':

```
q1':  PATTERN SEQ(Task a, Task b, Task c)
      WHERE  a.status = c.status = 1 AND b.status = 5
      AND    a.job = b.job = c.job AND a.machine = b.machine = c.machine
      AND    a.task = b.task AND a.priority = b.priority
      WITHIN 1000 data items
```

Comparing query q_1' with query q_1 from Section 4.1, we observe that the last line of the WHERE-clause renders the discovered query more specific. Similar observations are done for the descriptive queries discovered for the other samples. For instance, the above query strings s_2 and s_3 enforce conditions on the `priority` attribute that have not been part of the queries used to generate the samples.

Our results indicate that it is feasible to discover multi-dimensional swg-queries from real-world data. However, we also note that we discovered descriptive queries that would not match the last item of a trace, i.e., the supposed situation of interest. This highlights that the discovered queries will still have to be assessed by domain experts.

Runtime. Figure 3-5 provide the results for our runtime measurements in seconds, when varying the support threshold and the configuration of Algorithm 1 for selecting the next unvisited variable and the replacement preference. Overall, a higher support threshold yields smaller runtimes. This is due to a smaller number of supported types that have to be tested as well as the fact that the match test stops as soon as the support threshold cannot be satisfied any more. Moreover, the fastest runs have in common that the algorithm goes through the query string from left to right (l2r), while there is no clear trend for the replacement preference. However, this result highlights the potential for improvements based on heuristics to guide the exploration of the search space.

Performance profiling. We illustrate the results of the performance profile for $\mathcal{S}_3$ in Figure 6, using a SnakeViz' icicle plot [sna]. Here, each rectangle represents a function, while the layering of rectangles captures the call hierarchy between functions (the top function is called first). Moreover, each function is annotated with its overall runtime, which is visualized by the width of the rectangle.

We observe that the runtime of the main function of our experiments is dominated by the function compute_descr_swgquery, which implements Algorithm 1. It has to decide whether $\mathsf{supp}(q, \mathcal{S}) \geqslant \mathsf{sp}$ multiple times, i.e., for each unvisited variable x the algorithm

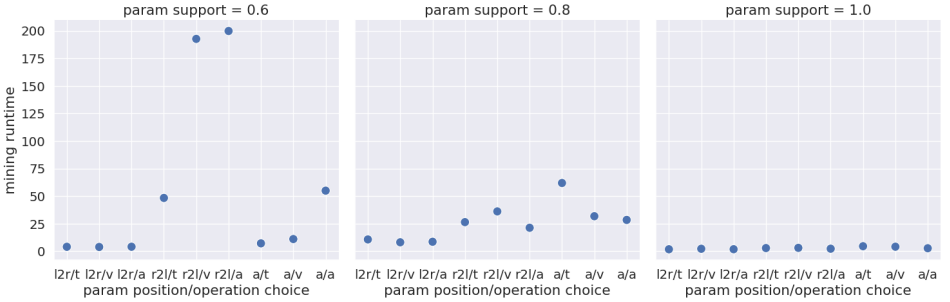

Fig. 3: Runtime measurements for sample $\mathcal{S}_1$.

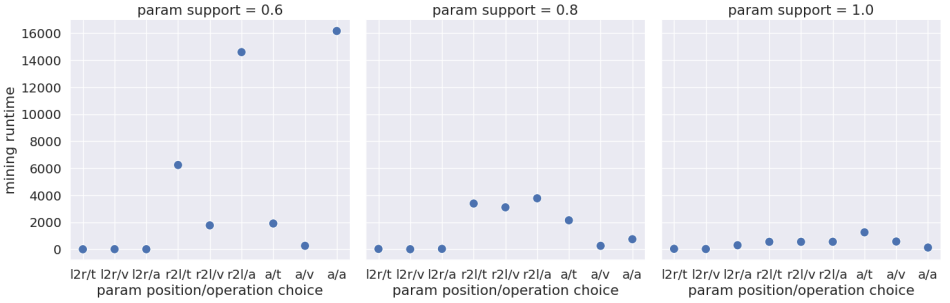

Fig. 4: Runtime measurements for sample $\mathcal{S}_2$.

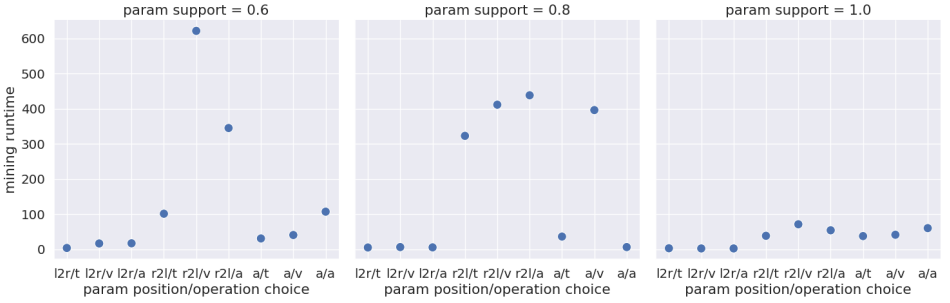

Fig. 5: Runtime measurements for sample $\mathcal{S}_3$.

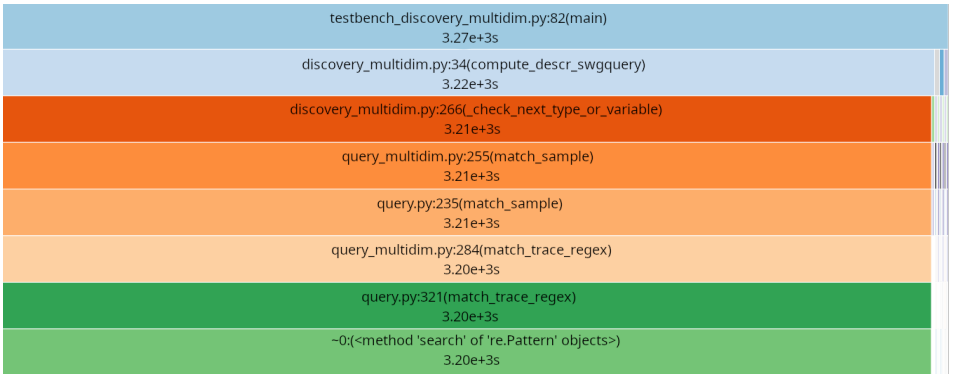

Fig. 6: Icicle visualisation of sample $\mathcal{S}_3$'s profile.

tests, whether there exists a type or variable so that the replacement yields a query satisfying the support threshold. This check is realised by function _check_next_type_or_variable. After building the regular expression corresponding to the current query, the matching problem is solved. Figure 6 illustrates that this function search of re.Pattern dominates the overall runtime. We conclude that performance improvements for query discovery may be achieved through optimizations of the function to decide whether a trace matches a query. In future work, we strive for algorithmic optimizations that exploit the fact that many subsequent match tests are conducted over the same set of traces with only slightly changed queries.

5 Concluding Remarks

Motivated by sequence data over a multi-dimensional schema, we defined an encoding to lift swg-queries and corresponding concepts as described in [Kle+22] to a multi-dimensional setting (Section 2). Furthermore we described our prototypical implementation of query discovery for multi-dimensional data (Section 3) and discussed experiments and their results on a real-world dataset (Section 4).

Our experiments' main result can be summarised as general feasibility of discovering swg-queries from multi-dimensional data: for each data set and configuration of Algorithm 1 we discovered a bunch of descriptive queries. This query set includes queries which are more specific than the queries we used for the data set generation.

Furthermore, our experiments suggest that structural characteristics of the sequence data plays an important role regarding the runtime of our discovery algorithm at various levels. Firstly, we observed notable differences in runtime regarding the way we select the next unvisited variable and its possible replacement. Hence we are interested in finding heuristics to predict which combination is most promising. Besides, we might be able to exploit the facts that q' does not change while testing $\mathsf{supp}(q', \mathcal{S}) \geqslant \mathsf{sp}$ once and that $\mathcal{S}$ does not change during the entire run of the discovery algorithm.

References

[Ang80] Dana Angluin. "Inductive Inference of Formal Languages from Positive Data". In: *Inf. Control.* 45.2 (1980), pp. 117–135. DOI: 10.1016/S0019-9958(80)90285-5.

[Art+14] Alexander Artikis et al. "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*. OpenProceedings.org, 2014, pp. 712–723. DOI: 10.5441/002/edbt.2014.77.

[Bab+02] Brian Babcock, Shivnath Babu, Mayur Datar, Rajeev Motwani, and Jennifer Widom. "Models and Issues in Data Stream Systems". In: *Proceedings of the Twenty-first ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems, June 3-5, Madison, Wisconsin, USA*. ACM, 2002, pp. 1–16. DOI: 10.1145/543613.543615.

[CM12] Gianpaolo Cugola and Alessandro Margara. "Processing flows of information: From data stream to complex event processing". In: *ACM Comput. Surv.* 44.3 (2012), 15:1–15:62. DOI: 10.1145/2187671.2187677.

[Day+21] Joel D. Day, Pamela Fleischmann, Maria Kosche, Tore Koß, Florin Manea, and Stefan Siemer. "The Edit Distance to k-Subsequence Universality". In: *38th International Symposium on Theoretical Aspects of Computer Science, STACS 2021, March 16-19, 2021, Saarbrücken, Germany (Virtual Conference)*. 2021, 25:1–25:19. DOI: 10.4230/LIPIcs.STACS.2021.25.

[Day+22] Joel D. Day, Maria Kosche, Florin Manea, and Markus L. Schmid. "Subsequences With Gap Constraints: Complexity Bounds for Matching and Analysis Problems". In: vol. abs/2206.13896. 2022. DOI: 10.48550/arXiv.2206.13896. arXiv: 2206.13896.

[Fer+18] Henning Fernau, Florin Manea, Robert Mercas, and Markus L. Schmid. "Revisiting Shinohara's algorithm for computing descriptive patterns". In: *Theor. Comput. Sci.* 733 (2018), pp. 44–54. DOI: 10.1016/j.tcs.2018.04.035.

[FR10] Dominik D. Freydenberger and Daniel Reidenbach. "Existence and nonexistence of descriptive patterns". In: *Theor. Comput. Sci.* 411.34-36 (2010), pp. 3274–3286. DOI: 10.1016/j.tcs.2010.05.033.

[FR13] Dominik D. Freydenberger and Daniel Reidenbach. "Inferring descriptive generalisations of formal languages". In: *J. Comput. Syst. Sci.* 79.5 (2013), pp. 622–639. DOI: 10.1016/j.jcss.2012.10.001.

[Gaw+21] Pawel Gawrychowski, Maria Kosche, Tore Koß, Florin Manea, and Stefan Siemer. "Efficiently Testing Simon's Congruence". In: *38th International Symposium on Theoretical Aspects of Computer Science, STACS 2021, March 16-19, 2021, Saarbrücken, Germany (Virtual Conference)*. 2021, 34:1–34:18. DOI: 10.4230/LIPIcs.STACS.2021.34.

[GCW16] Lars George, Bruno Cadonna, and Matthias Weidlich. "IL-Miner: Instance-Level Discovery of Complex Event Patterns". In: *Proc. VLDB Endow.* 10.1 (2016), pp. 25–36. DOI: 10.14778/3015270.3015273.

[Gia+20] Nikos Giatrakos, Elias Alevizos, Alexander Artikis, Antonios Deligiannakis, and Minos N. Garofalakis. "Complex event recognition in the Big Data era: a survey". In: *VLDB J.* 29.1 (2020), pp. 313–352. DOI: 10.1007/s00778-019-00557-w.

[Kle+22] Sarah Kleest-Meißner, Rebecca Sattler, Markus L. Schmid, Nicole Schweikardt, and Matthias Weidlich. "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*. Vol. 220. LIPIcs. 2022, 18:1–18:21. DOI: 10.4230/LIPIcs.ICDT.2022.18.

[Kos+22a] Maria Kosche, Tore Koß, Florin Manea, and Viktoriya Pak. "Subsequences in Bounded Ranges: Matching and Analysis Problems". In: *CoRR* abs/2207.09201 (2022). DOI: 10.48550/arXiv.2207.09201. arXiv: 2207.09201.

[Kos+22b] Maria Kosche, Tore Koß, Florin Manea, and Stefan Siemer. "Combinatorial Algorithms for Subsequence Matching: A Survey". In: *CoRR* abs/2208.14722 (2022). DOI: 10.48550/arXiv.2208.14722. arXiv: 2208.14722.

[MCT14] Alessandro Margara, Gianpaolo Cugola, and Giordano Tamburrelli. "Learning from the past: automated rule generation for complex event processing". In: *The 8th ACM International Conference on Distributed Event-Based Systems, DEBS '14, Mumbai, India, May 26-29, 2014*. ACM, 2014, pp. 47–58. DOI: 10.1145/2611286.2611289.

[MS19] Florin Manea and Markus L. Schmid. "Matching Patterns with Variables". In: *Combinatorics on Words - 12th International Conference, WORDS 2019, Loughborough, UK, September 9-13, 2019, Proceedings*. 2019, pp. 1–27. DOI: 10.1007/978-3-030-28796-2_1.

[RS97] Grzegorz Rozenberg and Arto Salomaa. "Patterns". In: *Handbook of Formal Languages*. Vol. 1. Springer, 1997, pp. 230–242.

[RWH11] Charles Reiss, John Wilkes, and Joseph L. Hellerstein. "Google cluster-usage traces: format+ schema". In: *Google Inc., White Paper* (2011), pp. 1–14.

[SA95] Takeshi Shinohara and Setsuo Arikawa. "Pattern Inference". In: *Algorithmic Learning for Knowledge-Based Systems, GOSLER Final Report*. 1995, pp. 259–291. DOI: 10.1007/3-540-60217-8_13.

[Sch22] Markus L. Schmid. "Extending Shinohara's Algorithm for Computing Descriptive (Angluin-Style) Patterns to Subsequence Patterns". In: *CoRR* abs/2206.13918 (2022). DOI: 10.48550/arXiv.2206.13918. arXiv: 2206.13918.

[Shi82] Takeshi Shinohara. "Polynomial Time Inference of Pattern Languages and Its Application". In: *Proceedings of the 7th IBM Symposium on Mathematical Foundations of Computer Science, MFCS*. 1982, pp. 191–209.

[sna] snakeviz. *SnakeViz documentation*. URL: https://jiffyclub.github.io/snakeviz/.

[TRP12] Kia Teymourian, Malte Rohde, and Adrian Paschke. "Knowledge-based processing of complex stock market events". In: *15th International Conference on Extending Database Technology, EDBT '12, Berlin, Germany, March 27-30, 2012, Proceedings*. ACM, 2012, pp. 594–597. DOI: 10.1145/2247596.2247674.

[Ver+15] Abhishek Verma, Luis Pedrosa, Madhukar Korupolu, David Oppenheimer, Eric Tune, and John Wilkes. "Large-scale cluster management at Google with Borg". In: *Proceedings of the Tenth European Conference on Computer Systems, EuroSys 2015, Bordeaux, France, April 21-24, 2015*. ACM, 2015, 18:1–18:17. DOI: 10.1145/2741948.2741964.