Faculty of Computer Science
Database Technology Group

Kurzfassung der Dissertation
zur Erlangung des akademischen Grades
Doktoringenieur (Dr.-Ing.)

Why-Query Support in Graph Databases

vorgelegt an der Technischen Universität Dresden
Fakultät Informatik

eingereicht von
MSc. Elena Vasilyeva
geboren am 10.April 1986 in Essoila

Gutachter: **Prof. Dr.-Ing. Wolfgang Lehner**
Technische Universität Dresden
Fakulät Informatik, Institut für Systemarchitektur
Lehrstuhl für Datenbanken
01062 Dresden, Deutschland

Associate Prof. Dr.-Ing. Katja Hose
Aalborg University
Department of Computer Science
9220 Aalborg, Dänemark

Fachreferentin: **Prof. Dr. Christel Baier**
Technische Universität Dresden
Fakulät Informatik, Institut für Theoretische Informatik
Lehrstuhl für Algebraische und Logische Grundlagen der Informatik
01062 Dresden, Deutschland

Eingereicht im August 2016

1 Introduction

In the last few decades, databases turned into powerful systems with a broad range of functionality. While the race for new functionality and effective algorithms gains momentum, the complexity of querying becomes an issue reducing their usability. This issue has been extensively studied by Jagadish et al. [5], who defined four classes of what users expect from databases: sophisticated querying, precise and complete answers, structured results, and creation and update of the databases. Any contradiction of these expectations with the system behavior can frustrate users and distract them from querying. One of the usability issues discussed by Jagadish et al. is the presence of unexpected pains especially in terms of query results. If results differ from expectations, users have to conduct try-and-error rewriting for failed queries to figure out what was wrong. To support them in such situations, the database systems should be able to provide explanations for unexpected results.

A group of database researchers came to the same conclusion in the Beckman report [1]: *"Explanation, provenance . . . crop up in all steps of the raw-data-to-knowledge pipeline. They will be critical to making analytic tools easy to use. . . . We must build tools and infrastructures that make the data consumption process easier, including the notions of trust, provenance, and explanation . . . "*

The problem of unexpected results can arise in multiple scenarios involving databases from the keyword search over relational data to online form-based querying of web databases. In all cases, users can wonder why results differ from their expectations. Is the query wrong? Is some data in the database missing? Or is the answer correct? To assist users in such situations, queried systems should be able to give explanations about the reasons of unexpectedness and rewrite failed queries if necessary. This functionality makes database systems more user-friendly and attractive especially for inexperienced users.

In the related work, the problem of unexpected results is investigated in the form of why-queries, which answer the question why retrieved result sets differ from user expectations. Why-queries consider several types of unexpected results such as absence of expected answers [2], presence of unexpected results [14], empty [9], too few [12], or too many answers [8]. Why-queries can be classified into content-based and cardinality-based ones depending on the problems they solve. The first group investigates missing expected and presenting unexpected results and is performed by why-not and why-so queries. The second group inspects why the result size differs from an expected cardinality and explores empty-answer, too-few- and too-many-answers problems in the form of why-empty, why-so-few, and why-so-many queries.

Why-queries provide explanations for unexpected results and thus improve the usability of databases. However, most related work investigates this problem in RDBMS and only limited attention is paid to other data models. For example, why-so queries are investigated in the object-relational setup [15], why-not pattern matching is proposed for multiple labeled graphs [4], why-empty queries can be defined for data in the RDF format (Resource Description Framework) [11], and why-so-many queries consider richly attributed graphs [10].

Referring to the usability study by Jagadish et al. [5], also new database types arise including triple stores, NoSQL, etc. Graph databases implementing the property-graph model belong to this novel development branch. They allow to store heterogeneous information in the form of a directed multi-graph, where entities are represented by vertices and edges describe relationships between them. Both edges and vertices can be annotated with multiple diverse attribute values. These databases become powerful tools allowing to keep information without defining a rigid schema and process complex analytical queries based on graph algorithms. The property-graph data model is implemented by modern graph databases such as NEO4J, SAP HANA, and ORACLE BIG DATA SPATIAL AND GRAPH.

The usability issue described above becomes even more complicated in graph databases. On the one hand, there is no rigid schema, which would help to construct queries. On the other hand, defining correct queries becomes even more complicated for graph queries containing multiple query constraints. Therefore, the usability issue requiring explanations as described by Jagadisch et al. [5] for query results is of high priority for graph databases. Considering the lack of research for graph databases and the complexity of graph queries, we focus on providing debugging support for graph queries over property graphs in a form of why-queries in this thesis.

Depending on the type of a query answer, two groups of graph queries can be distinguished: queries that deliver data subgraphs and queries that provide simple answers consisting of a Boolean value or a number. The first group comprising community-detection algorithms, pattern matching, and traversal queries represents the most general query types which can suffer from all kinds of unexpected results investigated by why-queries. The second group including shortest-path and reachability queries is typically affected by content-based problems. In this thesis, we focus on the first group of queries and consider as an example pattern-matching queries that return data subgraphs matching the query graph. These are high-constrained queries, where constraints are represented not only by predicates for attribute values but also by the query topology. For such queries, diverse explanations can be generated to resolve the same problem, where some of them can be irrelevant to a user and should be avoided. Therefore, generation of explanations has to be able to consider user interest in specific query constraints in order to deliver meaningful explanations. To summarize, in this thesis we investigate how to generate explanations for no, too few, or too many answers and support user-integration strategies in graph databases implementing the property-graph model.

1.1 Contributions

In order to increase the usability of graph databases via result explanations, we propose cardinality-based why-queries for pattern matching and do the following contributions in this thesis:

1. **Extensive study of state-of-the-art debugging approaches for why-queries.** We survey recent literature on methods for why-queries in different data models, classify them based on the explanation types they provide, and extract typical features for each specific why-query type. These aspects are used to compile a list of requirements that have to be fulfilled by the debugging tool for pattern matching queries over property graphs.

2. **Comparison metrics for explanations.** Each debugging method can provide several explanations. To support a user with the most appropriate explanations, we propose a comprehensive comparison on three levels: query, cardinality, and result levels. On the query level, we use a syntactic distance which describes how different two queries appear to a user. On the cardinality level, we analyze how the result size differs from an expected cardinality. On the result level, the content of results for compared explanations is tested against the result set of the failed query. This comprehensive analysis allows to fairly judge the proposed methods.

3. **Subgraph-based explanations.** The first question that has to be answered during debugging is why the query failed to deliver the expected results. Referring to the related work on why-not queries [2], we propose to generate a query-based explanation called a subgraph-based explanation by traversing a query graph and detecting a failed query part. For this purpose, we develop two algorithms DiscoverMCS and BoundedMCS for why-empty queries and why-so-few and why-so-many queries, respectively. To reduce the amount of large intermediate results and to improve the performance of these algorithms, we provide several optimization techniques, which choose a traversal path along the query and reduce the number of traversals.

4. **Modification-based explanations for why-empty queries.** Most approaches from the related work try to remove the burden of query rewriting from the users and produce refined non-failed queries. Following the same motivation, we propose a coarse-grained solution for rewriting why-empty pattern-matching queries considering modifications of topology and predicates to derive non-empty results. This approach does not consider the cardinality threshold and therefore is more appropriate for solving why-empty queries. For efficiency reasons, we further investigate caching of already processed queries and their re-use. We also describe several techniques to calculate and estimate cardinalities for parts of the original query.

5. **Modification-based explanations for why-so-few and why-so-many queries.** While why-empty queries can be rewritten by discarding some query parts, why-so-few and why-so-many queries require a fine-grained model for query modification, because any change should deliver specific cardinality improvement, which depends on a given cardinality threshold. Considering this fact, we propose the TraverseSearchTree method for modifying queries delivering unexpected cardinality and allow fine-grained modifications on the predicate level. This method constructs a modification tree at runtime, optimizes it by rejecting and re-arranging its branches, and guarantees the propagation of changes along the query.

6. **User-integration models.** To generate user-relevant explanations, we discuss how user interest in specific query parts can be incorporated in the generation process. We propose two user-integration models, namely: one for subgraph-based and one for modification-based explanations. The first model derives the most-relevant traversal path, which is adapted online during processing. This approach can also be re-used in modification-based explanations for why-so-few and why-so-many queries as a strategy for re-arranging modification-tree branches. The second model constructs a user-preference model during rewriting of why-empty queries and adapts the modification process accordingly.

In the following sections, we will describe these contributions in detail. In Section 2, we introduce the property-graph model and general debugging features extracted from the related work in order to enable why-queries in graph databases.

2 Why-Queries in Graph Databases

In this section, we introduce general debugging features extracted from the related work in order to deliver why-queries in graph databases implementing the property-graph model. As an underlying graph model, we use a property-graph model [13]

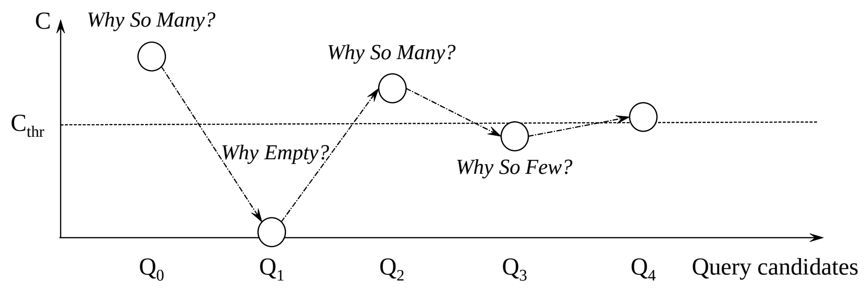


Figure 1: Holistic support of different cardinality-based problems

as one of the most general graph models, which is commonly used in modern graph databases like NEO4J, SAP HANA, SPARKSEE, and ORACLE BIG DATA SPATIAL AND GRAPH. A property graph is a general graph model representing data in a form of a directed graph, where vertices are entities and edges are relationships between them. Multiple edges can exist between the same vertices. Each vertex and edge can be annotated with properties, which are represented by key-value pairs. On the one hand, it allows storing and processing complex graph relationships with multiple diverse properties without a rigid schema. On the other hand, it provides us with the opportunity to conduct complex graph queries over data.

In this thesis, we investigate a pattern matching query that is a fundamental graph query. It describes a pattern to be discovered in a large data graph and retrieves data subgraphs matching to it. A pattern itself represents a property graph, where edges and vertices are defined with predicates for their properties.

To implement cardinality-based why-queries in graph databases, the following debugging features have to be investigated: holistic support of different cardinality-based problems, comprehensive comparison of explanations, explanation of unexpected results, query reformulation, and non-intrusive user integration.

Holistic Support of Different Cardinality-Based Problems During the debugging and query rewriting, the size of the result can oscillate around the cardinality threshold as presented in Figure 1, where the original query delivers too many results. Its first refinement Q_1 delivers no result, second one Q_2 retrieves too many, and the third one Q_3 results in too few answers. In this case, the system has to be able to automatically adapt the direction of the search and to provide holistic support for different cardinalities as illustrated in this figure such that finally Q_4 delivers expected answers. For dealing with too few or too many results, a cardinality threshold C_{thr} is required. Based on the result size and a given cardinality threshold, a debugging tool should automatically decide which query has to be executed: a why-empty, why-so-few, or why-so-many query.

Explanation of Unexpected Results and Query Reformulation One of the core functionalities extracted from the state-of-the-art systems represents discovering the reasons of unexpectedness and query refinement. This feature is represented by two kinds of explanations such as query-based and modification-based explanations. The first one describes why the query fails to deliver expected results in terms of a query part which violates a cardinality constraint. Referring to the fact that a query in a graph database implementing the property-graph model represents a property graph itself, a query-based explanation is a subgraph-based one that describes which part of a query graph in terms of its topology is responsible for an unexpected result. The second type of explanations, modification-based explanations, delivers a refined query that was modified in such that it delivers results closer to the cardinality threshold compared with the result of the original query. By modifying a query, its predicates and topology can be changed. This is the most important functional property, which the debugging tool should provide.

Comprehensive Comparison of Explanations Deriving an explanation for a query delivering unexpected results depends on how a query is traversed and modified. Multiple explanations can be generated for the same query. To choose the best explanation, we propose a comparison strategy, which considers specifics of the property-graph model and integrates two aspects including query topology and predicates. It compares explanations according to three characteristics: the content and the size of the results and the syntactic difference between explanations.

Syntactic Distance This metric shows how familiar an explanation appears to a user in reference to an original query. In order to calculate it as a set distance, we represent an explanation and original query as sets consisting of vertices and edges, which are sets of predicates, types, directions, in- and out-edges. The set-based representation of a graph query has the advantage of expressing the syntactic dissimilarity of two graph queries by a set-based distance and using well-studied set-based measures from related work. To syntactically compare two explanations, the set distance has to be calculated between them from the set distances of their subsets, which are derived according to the modified Hausdorff distance [3]. The proposed metric provides a fine-granular comparison in contrast to the graph edit distance. This is the first metric to qualify explanations. However, it does not describe how good explanations are in order to deliver a required cardinality.

Cardinality Distance The second level of comparing two explanations is the sizes of their results. Therefore, we calculate a cardinality distance that describes which explanation is closer to the cardinality threshold. For too-many- and too-few-answers problems, this investigation can be easily done by calculating the deviation of the result size from the cardinality threshold. For the empty-answer problem, the cardinality threshold is not given. The only known fact is that the result should include at least some answers. Therefore, the queries with smaller results are preferred.

Result Distance The comparison of the result content shows how the results of the compared explanation differ from the results of an original query, which can be derived from their overlapping parts. This measure allows to compare how many new results are introduced or how many answers are removed from the original result. It can be defined only for problems, where an original query delivers at least some answers.

To calculate a distance between two result sets, we calculate distances between each pair of data subgraphs as a graph edit distance including deletion, insertion, and substitution of vertices and edges. After calculating the distance for each pair of result graphs, a total distance between two result sets can be derived. We model this comparison as the maximum generalized assignment problem [7], where workers are the result graphs of the first result set and tasks are the result graphs of the second result set. Any worker can be assigned to any task, this assignment is characterized by its cost, which is modeled as a distance between two result graphs. The assignment of all workers to tasks aims at maximizing the profit and therefore at minimizing the costs of assignments, which means that the workers have to be assigned in such a way that overall assignment costs (total dissimilarity) are minimal. We assign result graphs by the Hungarian-based algorithm [6].

Non-Intrusive User Integration One additional aspect, which has to be considered during the development of a debugging tool, is user intention. Without its consideration, a user can potentially receive a non-interesting explanation. Therefore, the debugging tool should be able to integrate user interest into a debugging process, which can be realized for example as importance of some query subgraphs. Considering a graph query as a highly constrained problem, it is necessary to not overwhelm a user with decisions on how to process a query. User interest in a particular query subgraph has to be derived and configured based only on a feedback, showing how important, (ir)relevant some aspects of a query are to the user.

In this section, we discussed the core functionality that has to be considered in order to support cardinality-based why-queries in graph databases, namely holistic support of different cardinality-based problems, explanation of unexpected results and query reformulation, comprehensive comparison of explanations, and non-intrusive user integration. Before describing these features, we introduced the property-graph model and supported graph-query types. We also discussed in detail one of these properties, comprehensive comparison of explanations, and proposed three similarity metrics, namely: a syntactic difference between an original query and an explanation, size and content of result sets in reference to a cardinality threshold and a result set of an original query. The evaluation revealed that the syntactic and result distances are slightly correlated such that stronger differences in a query description lead to larger result distances. Moreover, the result distance is influenced by the number of results of the original query. By relaxing a query, the result information will be reduced, which leads to higher result distances. By extending the query, the original information is still presented in the result of the modified query, and therefore, the result distance is small. These measures are used in this thesis for quantifying the quality of generated explanations.

In the following sections, we will consider core debugging features: how to discover the reasons of a query failure and how to rewrite a query in order to deliver better results.

3 Explaining Unexpected Results

One of the important properties in the set of extracted features refers to the core debugging functionality, explanation of unexpected results and query reformulation. In this section, we start description of this property and reveal its first requirement, explanation of unexpected result. By dealing with graph queries, the reason of unexpectedness can be described in terms of a query subgraph, which is responsible for violation of a cardinality constraint. Therefore, we call this type of explanations *subgraph-based explanations*. For why-empty queries, a responsible subgraph is not represented in a data graph. For why-so-few or why-so-many queries, this subgraph forces a result cardinality to drop below or to exceed a cardinality threshold. To generate subgraph-based explanations, we propose two algorithms, DiscoverMCS and BoundedMCS, and several optimization techniques, which allow to improve the quality of explanations and performance of their generation. We also consider a model for integrating user preferences in generation of subgraph-based explanations, which allows to derive preference-aware explanations.

To understand, which part of a query corresponding to a cardinality constraint, can be found in a data graph and which part violates a cardinality constraint, the *maximum common connected subgraphs* (MCSs) between data and query graphs must be found and their differential graphs must be calculated. For too-many- and too-

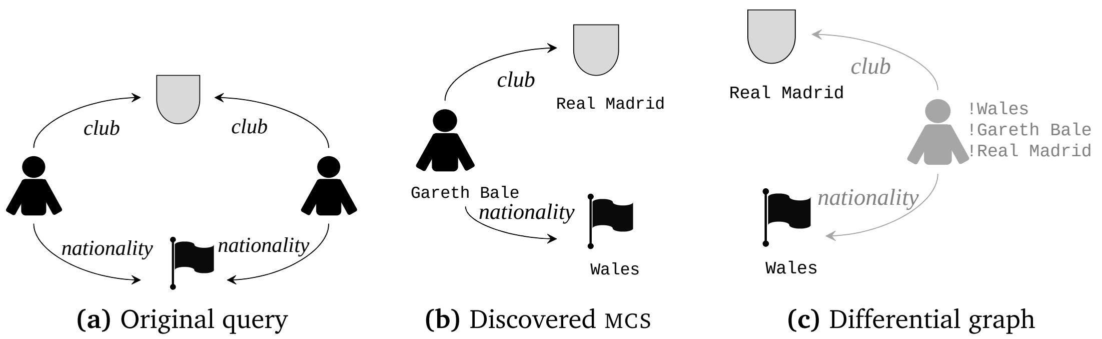


Figure 2: Original query delivering empty result and its subgraph-based explanation: which two vertices are from same country and play in same club?

few-answers problems, a maximum common connected subgraph is represented by a *cardinality-bounded* MCS describing such query subgraph, which does not violate a cardinality constraint. Cardinality-bounded maximum common connected subgraphs (BMCSs) show discovered partial results, before the resulting cardinality has exceeded or dropped below the cardinality threshold.

The DISCOVERMCS Algorithm To discover the reason of the empty answer, it is necessary to determine maximum existing and minimum failing query parts. The first part, a maximum common subgraph, represents that part of the graph query, which exists in the data graph. The second part, a minimum failing query, describes the missing query part. To determine them, we propose DISCOVERMCS that conducts a depth-first search along the query and considers only those data edges and vertices, which match the query predicates. To ensure that the algorithm finds all MCSs, it can be launched from all query vertices as multiple starting points and traverse all possible edge sequences. The discovery process continues until all investigated data subgraphs are processed and marked as complete. Afterwards, differential graphs can be calculated for them.

Example Assuming we search for two soccer players originating from the same country and playing in the same club as illustrated in Figure 2a. A possible answer to this why-empty query would consist of an MCS as shown in Figure 2b, and a differential graph as depicted in Figure 2c, which is the missing part of the query with constraints. 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 *Real Madrid* and *Wales*, missing query vertices and edges (gray), and constraints for vertices (gray).

To conclude, the DISCOVERMCS algorithm considers discovered data subgraphs individually and traverses them differently from each other. As a result, discovered data subgraphs can vary in size and the algorithm outputs only the largest one. This property does not allow to re-use this algorithm for why-so-few and why-so-many queries, which have to consider the number of output data subgraphs, which have to be isomorphic to the same query subgraph. Therefore, in the following section we present the BOUNDEDMCS algorithm, which overcomes these drawbacks and delivers BMCS with respect to the given cardinality threshold.

The BOUNDEDMCS Algorithm To discover the reasons of too few or too many answers, it is necessary to determine maximum cardinality-compliant and minimum cardinality-violating query parts. The first part, a maximum cardinality-compliant subgraph, describes that query part called BMCS which delivers less or more data subgraphs in reference to a cardinality threshold for the too-many- and too-few-answers problems. The second part, a minimum cardinality-violating subgraph, represents that query part which makes the number of data subgraphs violate the cardinality threshold.

Similar to DISCOVERMCS, the discovery process considers only those data vertices and edges which match query predicates. To ensure that the algorithm finds BMCSs, it is launched from all query vertices as multiple starting points. The core of the algorithm extends each discovered data subgraph with a new edge. If the size of extended set of discovered data subgraphs satisfies the cardinality constraint, then the edge is accepted and the traversal process continues. Otherwise, the edge is rejected and the last extension of data subgraphs is undone. After conducting the discovery process for each start vertex, the largest BMCSs are stored and delivered to a user.

DISCOVERMCS and BOUNDEDMCS have similar properties and therefore the same optimizations are applied for them such as online selection of a traversal path, construction of a spanning tree, and search restart. The first optimization, online selection of a traversal path, determines a next edge and vertex to process at runtime, which allows to traverse the query graph only once and to prevent discovery of duplicated MCSs (BMCSs). The evaluation results show the advantageous of using minimal-cardinality strategy for choosing a traversal path, which delivers the best results among all evaluated queries. To consider user intention in specific query elements, a heuristic for integrating user interest is discussed which can be used as a strategy for selecting a traversal path. The second optimization, construction of an all-covering spanning tree, allows to discover larger MCSs (BMCSs) by considering a query graph as weakly-connected. According to the experiments, it can improve the quality of detected MCSs and deliver larger subgraphs. The third optimization, restarting the algorithms, facilitates the discovery of MCSs (BMCSs) if an investigated query is split in several unconnected components during the search. The evaluation shows that restarting the search can facilitate the discovery of larger MCSs if the query was split in several unconnected components by the traversal strategy. It also can compensate the quality reduction injected by a used traversal strategy.

Subgraph-explanations are query-based explanations, which focus on the topology of the query and deliver MCSs. They can be used as an upper limit for rewriting an originally failed query. Still, this explanation does not give any detail about query predicates and defined types and therefore in the following more fine-granular explanations will be presented, which generate rewritten queries by considering also changes for types and predicates on query vertices and edges.

4 Coarse-Grained Why-Empty Query Modification

In the previous section, we proposed to explain reasons of a query failure in terms of its subgraph, which violates a cardinality constraint. This solution is based on the query topology, where the smallest element is represented by a vertex or an edge.

In this section, we go one step further and consider specifics of the property-graph model, namely support of predicates for attribute value on edges and vertices. We propose a next debugging step: a rewriting method for why-empty queries that introduces a new granularity level represented by predicates, types, and directions in addition to vertices and edges. In general, this approach generates modification-based explanations by removing some constraints of the failed query. Potentially, this method can also be used for solving too-few- and too-many-answers problems. However, it does not have a strong focus on the value of a cardinality threshold and therefore its use for these problems is rather suboptimal.

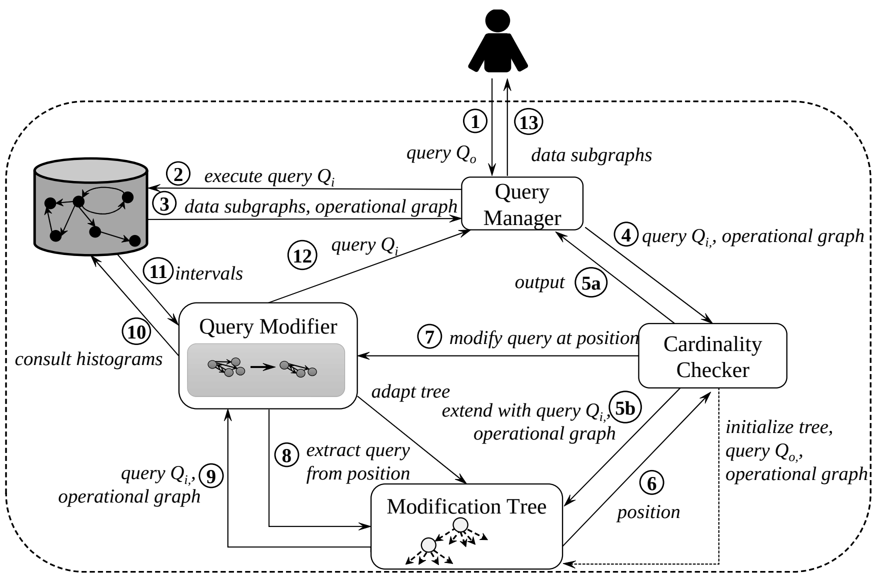

1 query Q_o
13 data subgraphs
2 execute query Q_i
3 data subgraphs, operational graph
Query Manager
4 query Q_i, operational graph
12 query Q_i
11 intervals
output 5a
10 consult histograms
Query Modifier
7 modify query at position
Cardinality Checker
adapt tree
extend with query Q_i, 5b operational graph
initialize tree, query Q_o, operational graph
8 extract query from position
query Q_i, 9 operational graph
Modification Tree
6 position

Figure 3: System architecture for why-empty query rewriting

If a query delivers no results, it is probably over-constrained with topological or attribute predicates, whose removal can potentially change the query in such a way that it delivers at least a few answers. This observation is taken in consideration by constructing the why-empty system illustrated in Figure 3 that is an extension of a graph database with the why-empty engine, which is activated by the user if an empty result set was delivered to his query. The relaxation process is managed by the *query manager* that receives user queries and redirects them to a graph database. If no data subgraphs match a request the relaxation process is triggered, which starts with initializing the *cardinality estimator* that maintains, calculates, and estimates cardinalities for cardinality queries. It considers cardinalities of vertices, edges, and paths up to size n. At the initialization, the cardinality estimator collects some cardinalities from the graph database and store them in the query-dependent statistics that is its core data structure allowing efficient storage of cardinalities. It keeps query-specific information such as the number of edges, vertices, predicates, which the original query has, and general statistics about the data graph such as the number of edges, vertices, and the vertex-edge mapping, i.e., source and target vertices of data edges, which are represented by their data identifiers.

After the query-dependent statistics has been initialized, the query manager triggers relaxation of a failed query, which is conducted by the *query relaxer* that generates multiple query candidates from it. This step is optimized with applying different relaxation strategies reducing the size of candidate space by choosing the most promising vertex and edge to be relaxed. Produced query candidates are transmitted to the *candidate selector* that stores them in a priority queue and provides the most promising query candidate to the query manager by request. The prioritization of query candidates is a crucial point in the rewriting process and is based on two criteria. The first criterion compares average path cardinalities of both queries and sorts them in a descending order dependent on them. In case the average path cardinalities are equal the queries are ordered according to the second criterion that considers the relative cardinality change induced by applied relaxations and shows how strong the cardinality change is in reference to the original query. A higher induced cardinality change describes stronger modifications, which may eliminate unique information from the query. Therefore, the candidate with a lower induced cardinality change is placed ahead in the queue.

After a new query candidate has been placed in the priority queue, the best query candidate is extracted from it and executed in the graph database. If this candidate failed to deliver a non-empty result the query manager forwards it to the query relaxer for its further relaxation. The process terminates if a query delivering a non-empty answer is found.

The proposed modification process is based on an A*-search, where in each step a set of query candidates is generated and stored in a priority queue of candidates. Those query candidates are tested first on the delivery of non-empty results which appear to be more promising to lead to a non-empty answer.

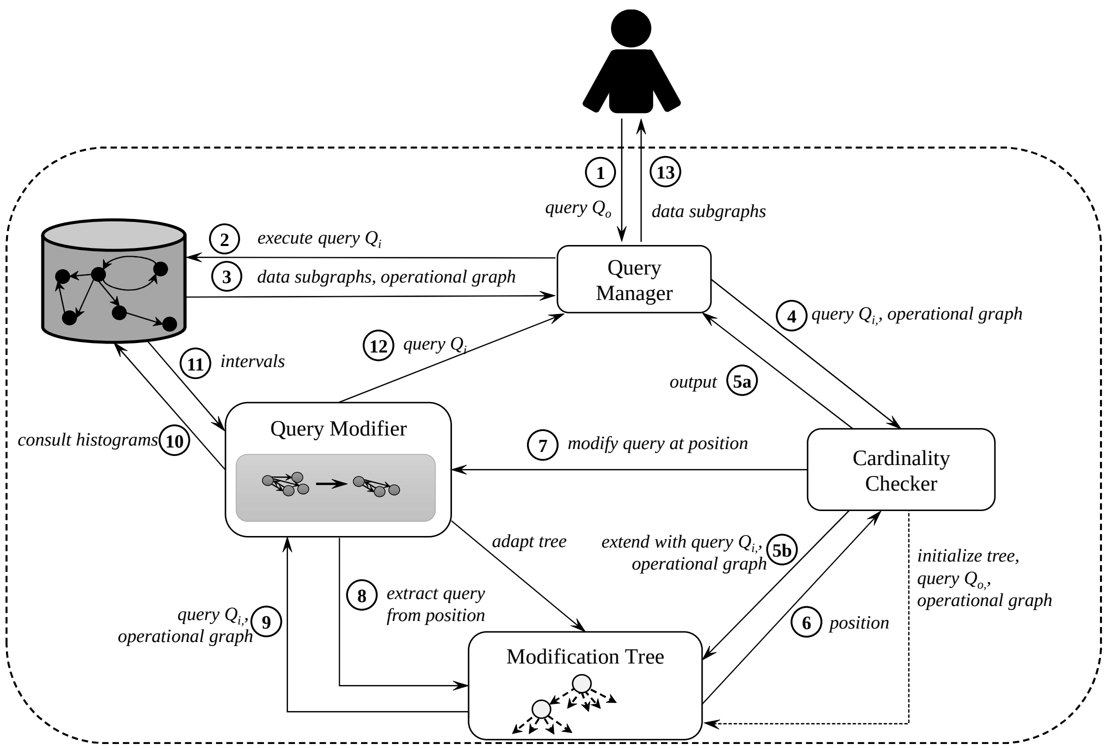
Without considering user interest during relaxation, we can miss more interesting explanations for a user and deliver non-interesting relaxed queries. To prevent deletion of sensitive information from the query, we incorporate user interest in specific query elements in the relaxation process as follows. After a user received an empty-answer, he triggers the why-empty engine to reformulate the original query. At each iteration, the produced query candidates are stored in the priority queue. The best candidate is extracted from the queue and tested on the delivery of any result. After a query solution with a non-empty result set is found the user can accept or reject it. For each rated solution, a user-preference model is calculated and incorporated in the overall preference model, which is valid only for relaxation of a specific query. All model changes are directly considered by the relaxation process. Any heuristic based only on the cardinality tends to follow a few branches from the relaxation tree. Therefore, after the calculating a user-preference model, we restart the process in order to induce the search along non-used relaxation branches according to the user interest. The search can be terminated if the desired query refinement is found, the number of iterations exceeds a predefined threshold, or no better query proposal can be found.

In this section, we proposed the why-empty engine for rewriting queries delivering empty results. The presented system conducts an A*-search, where new candidates are generated from the failed query by removing vertices, edges, and their properties. The refined queries are stored in a priority queue. The most promising query candidate extracted from the queue is checked on the delivery of a non-empty answer. If the new query failed it is relaxed and the process is repeated until a non-failed rewritten query is discovered.

The why-empty engine can also be potentially used for answering why-so-few and why-so-many queries by removing the most specific or general query parts, correspondingly. However, it has been optimized for why-empty queries and does not have a strong focus on a specific cardinality threshold. In addition, this system provides a coarse-grained rewriting and does not consider specific predicate changes. Therefore, in the following we go one step further and consider fine-grained changes in attribute description of a failed query for why-so-few and why-so-many queries.

5 Fine-Grained Cardinality-Driven Query Modification

In this section, we go one step further and propose a method for generating topology- and predicate-aware modification-based explanations, which considers the cardinality threshold and supports fine-grained topological and predicate changes. It generates explanations for both too-few- and too-many-answers problems. In addition to this approach, we introduce two new concepts, which are extensively used by the proposed method: an operational representation of a graph query and a modification tree for storing executed changes.



1 query Q_o
13 data subgraphs
2 execute query Q_i
3 data subgraphs, operational graph
Query Manager
4 query Q_i, operational graph
12 query Q_i
11 intervals
output 5a
consult histograms 10
Query Modifier
7 modify query at position
Cardinality Checker
adapt tree
extend with query Q_i, operational graph 5b
initialize tree, query Q_o, operational graph
8 extract query from position
query Q_i, operational graph 9
6 position
Modification Tree

Figure 4: Modification process for why-so-few and why-so-many queries

.

The rewriting procedure is presented in Figure 4. It is maintained by the *query manager*, which receives an original query from a user and redirects it to a graph database that executes it and returns matching data subgraphs together with the operational graph of a query. An operational graph is a representation of a query graph that describes how a query has been processed and is annotated with corresponding cardinalities. We use this model as a base for query modification in order to discover a query, which delivers results of a better cardinality.

After a query has been executed and its operational graph has been returned, the *cardinality checker* tests the size of a given result set and takes a decision, whether the query has to be modified. In case the result corresponds to a cardinality threshold, data subgraphs can be delivered to a user and the modification process terminates. Otherwise, the *modification tree* is initialized that is a data structure for tracking applied changes to a query. The root of this tree describes the initialization of the modification process and therefore it keeps the operational graph of an original query. Each node of the modification tree denotes a single iteration in the modification process and corresponds to a single change applied to a query of a parental node. Along each tree branch, every operator of an operational graph appears only once. Nodes of a modification tree keep the following information: a modified operator from an operational graph, a corresponding refined query with its operational tree. Traversing a modification tree top-down, we can collect all modifications and track corresponding cardinality changes applied to an original query in order to deliver a refined one at a destination node of a modification tree.

At initialization, the modification tree is extended with a failed query and its operational graph, and a position of a query in the tree is returned to the cardinality checker, which triggers the creation of a new query candidate. For this purpose, the *query modifier* extracts a query from a specified position of the modification tree and rewrites it by consulting a graph database and adapting the modification tree. The rewriting process supports both topological and predicate changes. Structural modifications are represented by removing vertices, edges, and subgraphs. Predicate changes include increase and decrease of predicate intervals individually and simultaneously. The order of changing query vertices and edges is determined online based on the problem which has to be solved and already executed changes. A produced new query is redirected to the query manager which repeats the process if necessary. The modification terminates if a query, which delivers a required number of results is found or no new query candidates can be produced.

To summarize, the modification process is implemented as an iterative procedure, which intensively uses a modification tree and operational graphs. By using an operational graph and constructing a modification tree, the algorithm exhibits the following properties: (1) Any change applied to the operational graph propagates to its output node and all non-contributing modifications are prohibited in order to keep the syntactic distance minimal. (2) The structure of the operational graph is considered in order to take advantages of known dependencies. (3) The query modification process adapts to prohibited changes and current output cardinality.

In contrast to subgraph-based explanations and modification-based explanations for why-empty queries, we do not present a new model for generating user-aware answers. We believe that the user-integration methods can be used, which were previously introduced for other explanations. For example, the user-preference model for subgraph-based explanations can be re-applied to modification-based explanations in order to adapt the modification tree by re-arranging branches: Such neighboring operational nodes have to be modified first, which are less relevant to a user.

In this section, we also compared our approach with three baselines according to the number of improved queries, cardinality, result, and syntactic distances, and the number of considered iterations. Our approach TRAVERSESEARCHTREE out-performs all competitors in terms of the quality of generated explanations and the quantity of modified queries. It also shows very similar performance to the A*-search. In order to improve the performance of our approach, we consider how topological changes influence the performance of the generation process. We came to the conclusion, that if a cardinality threshold differs from an original cardinality in several order of magnitude the topological changes have to be considered. For small cardinality distances, predicate changes are good enough to derive better explanations.

6 Conclusion

Graph databases implementing the property-graph model allow to store information of different degree of structure and provide sophisticated queries for data analysis. They keep heterogeneous information without a rigid schema as a property graph, where entities are represented by vertices and edges describe relations between them. The stored data graph can be easily modified by introducing new data or removing existing data. Keeping the data in the form of a graph makes it also possible to conduct complex graph algorithms over it and to combine standard queries with complex analytics. The flexibility of the property-graph model and various query types come at additional costs and complicate the query-answering process. Without deep data knowledge and with little experience in constructing graph-aware queries, a user can create queries that deliver no, too few, or too many results. He can get frustrated by receiving unexpected results, because the reason of unexpectedness is difficult to understand and resolve. Considering these facts, in this thesis we focus on the usability issues for graph databases implementing property graphs and study fundamental functionality for debugging the graph queries delivering unexpected results. We investigate the cardinality problems using pattern-matching queries as one of the commonly used graph-query types. To summarize, this thesis describes debugging features for pattern-matching queries delivering unexpected query results in the form of why-queries, which explain query results and thus make graph databases more user-friendly.

We classify unexpected results according to the subject of unexpectedness such as content or size of the received result. Content-based unexpectedness can mean presence of unexpected results or absence of expected results. Two kinds of why-queries deal with these issues: why-so and why-not queries. If the size of the result set does not satisfy user expectations because it has no, too few, or too many answers then we speak about why-empty, why-so-few, and why-so-many queries. Considering the fact that cardinality issues like receiving no or too many results are very typical for graph queries with multiple constraints, in this thesis we address them and provide cardinality-based why-queries over property graphs. In general, this thesis has the following contributions:

Extraction of Common Features for Why-Queries First, in this thesis we reviewed the existing state-of-the-art approaches for debugging unexpected results in order to extract general features enabling basic debugging capabilities. The extracted features include efficient generation of explanations, user integration, generation of different kinds of explanations, discovery of the reason of unexpectedness, and query refinement, which are further revised for graph databases.

Generation of Subgraph-Based Explanations One of the extracted features focuses on the discovery of the reason of unexpectedness. In the state-of-the-art systems, this aspect is represented by query-based explanations, which show the cause of the unexpected results as a part of a query graph. Speaking in graph terms, a query-based explanation for a pattern-matching query represents a query subgraph, which violates the cardinality constraint. We provided two methods for generating such explanations: DiscoverMCS and BoundedMCS algorithms for empty-answer and too-few- and too-many-answers problems, respectively. We evaluated both approaches using two data sets and showed several optimizations to improve their performance by preventing duplicate processing. We also increased the quality of generated subgraph-based explanations with considering weakly-connected and unconnected query subgraphs.

Generation of Modification-Based Explanations Instead of providing subgraph-based explanations, the user can also be directly supplied with the rewritten query, which corresponds to a given cardinality constraint. This answer is called a modification-based explanation and represents the second typical kind of explanations produced by state-of-the-art why-queries. We investigated two methods for generating such explanations: one for why-empty and another one for why-so-few and why-so-many queries. For the empty-answer problem, we proposed a query rewriting approach that relaxes specific query constraints and processes rewritten queries based on how likely they can deliver some results. For why-so-few and why-so-many queries we introduced the TraverseSearchTree algorithm, which supports fine-granular predicate and topological changes. This algorithm adapts to the cardinality problem that has to be solved, guarantees propagation of changes, and optimizes the search by rejecting non-contributing changes.

Comprehensive Analysis of Why-Explanations In this thesis, we proposed methods to explain unexpected results for pattern-matching queries over property graphs. In order to judge the quality of generated explanations, we compared them on three different levels: the syntactic, cardinality, and result distances. The syntactic distance describes how different the explanation appears to the user. The cardinality distance shows the difference between the user-defined cardinality threshold and size of the result, provided by the explanation. The result distance explains how many answers remain in the result set, after an explanation has been generated. The three-level comparison considers all important aspects for judging the quality of explanations.

Development of Models for User Integration To steer the generation of explanations according to the user interest, we proposed two ways for considering user preferences in specific query elements: The first approach, for generating subgraph-based explanations, requires a user to mark relevant query elements. Then it calculates the most relevant traversal path along the query, and traverses the query along it. The user integration is easily done by choosing the most relevant vertices and edges to process. The second approach for generating modification-based explanations for why-empty queries constructs a user-preference model from already rated explanations. Based on this model, the rewriting system adapts the modification process and discovers relevant explanations first. Both approaches are general enough and can be re-used in generating modification-based explanations for why-so-few and why-so-many queries.

With this thesis, we have taken the first step towards creating a debugging tool for why-queries over property graphs and opened up new challenges to be solved in order to make graph databases more usable by providing comprehensive explanation functionality.

References

[1] Daniel Abadi, Rakesh Agrawal, Anastasia Ailamaki, Magdalena Balazinska, Philip A. Bernstein, Michael J. Carey, Surajit Chaudhuri, Jeffrey Dean, AnHai Doan, Michael J. Franklin, Johannes Gehrke, Laura M. Haas, Alon Y. Halevy, Joseph M. Hellerstein, Yannis E. Ioannidis, H. V. Jagadish, Donald Kossmann, Samuel Madden, Sharad Mehrotra, Tova Milo, Jeffrey F. Naughton, Raghu Ramakrishnan, Volker Markl, Christopher Olston, Beng Chin Ooi, Christopher Ré, Dan Suciu, Michael Stonebraker, Todd Walter, and Jennifer Widom. The beckman report on database research. *SIGMOD Rec.*, 43(3):61–70, December 2014.

[2] Adriane Chapman and H. V. Jagadish. Why not? In *Proceedings of the 2009 ACM SIGMOD International Conference on Management of Data*, SIGMOD '09, pages 523–534, New York, NY, USA, 2009. ACM.

[3] Marie-Pierre Dubuisson and Anil K. Jain. A modified Hausdorff distance for object matching. In *Pattern Recognition, 1994. Vol. 1 - Conference A: Computer Vision amp; Image Processing., Proceedings of the 12th IAPR International Conference on*, volume 1, pages 566–568 vol.1, Oct 1994.

[4] Md. Saiful Islam, Chengfei Liu, and Jianxin Li. Efficient answering of why-not questions in similar graph matching. *Knowledge and Data Engineering, IEEE Transactions on*, 27(10):2672–2686, Oct 2015.

[5] H. V. Jagadish, Adriane Chapman, Aaron Elkiss, Magesh Jayapandian, Yunyao Li, Arnab Nandi, and Cong Yu. Making database systems usable. In *Proceedings of the 2007 ACM SIGMOD International Conference on Management of Data*, SIGMOD '07, pages 13–24, New York, NY, USA, 2007. ACM.

[6] Harold W. Kuhn. The hungarian method for the assignment problem. *Naval Research Logistics Quarterly*, 2(1-2):83–97, 1955.

[7] Silvano Martello and Paolo Toth. *Knapsack problems: algorithms and computer implementations*. John Wiley & Sons, Inc., New York, NY, USA, 1990.

[8] Davide Mottin, Francesco Bonchi, and Francesco Gullo. Graph query reformulation with diversity. In *Proceedings of the 21th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, KDD '15, pages 825–834, New York, NY, USA, 2015. ACM.

[9] Davide Mottin, Alice Marascu, Senjuti Basu Roy, Gautam Das, Themis Palpanas, and Yannis Velegrakis. A probabilistic optimization framework for the empty-answer problem. *Proc. VLDB Endow.*, 6(14):1762–1773, September 2013.

[10] Robert Pienta, Acar Tamersoy, Hanghang Tong, and Duen Horng Chau. MAGE: Matching approximate patterns in richly-attributed graphs. In *Big Data (Big Data), 2014 IEEE International Conference on*, pages 585–590, Oct 2014.

[11] Alexandra Poulovassilis and Peter T. Wood. Combining approximation and relaxation in semantic web path queries. In Peter F. Patel-Schneider, Yue Pan, Pascal Hitzler, Peter Mika, Lei Zhang, Jeff Z. Pan, Ian Horrocks, and Birte Glimm, editors, *The Semantic Web – ISWC 2010*, volume 6496 of *Lecture Notes in Computer Science*, pages 631–646. Springer Berlin Heidelberg, 2010.

[12] Bahar Qarabaqi and Mirek Riedewald. User-driven refinement of imprecise queries. In *Data Engineering (ICDE), 2014 IEEE 30th International Conference on*, pages 916–927, March 2014.

[13] Marko A Rodriguez and Peter Neubauer. Constructions from dots and lines. *Bulletin of the American Society for Information Science and Technology*, 36(6):35–41, 2010.

[14] Chad Vicknair. Research issues in data provenance. In *Proceedings of the 48th Annual Southeast Regional Conference*, ACM SE '10, pages 20:1–20:4, New York, NY, USA, 2010. ACM.

[15] A. Woodruff and Michael Stonebraker. Supporting fine-grained data lineage in a database visualization environment. In *Data Engineering, 1997. Proceedings. 13th International Conference on*, pages 91–102, Apr 1997.