Elena Vasilyeva, Maik Thiele, Christof Bornhövd, Wolfgang Lehner

# Answering “Why Empty?” and “Why So Many?” queries in graph databases

**Erstveröffentlichung in / First published in:**

*Journal of Computer and System Sciences*. 2015. 82(1), S. 3-22. Elsevier. ISSN 0022-0000.

DOI: https://doi.org/10.1016/j.jcss.2015.06.007

Final edited form was published in "Journal of Computer and System Sciences". 82 (1), S. 3-22. ISSN: 0022-0000.
https://doi.org/10.1016/j.jcss.2015.06.007

# Answering "Why Empty?" and "Why So Many?" queries in graph databases

Elena Vasilyeva [a,*], Maik Thiele [b], Christof Bornhövd [c], Wolfgang Lehner [b]

[a] *SAP SE, Dresden, Germany*
[b] *Database Technology Group, Technische Universität Dresden, Germany*
[c] *SAP Labs, LLC, Palo Alto, USA*

ARTICLE INFO

*Article history:*
Received 30 October 2014
Received in revised form 27 May 2015
Accepted 15 June 2015
Available online 8 July 2015

*Keywords:*
Graph databases
"Why?" query
"Why Empty?" query
"Why So Few?" query
"Why Not?" query
"Why So Many?" query

ABSTRACT

Graph databases provide schema-flexible storage and support complex, expressive queries. However, the flexibility and expressiveness in these queries come at additional costs: queries can result in unexpected empty answers or too many answers, which are difficult to resolve manually. To address this, we introduce subgraph-based solutions for graph queries "Why Empty?" and "Why So Many?" that give an answer about which part of a graph query is responsible for an unexpected result. We also extend our solutions to consider the specifics of the used graph model and to increase efficiency and experimentally evaluate them in an in-memory column database.

## 1. Introduction

New kinds of data and their analysis increase the demand for flexible data models supporting data of different degrees of structure. Graph databases implementing the property graph model [1] are a reasonable answer to this demand, because they support data with highly irregular structure in the form of a graph. A diverse schema for vertices and edges is represented by an arbitrary number of attributes, which can differ between vertices or edges of the same semantic type. A major advantage is that such systems do not require a predefined rigid database schema. In graph databases based on the property graph model, a query can be understood as a pattern that has to be sought in a large data graph.

However, the flexibility provided by graph databases and the property graph model comes at additional costs. Users of graph databases typically have only limited knowledge about the stored data, which complicates the creation of queries. They can overspecify or underspecify a query and, as a consequence, they can get unexpected result sets that can be empty, include too few or too many answers, or miss some subgraphs of interest. Any unexpected answer can cause confusion on the user side, since its reason is unclear: was the query overspecified/underspecified or was it correct and is the data not existing in the database? To answer these questions, users need a means for explorative queries and guidance through the query answering process. To allow this, a graph query processing engine has to be able to give intermediate results of query processing, which describe the already discovered and the still not processed parts of a query graph. As a result, users can discover overspecified or underspecified query parts or conclude that the query was correct.

* Corresponding author.
*E-mail address:* elena.vasilyeva@sap.com (E. Vasilyeva).

http://dx.doi.org/10.1016/j.jcss.2015.06.007

Final edited form was published in "Journal of Computer and System Sciences". 82 (1), S. 3-22. ISSN: 0022-0000.
https://doi.org/10.1016/j.jcss.2015.06.007

### 1.1. Contributions

In this extended version of our previous work on the empty-answer problem in graph databases [2] we describe our contributions from the prior work on a subgraph-based solution with an all-covering tree and several optimizations in the form of *"Why Empty?" queries*, i.e., differential queries, that determine existing and missing query parts, i.e. which query parts were discovered in a data graph and which are missing. For this, we (1) discover *maximum common subgraphs* in a data graph for a given query, and (2) calculate *differences* between them and a query graph. As a result, the graph processing engine yields a list of discovered maximum common subgraphs and undiscovered parts of a query graph.

In addition to our previous work [2], we *extend the paper scope to new answer types*: too many and too few answers, and missing results. We *classify which explanations* for these problems can be generated by analyzing solutions presented in relational database management systems. We *provide modification operations* (graph edit operations extended with new operations suitable for property graphs) to be used to (1) qualify how much information is missing in a data graph to answer a query for an empty-answer problem, (2) qualify how much information has to be removed from a query graph to deliver less results, or (3) rewrite a query with the purpose to deliver expected results. We also *propose a new subgraph-based solution* for an extreme of the classified unexpected answers: *for too many answers*. For this purpose, we introduce and evaluate *"Why So Many?" queries*, a new kind of graph queries, that show which vertices and edges can potentially be omitted from the query processing to reduce the number of answers. The response to a "Why So Many?" query includes (1) *data subgraphs corresponding to the expected size* of a result set, and (2) *corresponding differential graphs*, which make the result set exceed this expected cardinality. As a result, the graph processing engine yields a list of discovered cardinality-bounded data subgraphs and differential graphs, including elements that increase the cardinality of answers. With our *extended evaluation* for "Why Empty?" and "Why So Many?" queries in the in-memory column store GRATIN [3] we show that the construction of all-covering spanning trees allows to discover larger subgraphs and optimization techniques significantly reduce the response time for both query types.

The rest of the paper is structured as follows. First, we introduce graph databases with the property graph model in Section 2. Afterwards, we classify unexpected answers and corresponding "Why?" queries with their solutions in Section 3. Then, in Sections 4 and 5 we present "Why Empty?" and "Why So Many?" queries, respectively. In Section 6 we propose optimization strategies. We evaluate both query types in Section 7 and compare our solution with the state of the art in Section 8.

## 2. Graph database and underlying graph model

As an underlying data model we use the property graph model [1]. It represents a graph as a directed multigraph, where vertices are entities and edges are relationships between them.

**Definition 1** *(Property graph).* We define a **property graph** as a directed graph $G = (V, E, u, f, g, A_V, A_E)$ over attribute space $A = A_V \dot{\cup} A_E$, where: (1) $V, E$ are finite sets of vertices and edges; (2) $u : E \to V^2$; (3) $f : V \to A_V$ and $g : E \to A_E$ are attribute functions for vertices and edges; and (4) $A_V$ and $A_E$ are their attribute space.

**Definition 2** *(Path).* A **path** of a property graph $G$ is a directed property graph with distinct vertices $v_0, \ldots, v_n \in V$ and edges $e_0, \ldots, e_{n-1} \in E$ such that $e_i$ is an edge directed from $v_i$ to $v_{i+k} \forall i < k$.

**Definition 3** *(Connected graph).* A non-empty graph $G = (V, E, u, f, g, A_V, A_E)$ is connected, if $\forall v_i, v_j \in V$, where $i \neq j$ are linked by a path in $G$.

**Definition 4** *(Connected subgraph).* A non-empty graph $G' = (V', E', u', f', g', A_{V'}, A_{E'})$ is a connected subgraph of $G = (V, E, u, f, g, A_V, A_E)$, if $G'$ is connected, $V' \subseteq V, E' \subseteq E, u' = u\,|_{E'}, f' = f\,|_{V'}$, and $g' = g\,|_{E'}$.

Depth-first search can be applied to check, whether a property graph is connected. Since the complexity of a depth-first search is $O(k)$, with $k = m + n$, where $m$ is a number of edges and $n$ is a number of vertices this also holds for the computation of a connected subgraph.

**Definition 5** *(Common connected subgraph).* Given a data graph $G_d$ and a query graph $G_q$, the graph $G'_d = (V'_d, E'_d, u'_d, f'_d, g'_d, A'_{V'_d}, A'_{E'_d})$ is a common connected subgraph of graphs $G_d$ and $G_q$, if $G'_d$ is a connected subgraph of $G_d$ and $G'_d$ is a connected subgraph of $G_q$.

There may be multiple common connected subgraphs in a data graph $G_d$ for a query graph $G_q$.

In our prototypical system the property graph model is implemented as a graph-specific extension within a RDBMS, which uses column groups for vertices and edges. Vertices are described by a set of columns for their attributes, and edges are stored as simplified adjacency lists in a table. Each edge and vertex can have multiple attributes, which are stored together with their unique identifiers.

Final edited form was published in "Journal of Computer and System Sciences". 82 (1), S. 3-22. ISSN: 0022-0000.
https://doi.org/10.1016/j.jcss.2015.06.007

**Table 1**
Classification of "Why?" queries and their solutions.

| Query type | Problem | Received cardinality | Explanation | |
|---|---|---|---|---|
| | | | Subgraph-based | Query rewriting |
| Why Empty? | Empty result | $C_{recv} = 0$ | Maximum discovered subgraphs and differential graphs. | Answers to modified query delivering at least one result. |
| Why So Few? | Too few results | $C_{recv} < C_{target}$ | Maximum discovered subgraphs matching cardinality and differential graphs. | Answers to modified query delivering more results. |
| Why Not? | Missing items | – | Maximum subgraphs discovered from interesting elements and differential graphs. | Answers including subgraphs of interest and original results. |
| Why So Many? | Too many results | $C_{recv} > C_{target}$ | Maximum discovered subgraphs matching cardinality and differential graphs. | Answers to modified query delivering less results. |

To process such a graph efficiently, we use the in-memory column database GRATIN [3], which supports optimized flexible tables (new attributes can efficiently be added and removed) and provides advanced compression techniques for sparsely populated columns like in [4,5,3]. This abstraction allows us to store graphs with an arbitrary number of attributes without a predefined rigid schema. The graph database provides the following operations: insert, delete, update, filter based on attribute values, aggregation, and graph traversal in a breadth-first manner. Traversal along directed edges is supported for both directions with the same performance.

Queries to the database are represented by property graphs consisting of edges and vertices that can be annotated by types and predicates for attribute values. A specific vertex is represented by its identifier in a query graph. We provide an example how a query is formulated in Section 7.

## 3. Classification of "Why?" queries

In this section we classify unexpected result sets, present their corresponding "Why?" queries, possible solutions for these queries as well as modification operations to quantify or modify query parts responsible for an unexpected result set.

### *3.1. "Why?" queries*

We focus on four types of unexpected results: an empty result, too few or too many answers, or an interesting answer is missing.

Each of these scenarios can be studied by one of the "Why?" queries presented in Table 1. A *"Why Empty?"* query, also known as a differential query [2], solves an empty-answer problem by discovering query elements missing from a data graph. A *"Why So Few?"* query investigates an insufficient number of answers (data subgraphs). A *"Why Not?"* query detects why specific data subgraphs that are interesting to a user are missing from a result set. The subgraphs of interest are specified by their unique identifiers or values of attributes. A *"Why So Many?"* query discovers which query elements increase the size of a result set in such a way that it exceeds an expected number.

### *3.2. Classification of solutions*

We classify the solutions for "Why?" queries in two categories: subgraph-based approaches and solutions based on query rewriting techniques.

#### *3.2.1. Subgraph-based approach*

These methods study the graph query itself and match it to the data graph. As an answer they deliver intermediate results and "differences" that represent elements of a graph query that are potentially responsible for the delivery of an unexpected result. According to these methods, the original graph query is left unchanged. To quantify the difference between the graph query and its executed part, we propose a set of basic modification operations that we introduce later in Section 3.3.

#### *3.2.2. Query rewriting approach*

The methods of this group aim to rewrite the query in such a way that a desired goal is achieved. To modify the graph query, basic and complex operations are used that describe changes to be applied to a graph query. From possible query candidates generated during rewriting that one is chosen, which requires the least number of changes introduced by the basic operations applied to an original query to produce this candidate (see Section 3.3).

Final edited form was published in "Journal of Computer and System Sciences". 82 (1), S. 3-22. ISSN: 0022-0000.
https://doi.org/10.1016/j.jcss.2015.06.007

**Table 2**
Basic modification operations.

| Type | Target | Relaxation operation | Augmentation operation |
|---|---|---|---|
| Topology | Edge | Edge deletion | Edge insertion |
| | Vertex | Vertex deletion | Vertex insertion |
| | Edge | Direction deletion | Direction insertion |
| | Edge | Source (target) deletion | Source (target) insertion |
| Semantic | Edge, vertex | Predicate deletion | Predicate insertion |
| | Edge, vertex | Operator deletion | Operator insertion |
| | Edge, vertex | Constant deletion | Constant insertion |
| | Edge | Type deletion | Type insertion |

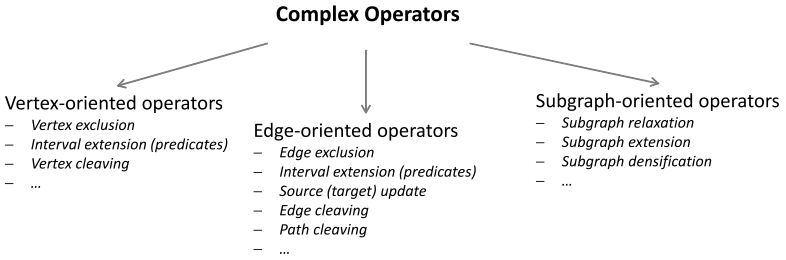



**Fig. 1.** Classification of complex modification operations.

### *3.3. Basic and complex operations*

To quantify the difference between the original query and its part delivering an expected answer and to rewrite queries, graph edit distance operations [6] can be used that include vertex/edge insertion, vertex/edge deletion, and vertex/edge substitution. We complete this list of graph edit operations with operations specific for property graphs and introduce them as basic and complex modification operations. As we can see in Table 2, we introduce semantic (notational) operations for predicates and types. A predicate is a constraint for an attribute value including an attribute name, a comparison operator, and a constant. We also distinguish a type as a special kind of attribute that vertex/edge can have. It allows us to extract such subgraphs from a data graph like "friend-of-friend" networks.

Basic operations describe minimal modifications that can be applied to a graph query. We classify them according to their target graph elements and types. The number of applied basic operations is used as a difference measure between the original query graph and its executed subgraph in subgraph-based methods and as a quantifier for the modifications in query rewriting methods (see Table 2). Some basic operations are not atomic, because they do not guarantee the correctness of a query after execution of these operations. Therefore, they are used in conjunction with other basic operations. For example, "Constant Deletion" removes a constant value from a predicate. After it has been applied to a query, the resulting query is not complete and the execution of the following basic operations is still required: "Operator Deletion" and "Predicate Deletion". The use of the operation "Constant Deletion" alone does not make any sense, but it is necessary in any changes of predicates like modifications of a predicate's interval. Operations that relax a graph query are called *relaxation operations*. Each operation has an inverse *augmentation operation*.

Complex operations express more sophisticated changes, executing several basic modifications at once. We classify them in three groups according to their targets: vertex-oriented, edge-oriented, and subgraph-oriented operations and present some examples in Fig. 1. For instance, to change a graph query, we can transform its subgraph by "Subgraph Densification" or "Subgraph Extension" operations. The first one increases the density of a subgraph: the number of vertices is left without any changes, but the amount of edges increases. The second one extends a subgraph: both the number of vertices and edges increases. We refer any complete predicate modification, e.g., "Interval Extension", to complex operations because it introduces several modifications at once: predicate, operation, and constant changes.

*Discussion* In this paper we focus mainly on two extreme cases: on the subgraph-based approach for "Why Empty?" and "Why So Many?" queries. "Why So Few?" queries can be generally represented by "Why So Many?" queries with an inverse goal – to increase the cardinality of the result set. "Why Not?" queries can be expressed by "Why Empty?" queries with constrains [7]. These queries conduct a search from the specified graph elements of interest and first discover more relevant subgraphs. Therefore, with these two extreme cases we can present all "Why?" queries. In addition, in this paper we highlight the topology of a graph and, as a consequence, focus on topological operations like for example "Vertex Deletion" and "Edge Deletion" for quantifying the differential graphs. Query rewriting and corresponding modification operations are left as future work.

Final edited form was published in "Journal of Computer and System Sciences". 82 (1), S. 3-22. ISSN: 0022-0000.
https://doi.org/10.1016/j.jcss.2015.06.007

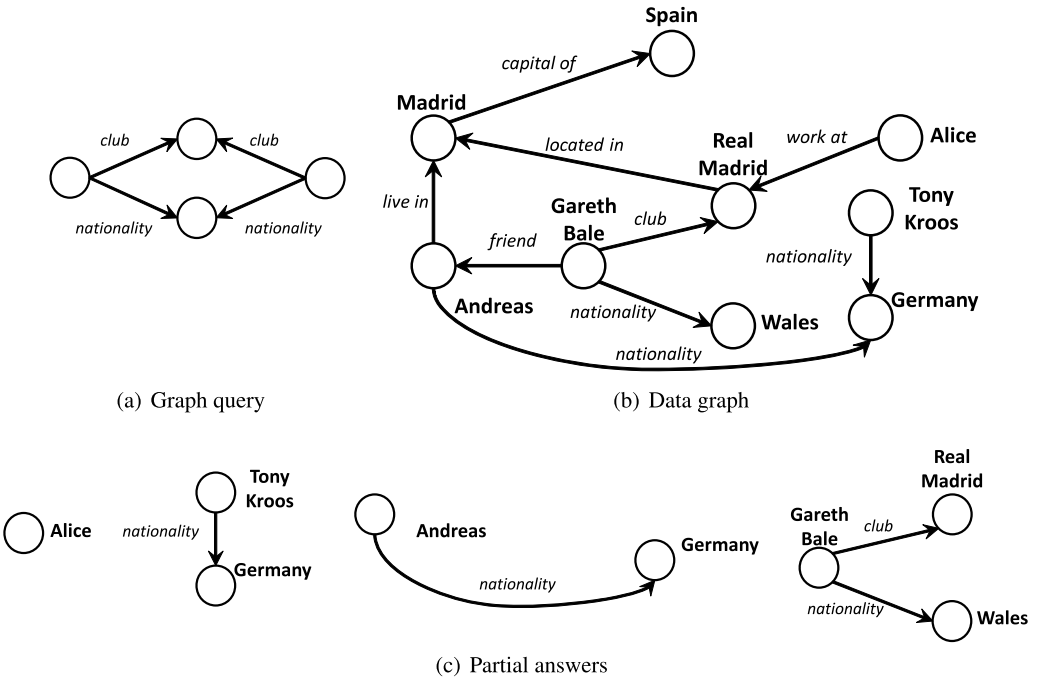



**Fig. 2.** Example of a "Why Empty?" query, a data graph and corresponding partial answers.

## 4. "Why Empty?" queries

In this section we present our work [2] for the first extreme case: a user receives an empty result set from a graph database.

### *4.1. Example*

In Fig. 2 we present an example of a query and a data graph. This query represents a pattern from four vertices and four edges of types "club" and "nationality". The search for this pattern in the data graph in Fig. 2(b) will deliver an empty result, because it is not represented in the data graph. Still we can see multiple partial results in the data graph, which vary from a subgraph containing a single vertex like "Alice" to a subgraph consisting of up to three vertices (some partial results are illustrated in Fig. 2(c)). The number of such partial answers can be very high and therefore to explain why each of them does not become an answer can be a non-feasible task. Based on this observation, we generate an explanation only for the largest discovered partial result like the right one in Fig. 2(c), because it holds the maximum available information in a data graph matching to a query.

### *4.2. Foundations*

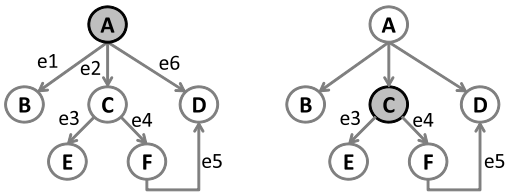
To understand, which part of a query can be found in a data graph and which part is missing, we have to find the maximum common subgraphs in a data graph $G_d$ for a query graph $G_q$ and then calculate the differential graphs between them and the original graph query.

**Definition 6** *(Maximum common connected subgraph (MCCS)).* Let $G_d = (V_d, E_d, u_d, f_d, g_d, A_{V_d}, A_{E_d})$ be a data graph and $G_q = (V_q, E_q, u_q, f_q, g_q, A_{V_q}, A_{E_q})$ be a query graph. A **maximum common connected subgraph** $G'_d = (V'_d, E'_d, u'_d, f'_d, g'_d, A_{V'_d}, A_{E'_d})$ for $G_d$ and $G_q$ is a common connected subgraph of $G_d$ and $G_q$ such that there is no common connected subgraph $G''_d = (V''_d, E''_d, u''_d, f''_d, g''_d, A_{V''_d}, A_{E''_d})$ with $V'' \supsetneq V'$ or $E'' \supsetneq E'$.

#### *4.2.1. Maximum common connected subgraphs discovery*

Maximum common subgraphs between a graph query and a data graph can be discovered by maximum common connected subgraph algorithms [8,9]. The computation depends on how a data graph is stored and processed. A common way to represent a data graph is an adjacency matrix or adjacency list [10]. For example, a matrix $M$ consists of $n \times n$ elements, where $n$ is the number of vertices in a graph. Each element of a matrix $a_{ij}$ with a value 1 represents an edge between vertices $i$ and $j$. A maximum common connected subgraph is calculated by linear algebra operations. If a graph is a property graph, then its attributes can be stored in separated structures and can be used during prefiltering.

Final edited form was published in "Journal of Computer and System Sciences". 82 (1), S. 3-22. ISSN: 0022-0000.
https://doi.org/10.1016/j.jcss.2015.06.007




(a) All edges processed (b) A few edges processed

**Fig. 3.** Depth-first search.

To discover maximum common connected subgraphs, Ullmann's [8] and McGregor's [9] algorithms can be used as a base for traversal operations in graph databases. Both methods are backtracking algorithms: While the Ullmann's algorithm is a tree enumeration procedure, the McGregor's method implements a depth-first search that begins at the root and traverses the graph as far as possible along each branch before backtracking. Assuming the depth-first procedure to start from the grey vertex in the example shown in Fig. 3(a), we begin from vertex $A$ and explore all edges of the graph as follows: $e1, e2, e3, e4, e5, e6$. If we start from vertex $C$ like in Fig. 3(b), then only edges $e3, e4, e5$ are traversed. To ensure the discovery of all maximum common connected subgraphs, the depth-first search is conducted for each vertex of a query, and a data graph is treated as undirected. Ullmann's [8] and McGregor's [9] algorithms work on matrices and provide extension points for pruning techniques and prefiltering options to reduce the search space. They rely on labeled graphs, which differ from our underlying property graph model [1]. To apply them to our use case, these algorithms have to be adapted to work with properties on edges and vertices. It means we have to consider only those edges and vertices which descriptions match the predicates and types given in a query graph.

A maximum common connected subgraph problem can also be modified for the search of a maximum clique like in the Durand–Pasari algorithm [11] and in the Balas Yu algorithm [12]. These algorithms are also tree-search algorithms. Some of them work better with sparse graphs, others with dense graphs. According to their comparison [13], the McGregor's algorithm shows good results in all cases and has the best space complexity. Based on these observations, we have chosen it as the base for our discovery of maximum common connected subgraphs.

*4.2.2. Differential graphs*

With a maximum common connected subgraph algorithm we can detect, which query part has an answer in a data graph. To determine which structural part is missing, we need to compute a differential graph – the difference between discovered maximum connected subgraphs and a query graph.

A differential graph includes those query vertices and edges, which were not discovered during query processing, and the instances of query vertices adjacent to a maximum common connected subgraph.

**Definition 7** *(Differential graph).* Let $G_d = (V_d, E_d, u_d, f_d, g_d, A_{V_d}, A_{E_d})$ be a data graph, $G_q = (V_q, E_q, u_q, f_q, g_q, A_{V_q}, A_{E_q})$ be a query graph, and $G'_d = (V'_d, E'_d, u'_d, f'_d, g'_d, A_{V'_d}, A_{E'_d})$ is a maximum common connected graph for $G_d$ and $G_q$. A graph $G'_q = (V'_q, E'_q, u'_q, f'_q, g'_q, A_{V'_q}, A_{E'_q})$ is a **differential graph** for $G_d, G_q, G'_d$ such that $E'_q \subset E_q$, $E'_q \not\subset E'_d$ and $V'_q = V'^{s}_d \cup V^{s}_q$, where $V'^{s}_d \subset V'_d$, $V^{s}_q \not\subset V'_d$ and $\forall v'_d \in V'^{s}_d, v_q \in V_q, v'_d = v_q : degree(v'_d) < degree(v_q)$.

The complexity of computing a differential graph is $O(k)$, where $k = m + n + l + s$, $m$ is a number of edges, $n$ is a number of vertices, $l$ is a number of attributes, and $s$ is a number of types in a query.

*4.2.3. "Why Empty?" queries – differential queries in graph databases*

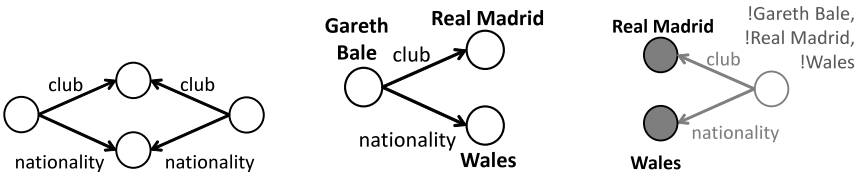
If a user gets an empty result set to a query, a "Why Empty?" query can be conducted that shows, which query part is addressed in data and which part is missing. For this purpose, a "Why Empty?" query detects maximum common connected subgraphs and computes their corresponding differential graphs, which prevent a graph processing engine from the delivery of a non-empty result set to a user.

Assuming we search for two soccer players originating from the same country and playing in the same club. Then the query graph could be represented like in Fig. 4(a). A possible answer to this "Why Empty?" query would consist of a maximum common connected subgraph $G'_d$ as shown in Fig. 4(b), and a missing part of a query with constraints $G'_q$ as in Fig. 4(c). The first part includes all discovered instances of edges and vertices like "Gareth Bale", "Real Madrid", and "Wales". The second part consists of instances of discovered adjacent vertices (dark grey), missing query vertices and edges (grey), and constraints for vertices (grey).

*4.3. Processing of "Why Empty?" queries in graph databases*

The processing of "Why Empty?" queries consists of two steps: (1) the detection of maximum common connected subgraphs by using an extended version of the McGregor's algorithm [9] for property graphs, and (2) the calculation of differential graphs.

Final edited form was published in "Journal of Computer and System Sciences". 82 (1), S. 3-22. ISSN: 0022-0000.
https://doi.org/10.1016/j.jcss.2015.06.007




(a) "Why Empty?" query (b) Maximum common connected subgraph (c) Differential graph

**Fig. 4.** A "Why Empty?" query and its answer: which two vertices are originally from the same country and play in the same club?

**Algorithm 1** GraphMCS.

```
1: function MCCSSEARCH(query graph G_q)
2:     result graphs graphs[]
3:     maximum subgraph result
4:     for all edge ∈ G_q do
5:         sources[] ← get sources for edge
6:         for all sourceVertex ∈ sources[] do
7:             graph ← ∅
8:             graph ← DFS(sourceVertex, edge, true, graph)
9:             graphs[] ← graph
10:    for all graph ∈ graphs[] do
11:        if size(graph) ≥ size(result) then result ← graph
       return result
12: /*depth-first search*/
13: function DFS(source, edge, isStart, graph)
14:    if !isStart then edge ← get next edge for edge
15:    if no further edge then return graph
16:    targets[] ← traverse edge from source
17:    targets[] ← filter targets[]
18:    for all targetVertex ∈ targets[] do
19:        extend graph with edge (source, targetVertex)
20:        graph ← DFS(targetVertex, edge, false, graph)
       return graph
```

*4.3.1. Detection of maximum common connected subgraphs*

To detect the maximum commonality between a query and a data graph, we have chosen the McGregor's maximum common connected subgraph algorithm [9] as the base for the GraphMCS algorithm presented in Algorithm 1, which uses a depth-first search (Fig. 3).

To leverage the McGregor's algorithm for property graphs, the edges and vertices tables of our graph database have to be processed. First, the projection on a vertices table reduces the number of start vertices at line 5. Second, each edge is processed by the graph traversal operator at line 16. Finally, the target vertices are filtered according to their predicates (see line 17). To ensure that the algorithm finds a maximum common connected subgraph, it is started from all query vertices as multiple starting points at lines 4–9. The maximum common connected subgraph is stored for each starting point in a set (see line 9). After all runs the best subgraph is chosen from the collected set (see lines 10–11).

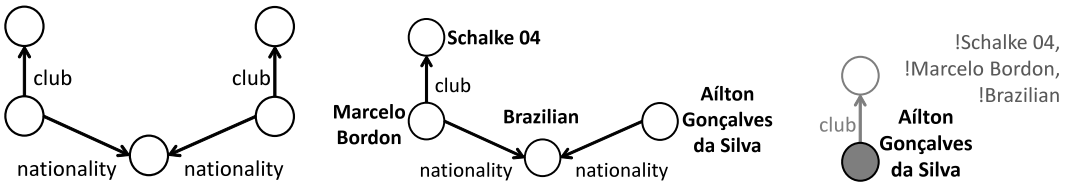
*4.3.2. Calculation of differential graphs*

To compute a missing part of a query, we use a query graph and a discovered maximum common connected subgraph. The process consists of two steps: (1) the split of discovered and undiscovered vertices and edges, and (2) the completion of an undiscovered part with attributes or vertices conditions.

In our first step, during processing we store the mapping between data edges and query edges as well as data vertices and query vertices in temporary tables. The differential graph consists of edges and vertices of a query graph, which have no instances in these temporary tables. Some edges in the differential graph will have only single vertices at their ends, because other end vertices have already been traversed. Therefore, we have to include the discovered edges' ends into the differential graph in the second step – the completion of the differential graph with attributes or vertices conditions.

In the second step, we detect, which conditions have to be applied to the graph retrieved in the first step. We study the table with discovered vertices and the query description and assign conditions to the differential graph according to several rules: (1) If a query edge is not discovered, but at least one of its end vertices has already been found, then this is a positive condition. It means we include a discovered end vertex into the differential graph. In the example presented above, the two dark grey vertices represent such conditions (see Fig. 4(c)). These vertices are included in a differential graph and can be used as starting points for a future explorative search. (2) If a query vertex and all its query edges (incoming and outgoing) are discovered in a data graph, then this vertex is a negative condition, and its instance has to be excluded

Final edited form was published in "Journal of Computer and System Sciences". 82 (1), S. 3-22. ISSN: 0022-0000.
https://doi.org/10.1016/j.jcss.2015.06.007




(a) "Why So Many?" query (b) Cardinality-bounded maximum common connected subgraph (c) Differential graph

**Fig. 5.** A "Why So Many?" query and its answer: which two players of the same nationality play in different clubs?

from the non-discovered query vertices. In our example this can be "Gareth Bale" (see Fig. 4(c)), which does not have to be considered in a future explorative search.

To quantify a differential graph, we use the information collected at the discovery of a maximum common connected subgraph. At that step we know, which edges were not discovered by the search.

**Definition 8.** Let $S$ be a set of basic modification operations, $G_q$ be a query graph, $G_d$ be a data graph, $G'_d$ be a maximum discovered common subgraph for $G_d$ and $G_q$, and $G'_q$ be a differential graph. A **graph edit distance** (difference) between $G_d$ and $G_q$ is a number of operations $s_i \in S$ applied to $G_q$ in order to get $G'_d$.

Assuming our running example in Fig. 4, the differential graph includes the non-discovered vertex, two discovered vertices, and two non-discovered edges "club" and "nationality". To get a query corresponding to the response in Fig. 4(b), we have to apply the following operations to the original query in Fig. 4(a): "Edge Deletion" for edges "club" and "nationality", "Type Deletion", "Direction Deletion", "Source Deletion", and "Target Deletion" for these edges, and "Vertex Deletion" for the vertex between these two edges. Therefore, the graph edit distance is equal to eleven.

## 5. "Why So Many?" queries

In this section we present the second extreme case, when a user receives too many answers from a graph database.

### 5.1. Foundations

If a user gets an unexpectedly large result set to a query, a "Why So Many?" query can be conducted that shows discovered intermediate results, before the resulting cardinality has exceeded the expected cardinality, and calculates their corresponding differential graphs.

For this purpose, a graph processing engine has to detect (1) a cardinality-bounded maximum common connected subgraph between a query graph and a data graph, which has a number of data instances that does not exceed the expected cardinality, and (2) its corresponding differential graph that prevents a graph processing engine from the delivery of an expected smaller result set.

**Definition 9** *(Cardinality-bounded MCCS).* Let $G_d$ be a data graph, $G_q$ be a query graph, and $C_{maxExp}$ be a maximum expected cardinality of a result set. A **cardinality-bounded maximum common connected subgraph** $G_d^{bnd}$ for $G_d$, $G_q$, and $C_{maxExp}$ is a common connected subgraph of $G_d$ and $G_q$ such that $\forall$ common connected subgraph $G''_d$ for $G_d$, $G_q : G_d^{bnd}$ is a connected subgraph of $G''_d$, a total number of such graphs $\sum G''_d > C_{maxExp}$.

Assuming a modified version of our running example in Fig. 5(a), we search for two soccer players from the same country who play in different clubs. We want to get not more than ten answers ($C_{maxExp} = 10$). But we get too many answers: the number of discovered answers $K = 84$ exceeds the predefined maximum expected cardinality $C_{maxExp} = 10$. By processing the graph, the number of answers grows, when we search for the "club", because the same player can play at different times during his career in several clubs. While the player "Marcelo Bordon" played in four different clubs, the player "Aílton Gonçalves da Silva" played in 21 different soccer clubs. We want to get less number of answers and therefore run a "Why So Many?" query. A possible answer to this "Why So Many?" query would consist of a cardinality-bounded maximum common connected subgraph $G_d^{bnd}$ as shown in Fig. 5(b), and an unbounded part of a query with constraints $G_q^{ubnd}$ as in Fig. 5(c). The first part includes, for example, all discovered instances of vertices and edges like "Aílton Gonçalves da Silva", "Schalke 04", "Brazilian", and "Marcelo Bordon". The second part consists of instances of discovered adjacent vertices (dark grey), unbounded query vertices and edges (grey), and constraints for vertices (grey).

Final edited form was published in "Journal of Computer and System Sciences". 82 (1), S. 3-22. ISSN: 0022-0000.
https://doi.org/10.1016/j.jcss.2015.06.007

**Algorithm 2** BoundedMCS.

```
1: function BOUNDEDSEARCH(query G_q, expected cardinality C_maxExp)
2:     maximum connected data graphs result
3:     discovered finalGraphs[], data edges edgeBaskets[]
4:     rejectedEdges[], acceptedEdges[]
5:     for all edge ∈ G_q do
6:         edgeBaskets[](edge) ← get data instances of edge
7:         edgeBaskets[](edge) ← filter ends for edgeBaskets[](edge)
8:     for all edge ∈ G_q do
9:         if size(edgeBaskets[](edge)) ≤ C_maxExp then
10:            acceptedEdges[] ← ∅
11:            extend graphs[] with edgeBaskets[](edge)
12:            acceptedEdges[] ← edge
13:            joinAll(G_q, edge, edgeBaskets[], rejectedEdges[], acceptedEdges[], graphs[])
14:            finalGraphs[] ← graphs[]
15:    for all graph ∈ finalGraphs[] do
16:        if size(graph) ≥ size(result) then
17:            result ← graph
       return result
18: /*join all edge baskets*/
19: function JOINALL(query, edge, edgeBaskets[], rejectedEdges[], acceptedEdges[], graphs[])
20:    adjacentEdges[] ← get adjacent edges for edge
21:    for all edge ∈ adjacentEdges[] do
22:        if not processed edge then extend graphs with edgeBaskets[](edge)
23:        if size(graphs[]) ≤ C_maxExp then
24:            accepctedEdges[] ← edge
25:            joinAll(G_q, edge, edgeBaskets[], rejectedEdges[], acceptedEdges[], graphs[])
26:        else
27:            remove edge from graphs[]
28:            rejectedEdges[] ← edge
29:    for all edge in query do
30:        if edge ∉ acceptedEdges[] AND edge ∉ rejectedEdges[] then
31:            rejectedEdges[] ← edge
       return
```

### 5.2. Processing of "Why So Many?" queries

A "Why So Many?" query consists of the original user query that delivered too many results plus a maximum cardinality threshold. Its processing consists of two steps: (1) the detection of cardinality-bounded maximum common connected subgraphs and (2) the calculation of differential graphs (unbounded query parts).

#### 5.2.1. Detection of maximum cardinality-bounded common connected subgraphs

To detect which part of a query matches the maximum expected cardinality $C_{maxExp}$, we have to find such maximum common connected subgraphs in a data graph $G_d$ for a query graph $G_q$ which number is lower than the expected cardinality. We developed an algorithm for the detection of cardinality-bounded maximum common connected subgraphs "BoundedMCS" (see Algorithm 2). It consists of several steps. First, we retrieve all edges from the database at lines 5–7. Second, we join them based on their ends (see line 13) using depth-first search in *JoinAll* function at lines 19–31. Third, we reject such joins that cause the resulting cardinality to exceed the maximum expected threshold at lines 27–28. The search is conducted from all edges as starting points at lines 8–13. Finally, the largest cardinality-bounded maximum common connected subgraphs are delivered to a user at lines 15–17.

#### 5.2.2. Calculation of differential graphs

To calculate differential graphs, we use information collected at the discovery of a cardinality-bounded maximum common connected subgraph. At that step we know which edges were rejected by the search. For the calculation we use collections of rejected edges and discovered subgraphs. The difference is calculated based on the number of basic modification operations (see Section 3.3) that have to be applied to a query graph in order to get discovered cardinality-bounded maximum query subgraphs. Assuming our running example in Fig. 5. The differential graph includes two discovered vertices and not discovered edge "club". To get a query corresponding to the response in Fig. 5(b), we have to apply the following operations to the query in Fig. 5(a): "Edge Deletion", "Type Deletion", "Direction Deletion", "Source Deletion", and "Target Deletion" for edge "club" and "Vertex Deletion" for its target vertex. Therefore, the size of the differential graph (the graph edit distance) is six.

## 6. Strategies to increase performance and quality

In this section we describe problems for the processing of traversal-based "Why Empty?" and join-based "Why So Many?" queries and propose our solutions for them. Afterwards, we provide several heuristic optimizations to increase the efficiency of these algorithms.

Final edited form was published in "Journal of Computer and System Sciences". 82 (1), S. 3-22. ISSN: 0022-0000.
https://doi.org/10.1016/j.jcss.2015.06.007

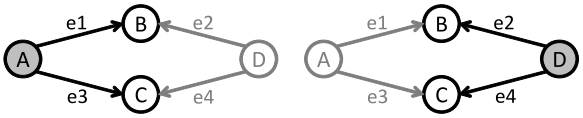



**Fig. 6.** Weakly connected graphs.

### 6.1. Problems of multiple starts and weakly connected graphs

Although the algorithms GraphMCS and BoundedMCS for both kinds of queries are different, they have several common features: They are based on the depth-first search and require multiple runs from different starting vertex (edge) to guarantee the delivery of the best results. Therefore, they have the same performance and quality problems and we can apply the same improvements to both of them.

GraphMCS and BoundedMCS take all vertices of a query as starting points and search for a maximum common subgraph or a cardinality-bounded maximum common subgraph from each query vertex. So, the graph processing engine touches the same data edges multiple times. On the one hand, this ensures that no edge is left out and all cardinality-bounded or maximum common connected subgraphs are discovered. On the other hand, this generates duplicate intermediate results and increases the response time dramatically.

We identified two problems, whose solution can increase the quality and the performance of the algorithms: to find larger graphs and to reduce the number of runs. First, both algorithms work only with connected graphs, therefore, only one-directed search for directed graphs is done. This can be solved by the extension of the search for weakly connected graphs. Second, if we reduce the number of runs by starting just from a single vertex, we can miss some maximum common connected subgraphs. However, the reduction of the number of runs can be done by a restart strategy for non-studied edges. We apply these extensions to both algorithms: GraphMCS and BoundedMCS.

#### 6.1.1. Processing of weakly connected graphs

The GraphMCS and BoundedMCS algorithms process the directed graph only in a forward direction, according to the depth-first search strategy. This can limit the size of discovered subgraphs and deliver subgraphs of potentially smaller size than could be determined if edges were traversed in both directions. To ensure the discovery of a maximum subgraph, we have to choose that vertex as a root, from where all vertices can be reached. Because the algorithms work only in a forward direction, it is not always possible to find the best start vertex.

For example, the query presented in Fig. 6 does not have any ideal start vertex. This is a weakly connected graph: it is connected, if directions of edges are not considered. For this query the depth-first search can discover subgraphs only with two edges and three vertices (*ABC* or *BCD*). Therefore, we need to modify the algorithm to also consider unreachable subgraphs.

To process queries with unreachable subgraphs, we introduce an all-covering spanning tree.

**Definition 10.** Let $G_q = (V_q, E_q, u_q, f_q, g_q, A_{V_q}, A_{E_q})$ be a graph query with $M_q$ edges and $N_q$ vertices. An **all-covering spanning tree** is a traversal order for $G_q$ presented as a set of tuples $a_i = (prev_i, source_i, next_i)$ such that

$$(1)\ prev_i = \begin{cases} \emptyset & \text{if } i = 0 \text{ OR } source_i = source_0, \\ next_{i-1} & \text{otherwise,} \end{cases} \tag{1}$$

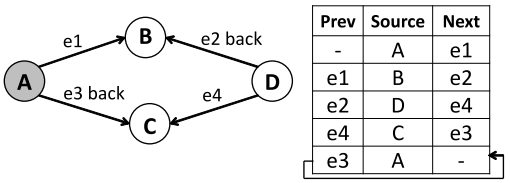
$$(2)\ source_i = \begin{cases} source(next_i) & \text{if } next_i \neq \emptyset \text{ AND traversed forward,} \\ target(next_i) & \text{if } next_i \neq \emptyset \text{ AND traversed backward,} \\ source(prev_i) & \text{if } prev_i \neq \emptyset \text{ AND traversed backward,} \\ target(prev_i) & \text{if } prev_i \neq \emptyset \text{ AND traversed forward,} \end{cases} \tag{2}$$

$$(3)\ next_i = \begin{cases} \emptyset & \text{if a number of non-traversed adjacent edges } k = 0, \\ e_j & \text{is a non-traversed edge adjacent to } source_i. \end{cases} \tag{3}$$

If the whole query graph is available in the data graph, then the all-covering spanning tree is able to cover all query vertices and edges in a single run. An edge can be included into the search in forward or backward direction. In case of a backward direction, an edge is marked with a flag “back”. This can be done without additional effort because of the underlying data model and the graph traversal operator provided by our graph database like in [5]. Another way would be to make a graph basically undirected by adding for each edge also an edge in the opposite direction or double table scans (one for forward traversal and one for backward traversal), which is less efficient.

We adapt Algorithms 1 and 2 to work with an all-covering spanning tree. From now on, we consider all edges for each vertex. Outgoing edges have priority over incoming edges and are processed first. After all outgoing edges are traversed, incoming edges are considered.

Final edited form was published in "Journal of Computer and System Sciences". 82 (1), S. 3-22. ISSN: 0022-0000.
https://doi.org/10.1016/j.jcss.2015.06.007




| Prev | Source | Next |
|---|---|---|
| - | A | e1 |
| e1 | B | e2 |
| e2 | D | e4 |
| e4 | C | e3 |
| e3 | A | - |

**Fig. 7.** All-covering spanning tree and the backtracking procedure.

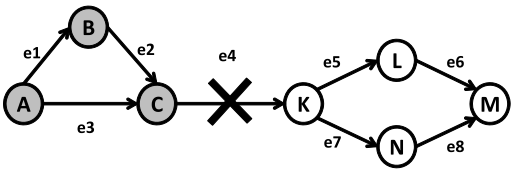



**Fig. 8.** Only white or grey part is traversed.

To guarantee a correct search, we maintain the all-covering spanning tree as a temporary table and refer to it during the backtracking procedure. This table records the mapping between previously traversed and next edges. The all-covering spanning tree has three columns: a previous edge, a source vertex, and a next edge. To save space, we can use a Boolean identifier of a traversed edge instead of an identifier for a source vertex. To keep our presentation simple, we use identifiers for the source vertices in the following example.

Assuming the search for the graph query presented on the left side in Fig. 7 begins from vertex $A$. At initialization, the spanning tree is empty. Vertex $A$ has two outgoing edges. After we have followed edge $e1$, we add the following entry into the table: no previous edge, source vertex is $A$, next edge is $e1$ $(-; A; e1)$. Now we are at vertex $B$ without any outgoing edges. We take incoming edge $e2$, mark it with "back", and add an entry into the table $(e1; B; e2)$. We repeat the process and traverse edges $e4, e3$. Finally, we are at vertex $A$ without any non-traversed edges and start the backtracking.

The backtracking procedure is done according to the created all-covering spanning tree. The last traversed edge is $e3$. We check its entry (column "Next") in the mapping table, take its previous edge $e4$ and go to source vertex $C$. There are no other non-traversed edges for vertex $C$, and so we continue the backtracking. The predecessor of $e4$ is $e2$ with vertex $D$, so we move to it. The procedure continues until it gets to source vertex $A$, where no further non-traversed edges exist.

A graph database gives the possibility of changing the direction through suitable storing and processing of edges. All edges are stored in a forward direction – "from a source to a target". For a "Why Empty?" query, if a query graph has to be traversed backward, the graph traversal operator changes the order of columns to be searched: "from a target to a source". Therefore, we need only to change the direction of an edge in the query description and pass it to the traversal operator. For a "Why So Many?" query, if a query graph has to be backward traversed, the join operator has to switch the columns to be joined: "from a target to a source".

*6.1.2. Restart strategy*

With the all-covering spanning tree, we can construct a traversal path for "Why Empty?" queries or establish a join order for "Why So Many?" queries, which include all vertices and edges, and process weakly connected graphs. Hereby, we solve the first problem of the uni-directed search. Now we do not need to iterate over all vertices multiple times. We just take one vertex and search from it. This approach works well if all edges of a query graph can be considered. In case of "Why Empty?" queries, however, some edges can be missing from a data graph. In case of "Why So Many?" queries some edges are not considered, otherwise, the result cardinality would exceed a maximum expected value. Therefore, such edges can split a query graph into several subgraphs, which are unreachable from each other. In this case if we start with a vertex from a smaller subgraph, we will miss a maximum common connected subgraph from another subgraph. Thereby, it leads to the second problem of missing maximum subgraphs, which can be solved by a restart strategy.

If we start a search from a single node that is located in the smaller connected subgraph of a query graph, we can potentially miss the larger subgraph, provided by another subgraph. The problem can be explained with a query graph containing a bridge. If a query has a bridge (see Fig. 8), which is not addressed in the data graph, then only a subset of vertices and edges is traversed. In our example query edge $e4$ does not have any matching data edges for "Why Empty?" queries or it increases the result cardinality dramatically for "Why So Many?" queries. In this case, a maximum common connected subgraph found by our algorithm GraphMCS would be the white or the dark-grey part. Therefore, if we do a single run in the dark area the maximum common connected subgraph will not be found, which is located in the white area. To solve this problem, we can resume the search with the edges, which were not traversed. The final maximum common subgraph would be unconnected and would contain all discovered maximum common connected subgraphs.

We maintain a list of traversed edges of a query graph. After the first set of maximum common connected subgraphs is returned, we remove those edges from the list that have already been traversed. The next step is taken from this set. This

Final edited form was published in "Journal of Computer and System Sciences". 82 (1), S. 3-22. ISSN: 0022-0000.
https://doi.org/10.1016/j.jcss.2015.06.007

strategy ensures the discovery of a maximum common subgraph for a given start vertex, if an all-covering spanning tree was constructed.

For example, at the beginning a query graph in Fig. 8 has an empty list of traversed edges. Assuming we start from the edge $e1$ and find edges $e1, e2, e3$. Edge $e4$ is not represented in a data graph. We add all four edges into the list of traversed edges and remove them from the list of start edges. We choose the next start among the edges $e5, e6, e7, e8$. The search from any of them will find the same subgraph of four edges. This is the maximum common connected subgraph. If we concatenate it with the first discovered maximum common connected subgraph, then we will get a maximum common unconnected subgraph. So, as a maximum common subgraph we will get a set of unconnected parts. This reduces the number of intermediate results. Such a methodology can potentially return larger subgraphs than the strategy for connected subgraphs.

### *6.2. Summary*

With an all-covering spanning and restart strategies tree we can detect larger cardinality-bounded maximum common subgraphs for "Why So Many?" queries and larger maximum common subgraph for "Why Empty?" queries and reduce the number of restarts. In the following we use maximum common connected and unconnected subgraphs and refer to them jointly as maximum common subgraphs.

### *6.3. Optimization strategies*

GraphMCS and BoundedMCS take all vertices of a query as starting points and search for a maximum common subgraph or a cardinality-bounded maximum common subgraph from each query vertex. This is a general solution which is time-consuming and leads to longer response time. To increase the efficiency of the proposed algorithms, we have developed several optimization strategies for start and restart edges for both kinds of queries, and for early termination conditions for "Why Empty?" queries.

#### *6.3.1. Choice of start and restart edges*

The GraphMCS and BoundedMCS algorithms process a graph from all query vertices, and then the largest graph is chosen and delivered as a maximum common connected subgraph. With an all-covering spanning tree and multiple restarts as proposed in Section 6.1, we can ensure that the whole query can potentially be traversed from each query edge in the best case. The question is: Which query edge should be taken as a start to overcome the multiple search from all query vertices and multiple traversal of the same data edges? This would increase the response time of the system which strongly depends on the cardinality of the processed edges and vertices. To decrease the number of intermediate results and solve multiple search over the same data, we extend our algorithms with several heuristics to select a start vertex, a start edge, and a next edge to traverse. These heuristics increase the performance of the search, but do not guarantee the discovery of an optimal solution.

*Heuristic I: Maximal in- and out-degree* This heuristic chooses the first query vertex and query edge to be processed based on the in- and out-degree of a query vertex. A query vertex with the maximal in-degree or out-degree is selected as the starting point.

**Heuristic 1** *(Maximal in- and out-degree). Let* $G_q = (V_q, E_q, u_q, f_q, g_q, A_{V_q}, A_{E_q})$ *be a query graph. A vertex* $v_i \in V_q$ *is a* ***starting vertex*** *if* $\textit{in-degree}(v_i) \geq \textit{in-degree}(v_j) \forall v_j \in V_q, i \neq j$ *or* $\textit{out-degree}(v_i) \geq \textit{out-degree}(v_j)\ \forall v_j \in V_q, i \neq j$.

This heuristic relies only on a query graph and does not consider the statistical information about the underlying data graph. This heuristic is supported by the fact that for a vertex with a higher degree, more edges need to be processed, and, therefore, we can discover a maximum common subgraph earlier. This strategy can potentially reduce also the number of restarts.

*Heuristic II: Minimal edge and vertex cardinality* This heuristic chooses the first query vertex and query edge to be processed based on the their cardinalities. A cardinality of an edge (vertex) shows the number of data instances of a specific query edge (vertex). A query edge (vertex) with the minimal cardinality is selected as the starting point.

**Heuristic 2** *(Minimal edge and vertex cardinality). Let* $G_q = (V_q, E_q, u_q, f_q, g_q, A_{V_q}, A_{E_q})$ *be a graph query and* $G_d = (V_d, E_d, u_d, f_d, g_d, A_{V_d}, A_{E_d})$ *be a data graph. A vertex* $v_i \in V_q$ *is a* ***starting vertex*** *if its cardinality* $0 < C(v_i) \leq C(v_j)\ \forall v_j \in V_q, i \neq j$.

Analogously, we choose a staring edge based on its cardinality. This heuristic requires the calculation of the cardinalities for all query edges and vertices in advance before a query is executed. Vertices and edges are sorted separately according to their cardinality in an ascending order. This heuristic aims to reduce the size of intermediate results in order to increase the performance of queries. For this, an edge with the lowest cardinality is chosen as the starting edge. This heuristic with

Final edited form was published in "Journal of Computer and System Sciences". 82 (1), S. 3-22. ISSN: 0022-0000.
https://doi.org/10.1016/j.jcss.2015.06.007

the use of an all-covering spanning tree allows also to choose a search direction, based on the cardinality of a source and a target vertex. If the cardinality of a target is less than the cardinality of a source then an edge has to be traversed in a backward direction. Otherwise, we use forward processing. We can use this heuristic also for restarting the search. In this case, the cardinality of edges is calculated. In addition, for "Why Empty?" queries this method has the advantage that if an edge has cardinality $C(e_j) = 0$ then it is discarded from the search. This reduces the number of table scans and therefore makes the search more efficient.

*6.3.2. Threshold-based termination condition for "Why Empty?" queries*

In general, the GraphMCS algorithms for "Why Empty?" queries stops when no more edges are found and a backtracking procedure returns to start. In addition, there can be the cases, when a graph processing engine can stop the search earlier. A threshold can be an estimated size of a maximum common subgraph or the number of discovered maximum common subgraphs, which could be derived from a data graph. To calculate these numbers, we can reuse the above presented cardinality of a query. If a query graph has $N$ edges, and $M$ edges ($M \in N$) are represented in a data graph ($C(e_j) > 0$), then the maximum common subgraph can have only $M$ edges. After $M$ edges are found, the search can be stopped. Similar rules can be formulated for sources and targets.

Assuming we have a query with four vertices and three edges with the following predicate cardinalities: $C(e_1) = 5$, $C(e_2) = 2$, $C(e_3) = 0$, then the maximum common subgraph can only consist of up to two edges, and we can have a maximum of five graphs like this. We can terminate our search, after the first subgraph with two edges has been discovered.

## 7. Evaluation

In this section we evaluate "Why Empty?" and "Why So Many?" queries in terms of the best algorithms' configuration, scalability with the increasing size of a query graph for different topologies, efficiency of optimization techniques, and cardinality of a result set for different thresholds. We describe the evaluation setup in Section 7.1. Then, we discuss configurations and scalability of proposed algorithms on the example of "Why Empty?" queries in Sections 7.2–7.3, 7.5. We discuss optimization techniques for start and restart strategies for both query types in Section 7.4. Finally, we evaluate a special characteristic of "Why So Many?" queries – maximum expected cardinality in Section 7.6.

### *7.1. Evaluation setup*

We have implemented our algorithm and its optimizations in the in-memory column database GRATIN [3], which provides the graph abstraction as described in Section 2 with an index accelerating graph traversal operations. Each query to our graph database is formulated as a property graph consisting of edges and vertices that can be annotated by types and predicates for attribute values. They are represented by a JSON-like internal query language. Each vertex and edge has an id that is unique inside a query. A description of an edge includes edge id, ids of source and target vertices, type, predicates. A description of a vertex includes a vertex id, ids of outgoing and incoming edges, predicates. Below we present an example of a query with a single edge of type *friendOf* that connects a vertex with $age = 10$ with a vertex of type *person*.

```
1 {"query":{
2     "graph":"dbpedia"
3     "vertices":[
4         {"id":"v0","outEdges":[{"id":"e1"}],"inEdges":[],"properties":[{"expression
              ":"age=10"}]},
5         {"id":"v1","inEdges":[{"id":"e1"}],"outEdges":[],"properties":[{"expression
              ":"type=person"}]}]
6     "edges":[
7         {"id":"e1","type":"friendOf","source":"v0","target":"v1","properties":[]}]
8 }}
```

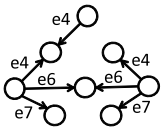
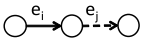
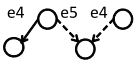
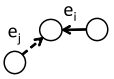
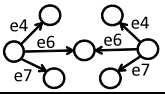
We have created a property graph from DBpedia data, where labels represent attribute values of entities. Our property graph has about $30K$ vertices and $213K$ edges. We have tested each query ten times and have taken the average response time and an average size of a discovered maximum common subgraph or a discovered cardinality-bounded maximum common subgraph as measures. The evaluated "Why Empty?" queries are presented in Table 3.

### *7.2. Configuration*

In this evaluation we study several configurations of the algorithm GraphMCS for "Why Empty?" queries. We evaluate the use of an all-covering spanning tree and a restart strategy. The first configuration "multiple start, no tree" is a basic configuration: a search is conducted from all query vertices, edges are traversed only in a forward direction (an all-covering spanning tree is not constructed). This configuration does not include our optimization strategies and can potentially miss larger common subgraphs. The second configuration "multiple start, with tree" introduces a first extension: an all-covering

Final edited form was published in "Journal of Computer and System Sciences". 82 (1), S. 3-22. ISSN: 0022-0000.
https://doi.org/10.1016/j.jcss.2015.06.007

**Table 3**
"Why Empty?" query templates used in the evaluation.

| Experiment | Figure | Query | Edge types |
|---|---|---|---|
| Configuration | Fig. 9 |  | Three different types |
| Topology: path | Figs. 10(a), 10(b) |  | Random edges |
| Topology: zigzag | Figs. 10(c), 10(d) |  | Two different types |
| Topology: star | Figs. 10(e), 10(f) |  | Random edges |
| Optimization | Fig. 11(a) |  | Three different types |

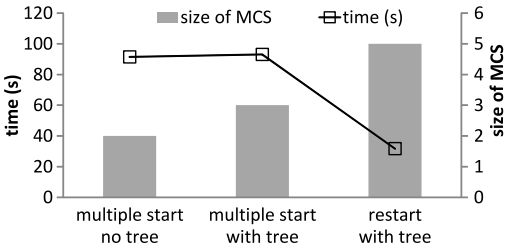



**Fig. 9.** Configuration: response time and size of maximum common subgraph for different traversing strategies.

spanning tree. In this case a search is conducted from all query vertices, edges are traversed in both directions. This configurations allows to process weakly connected graphs. The third strategy "restart, with tree" uses also a restart strategy (processing of unconnected components). In this case the search is triggered only from one starting vertex. After the search has finished, a restart strategy triggers the processing of non-traversed edges. This configuration also supports the processing of weakly connected graphs.

In Fig. 9, we present the response times in seconds and average size of MCS for each configuration of a "Why Empty?" query. As we can see in Fig. 9, the restart strategy discovers larger graphs with shorter response times and less intermediate and final results. Although the method with the all-covering spanning tree has a longer response time (because of the tree construction), it can discover larger subgraphs. The response time and the size of a maximum common subgraph (MCS) are the best for the restart strategy with an all-covering spanning tree construction.

### *7.3. Topology*

Next we conduct a detailed analysis of the best configuration of the previous step: restart with the all-covering spanning tree "restart, with tree" regarding different topologies of a graph query. For this, we have constructed several queries, which consist of edges of similar semantics (for a specific topology). The star and path topologies use randomly chosen edges from the data set. In this case, we evaluate a general situation, when no knowledge about the data graph is available. The zigzag evaluates queries with two edge types. In this case we introduce some knowledge about the data graph into the evaluation: we assume having edges of two different types connected on a target vertex. Here we evaluate the response time, an average size and the number of discovered maximum common subgraphs for three topologies for an increasing size of a graph query and present results in Fig. 10.

In the path topology, the first eight edges are rarely presented in a data set and produce few MCS (see Fig. 10(b)). The ninth and tenth edges are more often representative in the data graph and therefore cause a strong increase in the number of discovered MCS. This result shows the dependency of the response time on the size of discovered MCS (see Fig. 10(a)).

In the zigzag topology, the number of MCS changes significantly. In this case, the response time corresponds to the change of the number of discovered MCS (see Figs. 10(c)–10(d)). This evaluation shows fluctuations in the results, which are caused by the fact that each second or third edge is missing from a data graph. This can be seen in Fig. 10(c), where for example discovered subgraphs for Query 2 and Query 3 have the same size. If a new added edge is discovered during a restart, then it increases the number of subgraphs, otherwise it reduces the size of a result.

Final edited form was published in "Journal of Computer and System Sciences". 82 (1), S. 3-22. ISSN: 0022-0000.
https://doi.org/10.1016/j.jcss.2015.06.007

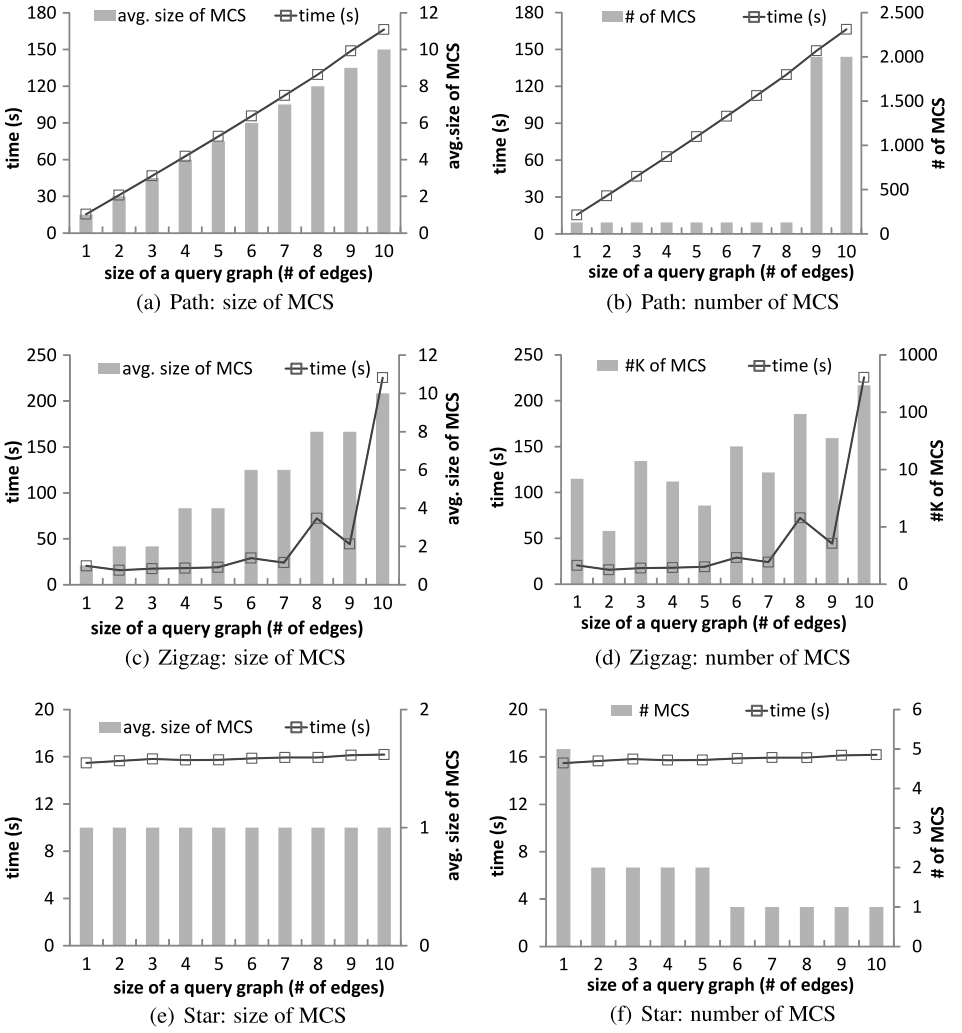



**Fig. 10.** Evaluation of different configurations and topologies.

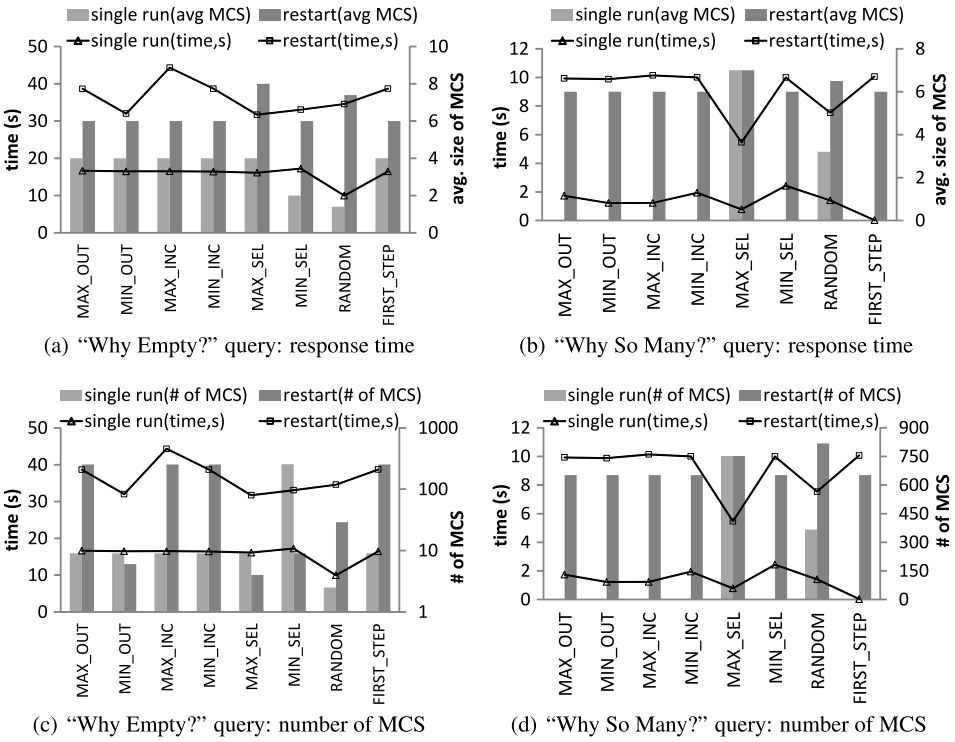
The third evaluation for the star topology (see Figs. 10(e)–10(f)) highlights the benefit of our optimization strategy "minimum cardinality". For each new query the graph processing engine chooses an edge with the minimal number of instances. Therefore, the size of the result set for the star topology is reduced, but the size of MCS stays the same. It shows that there are no edges presented in the query that are adjacent in a data graph. In addition, the size of MCS cannot be increased by the restart strategy because of the star topology, if the first processed edge has already been discovered.

Comparing the results of the evaluation on our three topologies, we conclude that (1) the response time depends on the number of discovered MCS, the size of MCS, and the size of the query graph, (2) to decrease the complexity of the search, optimization strategies have to be used, which reduce the number of intermediate results, (3) the use of a restart strategy for the star topology does not increase the size of MCS, if the first processed edge is present in a data set.

### *7.4. Optimization strategies*

In this section, we evaluate the optimization heuristics for start and restart strategies presented in Section 6 based on the vertex in- and out-degrees, and cardinality of an edge and a vertex. The evaluation results of start strategies for a single start and for restart are presented in Fig. 11. We evaluate maximum out-degree "MAX_OUT", minimum out-degree "MIN_OUT", maximum in-degree "MAX_INC", minimum in-degree "MIN_INC", minimum cardinality "MAX_SEL", maximum cardinality "MIN_SEL", random "RANDOM", first-step described "FIRST_STEP" heuristics. While Figs. 11(a)–11(c) show the evaluation for the "Why Empty?" query, Figs. 11(b)–11(d) present solutions for the "Why So Many?" query of a star topology with maximum expected cardinality $C_{maxExp} = 1000$. We observe that with restarts we can increase the average size of MCS. Regardless of the strategy, we get MCSs of the same or larger size by using the restart configuration. The response time for the search can be reduced by using an appropriate optimization strategy. For example, the "MAX_SEL" strategy reduces the number of intermediate results, the number of MCSs, and the response time. The experiments for both queries show similar

Final edited form was published in "Journal of Computer and System Sciences". 82 (1), S. 3-22. ISSN: 0022-0000.
https://doi.org/10.1016/j.jcss.2015.06.007




(a) "Why Empty?" query: response time

(b) "Why So Many?" query: response time

(c) "Why Empty?" query: number of MCS

(d) "Why So Many?" query: number of MCS

**Fig. 11.** Optimization of "Why Empty?" and "Why So Many?" queries.

**Table 4**
Scalability: data sets.

| Data set | Number of edges | Number of vertices | Number of attributes |
|---|---|---|---|
| D1 | 100*K* | 40*K* | 163 |
| D2 | 213*K* | 30*K* | 740 |
| D3 | 819*K* | 182*K* | 1542 |

results: higher response times with restart strategies and larger or the same size of result graphs. The restart strategy can deliver positive results also for the star topology, if the first processed edge was not suitable for the processing goals. For example, in Fig. 11(b) some strategies for a single run do not deliver any results, because chosen starting edges have higher cardinality than the expected number. The restart strategy compensates this issue.

Comparing the results of this evaluation, we conclude that with the restart strategy we can find larger common subgraphs without starting from each vertex. Our optimizations can reduce the number of intermediate results, the number of MCSs, and the response time. If characteristics of edges (degree, predicate) are similar, all strategies provide similar results. The strategies based on the cardinality can be even more efficient, if after an edge is selected, the direction of its processing is chosen according to the cardinalities of its adjacent vertices. In addition, for "Why So Many?" queries we see that the threshold for the maximum expected cardinality $CmaxExp = 1000$ was not violated. Regardless of any optimization, the cardinality of a result set stays below the threshold.

### 7.5. Scalability

In this evaluation, we study the scalability of subgraph-based solutions using the example of "Why Empty?" queries. For this test, we have created three data sets (see Table 4) from DBpedia data and a basic query with five configurations. This query consists of five edges and six vertices. The cardinalities of edges of the basic query $Q1$ for different data sets is provided in Table 5. We change a predicate on vertex $v_4$ of the basic query and present its changing cardinality over the data sets in Table 6.

As we can see in Fig. 12, the evaluation results depend strongly on the cardinalities of query elements. The response time of basic query $Q1$ in Fig. 12(c) increases with the size of a data graph due to increasing cardinalities of query edges (see Table 5) and increasing size of discovered subgraphs (see Fig. 12(b)).

The second part of the evaluation shows the dependency on changing predicates of vertex $v_4$ presented in Table 6. The size and number of MCSs decreases according to the changing cardinality of vertex $v_4$.

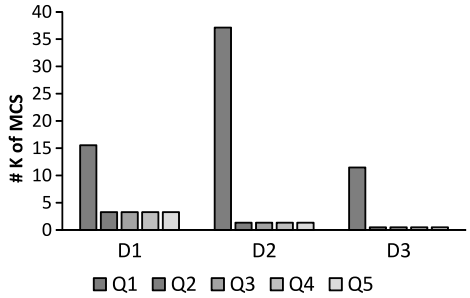
Final edited form was published in "Journal of Computer and System Sciences". 82 (1), S. 3-22. ISSN: 0022-0000.
https://doi.org/10.1016/j.jcss.2015.06.007

**Table 5**
Scalability: cardinality of a basic query.

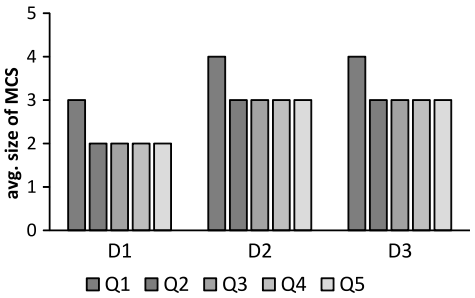
| Data set | $e_1$ | $e_2$ | $e_3$ | $e_4$ | $e_5$ |
|---|---|---|---|---|---|
| D1 | 0 | 0 | 1739 | 54378 | 1739 |
| D2 | 179 | 0 | 2263 | 147747 | 2263 |
| D3 | 180 | 0 | 14887 | 499620 | 14887 |

**Table 6**
Scalability: cardinality modifications for vertex v4.

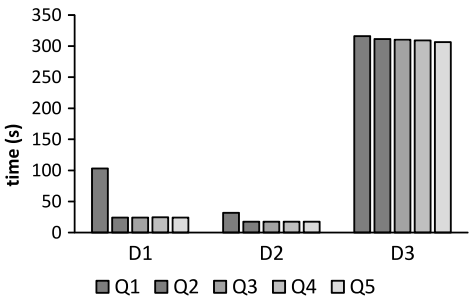
| Data set | Q1 | Q2 | Q3 | Q4 | Q5 |
|---|---|---|---|---|---|
| D1 | 40*K* | 0 | 1341 | 0 | 0 |
| D2 | 30*K* | 2519 | 1 | 56 | 391 |
| D3 | 182*K* | 2520 | 713 | 56 | 2952 |




(a) "Why Empty?" query: number of MCS




(b) "Why Empty?" query: size of MCS




(c) "Why Empty?" query: response time

**Fig. 12.** Scalability of "Why Empty?" queries.

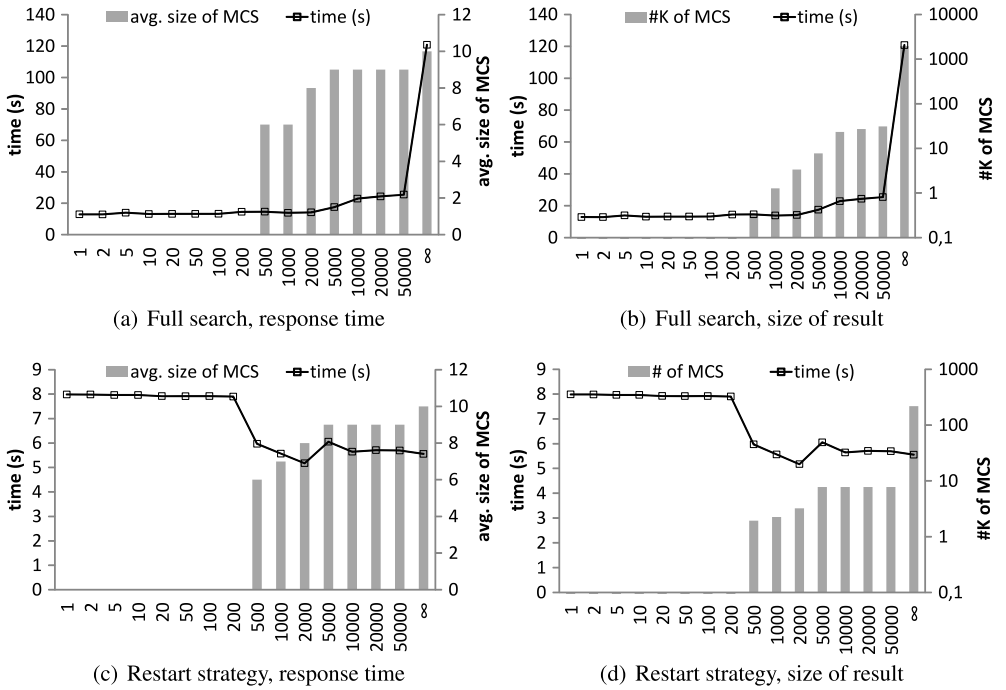
Comparing the results of this evaluation we can conclude, that our optimized subgraph-based solutions scales with an increasing size of a data graph. The response time and discovered subgraphs strongly depend on the minimal cardinality over all query edges. While with an increasing size of a data graph, the size of MCS can be higher, the restrictions on predicates can reduce it.

### *7.6. Maximum expected cardinality*

In this section we evaluate maximum cardinality thresholds for "Why So Many?" queries. For this purpose, we took the most typical topology for our data set: a star topology. Our query consists of nine edges of different semantic types. In this experiment we change the maximum expected cardinality from 1 to unlimited and conduct the query as a restart query with the maximum-cardinality threshold and as a full search from each edge. The experimental results are provided in Fig. 13.

As we can see from the evaluation results, the first possible threshold for the query that delivers a non-empty answer is 500. It can be explained by the general description of the query: only edges have a semantic description. We also observe that the full search can violate the set threshold. This effect can be explained by the following facts: a threshold is set separately for each run. Therefore, when all runs are conducted and duplicates are eliminated, the final number of results can exceed the expected cardinality. Therefore, to use the full search for "Why So Many?" queries the local thresholds have to be adjusted to a global threshold. The full search and restart strategy show that the response time depends on the number of discovered cardinality-bounded MCSs and their sizes. The restart strategy has higher response times until the

Final edited form was published in "Journal of Computer and System Sciences". 82 (1), S. 3-22. ISSN: 0022-0000.
https://doi.org/10.1016/j.jcss.2015.06.007




(a) Full search, response time (b) Full search, size of result

(c) Restart strategy, response time (d) Restart strategy, size of result

**Fig. 13.** Evaluation of maximum cardinality thresholds for "Why So Many?" queries.

cardinality threshold reaches the 500-th level. This is explained by the restart from each edge. Since a threshold meets at least one edge matching the cardinality threshold the response time reduces and follows the number of cardinality-bounded MCSs (see Fig. 13(d)).

### *7.7. Discussion*

The evaluation shows that the restart configuration and an all-covering spanning tree can be used without starting from each query edge. They facilitate to find larger maximum common unconnected subgraphs and at the same time reduce the response time. The presented optimizations can further decrease the response time, but at the expense of a lower number of MCSs.

## 8. Related work

In this section we discuss state of art approaches to handle "Why?" queries in relational database management systems and graph processing engines.

### *8.1. Problem of unexpected answers in relational database systems*

Classical data provenance approaches study how a particular output tuple was produced from the input data [14]. This research was extended by the question why particular elements are not represented in a result set. This problem of unexpected answers originates from the relational databases as "Why Not?" queries.

*"Why Not?" queries* If a result of a query does not meet the user expectations, a user can issue a "Why Not?" query [15] that determines why the result set does not include the items of interest. It is assumed here that a user cannot process the data manually because of their large volume and complexity. A user specifies items of interest with attributes or key values. Then a "Why Not?" query could be "Why are the items with predicate P not in the result set?"

"Why Not?" queries can be classified into two groups according to the kinds of explanations they provide: provenance-based and query rewriting. In the provenance-based methods, query-based [15,16], instance-based [17–19], and hybrid explanations [20] are generated. Query-based explanations [15] study which operators of a query tree are responsible for the rejection of interesting tuples from the result set. For this purpose, the authors apply a set of manipulations to the original query. Bidoit et al. [16] extend this approach by proposing a more efficient and correct solution for query-based answers. Instance-based solutions [17–19] explain how to change the data source to deliver missing answers. Such information can be used in data integration tasks, when an extraction process can fail and deliver wrong data or when information sources have different trust levels.

Final edited form was published in "Journal of Computer and System Sciences". 82 (1), S. 3-22. ISSN: 0022-0000.
https://doi.org/10.1016/j.jcss.2015.06.007

In the query rewriting research [21,22], an original query is refined in such a way, that the items of interest appear in the result set. In this case, the explanation for a "Why Not?" query represents a modified query [21], which has a response that consists of the original results and the items of interest. Islam et al. [22] transform a query so that the result set includes expected answers and that unexpected results are removed. This approach requires user interaction to provide at least a subset of tuples to be delivered.

In contrast, our problem investigates missing structural parts of a graph query that prevent a graph processing engine from delivering a non-empty result set. A relational "Why Not?" query studies the problem only on the level of a vertex and not on the level of a subgraph.

*"Why Empty?" queries* To solve the empty-answer problem, automatic and interactive solutions are proposed. In the first case, in automatic solutions, an original query is refined automatically by considering contextual information, preliminary information, or historical data. Extended queries [23] contain additional constraints for a result set that have to be guaranteed during the query rewriting. Each modification operation that can be applied to a query is limited by a set of its application conditions and rules. Query rewriting techniques for the empty-answer problem are also widely used in recommender systems [24]. In the second case, in interactive solutions, a user decides, which query candidate should be followed. An optimization-based interactive relaxation framework [25] dynamically proposes alternative queries with a reduced amount of constraints in comparison to the original query. The relaxation terminates if a proposed query candidate has delivered a non-empty result set or if a query candidate was found that delivers an empty result set and cannot be further relaxed. In contrast, we do not rewrite queries, but we deliver intermediate results of query processing and derive differential graphs that show missing query parts as the reason for an empty result set.

*"Why So Many?" queries* To solve the many-answer problem, two approaches are typically used: ranking and categorization. In the first case [26,27], the results are sorted according to a scoring function, and more interesting results are ranked higher. In such solutions, a suitable scoring function has to be defined to deliver results interesting to a user. The second case [28,29] is a data-driven approach: tuples are categorized into several groups and then proposed to a user for further refinement. For example, a cost-driven approach [29] based on a faceted navigation can be used. To reduce the number of answers, a user has to choose value conditions for rejection of particular facets. In our application scenario the use of faceted search would make the solution even more complicated because of the high number of vertices and connections between them and, as a consequence, a high number of possible facets. To rank the result, domain knowledge is required, which we do not model in our current research.

### *8.2. "Why?" queries in graph databases*

In our previous work [2,7] we have already proposed our solutions for "Why Empty?" and "Why Not?" queries in graph databases. A subgraph-based solution for "Why Empty?" queries is proposed as differential queries [2], where a user receives a list of maximum common unconnected data subgraphs and differential graphs. The goal of this query is to find the reasons of an empty answer in the form of subgraphs missing in a data graph. In our further work on top-k differential queries [7] we extended "Why Empty?" queries to "Why Not?" queries by considering a user's interest that is represented by relevance weights. With this kind of graph queries a user can annotate graph elements with relevance weights, showing their relevance to a user. These weights are used for several purposes. On the one hand, they manipulate the search of maximum common subgraphs and facilitate the early discovery of more interesting query parts to a user. On the other hand, they are used for the ranking calculation of already discovered answers. To the best of our knowledge, there is no further research work addressing "Why?" queries in graph databases.

## 9. Conclusion

To express graph queries correctly can be a difficult task, because of the diversity and schema flexibility of a data graph. If a query derives an empty result set, a user requires support to understand, what the reason was: either an overspecified query or missing data. For this case, we present in this paper our work on *"Why Empty?"* queries. The response to this query describes the parts of a graph query that are addressed and those that are missing in a data graph. The processing of a differential query consists of two steps: (1) the discovery of a maximum common subgraph of the query graph and the data graph implemented by the traversal-based algorithm GraphMCS originated from the McGregor's maximum common connected subgraph strategy and (2) the computation of a differential graph between the query graph and the discovered MCS.

If a query derives too many answers, a user requires support to understand, what the reason was: an underspecified query or correct data. In this paper, we introduce the idea of *"Why So Many?"* query, a new kind of graph queries, that support a user in such cases. The response to this query describes cardinality-bounded and unbounded parts of a graph query. Its processing consists of two steps: (1) the discovery of a cardinality-bounded maximum common subgraph by the join-based algorithm BoundedMCS and (2) the computation of a differential graph.

We adapt both algorithms for directed weakly-connected property graphs with an all-covering spanning tree and reduce the number of lookups with the restart strategy, which searches from a single edge and does restarts. We show that this

Final edited form was published in "Journal of Computer and System Sciences". 82 (1), S. 3-22. ISSN: 0022-0000.
https://doi.org/10.1016/j.jcss.2015.06.007

can be improved by the suitable choice of a start and restart vertex and edge. After the answer is delivered to a user, he can do explorative search of missing data in external sources for the empty-answer problem or modify the query according to the derived differential graph.

In addition, we classify unexpected answers and corresponding “Why?” query types and, to quantify the differential graphs, we introduce basic and complex operations as a means for calculating the difference between discovered result sets and an original graph query. With our evaluation we show that these queries can be efficiently processed by GraphMCS and BoundedMCS algorithms enhanced with optimization techniques and the all-covering spanning tree.

In the future, we want to investigate the solutions based on query rewriting for all four “Why?” queries. Moreover, we see the advantages of user interaction for such approaches and plan to enhance the proposed solution with a user provided domain knowledge for more guided subgraph search.

## Acknowledgment

This work has been partially supported by the FP7 EU project LinkedDesign (grant agreement no. 284613).

## References

[1] M.A. Rodriguez, P. Neubauer, Constructions from dots and lines, Bull. Am. Soc. Inf. Sci. Technol. 36 (6) (2010) 35–41.
[2] E. Vasilyeva, M. Thiele, C. Bornhövd, W. Lehner, GraphMCS: discover the unknown in large data graphs, in: Proceedings of the Workshops of the 2014 Joint Conference, EDBT/ICDT, 2014, pp. 200–207.
[3] M. Paradies, M. Rudolf, C. Bornhövd, W. Lehner, GRATIN: accelerating graph traversals in main-memory column stores, in: Proceedings of Workshop on GRAph Data Management Experiences and Systems, GRADES'14, ACM, New York, NY, USA, 2014, pp. 9:1–9:6.
[4] C. Bornhövd, R. Kubis, W. Lehner, H. Voigt, H. Werner, Flexible information management, exploration and analysis in SAP HANA, in: DATA, 2012, pp. 15–28.
[5] M. Rudolf, M. Paradies, C. Bornhövd, W. Lehner, The graph story of the SAP HANA database, in: Proceedings, Datenbanksysteme für Business, Technologie und Web (BTW), 15. Fachtagung des GI-Fachbereichs “Datenbanken und Informationssysteme”, DBIS, 2013, pp. 403–420.
[6] X. Gao, B. Xiao, D. Tao, X. Li, A survey of graph edit distance, Pattern Anal. Appl. 13 (1) (2010) 113–129.
[7] E. Vasilyeva, M. Thiele, C. Bornhövd, W. Lehner, Top-k differential queries in graph databases, in: Advances in Databases and Information Systems, in: Lect. Notes Comput. Sci., vol. 8716, Springer International Publishing, 2014, pp. 112–125.
[8] J.R. Ullmann, An algorithm for subgraph isomorphism, J. ACM 23 (1) (1976) 31–42.
[9] J.J. McGregor, Backtrack search algorithms and the maximal common subgraph problem, Softw. Pract. Exp. 12 (1) (1982) 23–34.
[10] T. Cormen, C. Leiserson, R. Rivest, C. Stein, Introduction to Algorithms, MIT Press, 2001.
[11] P.J. Durand, R. Pasari, J.W. Baker, C.-c. Tsai, An efficient algorithm for similarity analysis of molecules, Internet J. Chem. 2 (17) (1999) 1–16.
[12] E. Balas, C.S. Yu, Finding a maximum clique in an arbitrary graph, SIAM J. Comput. 15 (4) (1986) 1054–1068.
[13] D. Conte, P. Foggia, M. Vento, Challenging complexity of maximum common subgraph detection algorithms: a performance analysis of three algorithms on a wide database of graphs, J. Graph Algorithms Appl. 11 (1) (2007) 99–143.
[14] J. Cheney, L. Chiticariu, W.-C. Tan, Provenance in databases: why, how, and where, Found. Trends® Databases 1 (4) (2009) 379–474.
[15] A. Chapman, H.V. Jagadish, Why not?, in: Proceedings of the 2009 ACM SIGMOD International Conference on Management of Data, SIGMOD'09, ACM, 2009, pp. 523–534.
[16] N. Bidoit, M. Herschel, K. Tzompanaki, Query-based why-not provenance with NedExplain, in: Proceedings of the 17th International Conference on Extending Database Technology, EDBT, 2014, pp. 145–156.
[17] J. Huang, T. Chen, A. Doan, J.F. Naughton, On the provenance of non-answers to queries over extracted data, Proc. VLDB Endow. 1 (1) (2008) 736–747.
[18] M. Herschel, M.A. Hernández, Explaining missing answers to SPJUA queries, Proc. VLDB Endow. 3 (1–2) (2010) 185–196.
[19] M. Herschel, M.A. Hernández, W.-C. Tan Artemis, A system for analyzing missing answers, Proc. VLDB Endow. 2 (2) (2009) 1550–1553.
[20] M. Herschel, Wondering why data are missing from query results?: ask conseil why-not, in: Proceedings of the 22nd ACM International Conference on Conference on Information and Knowledge Management, CIKM'13, ACM, 2013, pp. 2213–2218.
[21] Q.T. Tran, C.-Y. Chan, How to ConQueR why-not questions, in: Proceedings of the 2010 ACM SIGMOD International Conference on Management of Data, SIGMOD'10, ACM, New York, NY, USA, 2010, pp. 15–26.
[22] M.S. Islam, C. Liu, R. Zhou, On modeling query refinement by capturing user intent through feedback, in: Proceedings of the Twenty-Third Australasian Database Conference, ADC'12, vol. 124, Australian Computer Society, Inc., Darlinghurst, Australia, Australia, 2012, pp. 11–20.
[23] S. Chaudhuri, Generalization and a framework for query modification, in: Proceedings of the Sixth International Conference on Data Engineering, IEEE Computer Society, Washington, DC, USA, 1990, pp. 138–145.
[24] D. Jannach, Techniques for fast query relaxation in content-based recommender systems, in: Proceedings of the 29th Annual German Conference on Artificial Intelligence, KI'06, Springer-Verlag, Berlin, Heidelberg, 2007, pp. 49–63.
[25] D. Mottin, A. Marascu, S.B. Roy, G. Das, T. Palpanas, Y. Velegrakis, A probabilistic optimization framework for the empty-answer problem, Proc. VLDB Endow. 6 (14) (2013) 1762–1773.
[26] S. Chaudhuri, G. Das, V. Hristidis, G. Weikum, Probabilistic information retrieval approach for ranking of database query results, ACM Trans. Database Syst. 31 (3) (2006) 1134–1168.
[27] S. Chaudhuri, G. Das, V. Hristidis, G. Weikum, Probabilistic ranking of database query results, in: Proceedings of the VLDB Endowment, 2004, pp. 888–899.
[28] S. Basu Roy, H. Wang, G. Das, U. Nambiar, M. Mohania, Minimum-effort driven dynamic faceted search in structured databases, in: Proceedings of the 17th ACM Conference on Information and Knowledge Management, CIKM'08, ACM, New York, NY, USA, 2008, pp. 13–22.
[29] A. Kashyap, V. Hristidis, M. Petropoulos, Facetor: cost-driven exploration of faceted query results, in: Proceedings of the 19th ACM Conference on Information and Knowledge Management, ACM, 2010, pp. 719–728.