Elena Vasilyeva, Thomas Heinze, Maik Thiele, Wolfgang Lehner

DebEAQ - debugging empty-answer queries on large data graphs

Erstveröffentlichung in / First published in:

IEEE 32nd International Conference on Data Engineering (ICDE). Helsinki, 16.05.-20.05.2016. IEEE, S. 1402-1405. ISBN 978-1-5090-2020-1.

DOI: http://dx.doi.org/10.1109/ICDE.2016.7498355

Final edited form was published in "IEEE 32nd International Conference on Data Engineering (ICDE). Helsinki 2016", S. 1402-1405, ISBN 978-1-5090-2020-1
http://dx.doi.org/10.1109/ICDE.2016.7498355

DebEAQ – Debugging Empty-Answer Queries On Large Data Graphs

Elena Vasilyeva[§*1], Thomas Heinze[§2], Maik Thiele[*3], Wolfgang Lehner[*4]
[§]SAP SE, Walldorf, 69190 Germany
[*]Database Technology Group, TU Dresden, Dresden, 01062 Germany
[1]elena.vasilyeva@sap.com, [2]thomas.heinze@sap.com, [3]maik.thiele@tu-dresden.de, [4]wolfgang.lehner@tu-dresden.de

***Abstract*—The large volume of freely available graph data sets impedes the users in analyzing them. For this purpose, they usually pose plenty of pattern matching queries and study their answers. Without deep knowledge about the data graph, users can create 'failing' queries, which deliver empty answers. Analyzing the causes of these empty answers is a time-consuming and complicated task especially for graph queries. To help users in debugging these 'failing' queries, there are two common approaches: one is focusing on discovering missing subgraphs of a data graph, the other one tries to rewrite the queries such that they deliver some results. In this demonstration, we will combine both approaches and give the users an opportunity to discover why empty results were delivered by the requested queries. Therefore, we propose DebEAQ, a debugging tool for pattern matching queries, which allows to compare both approaches and also provides functionality to debug queries manually.**

I. Introduction

Following the principle 'data comes first, schema comes second', graph databases allow to store data without having a predefined, rigid schema and enable a gradual evolution of data together with its schema. Moreover, graph databases enrich analytical queries known from relational databases with graph-specific features such as a shortest path, pattern matching etc. The schema flexibility and complexity of graph queries make it extremely difficult for users to formulate appropriate queries. This often leads to a situation that a query result is empty, which is denoted as an empty-answer problem [1], [2], [3], [4]. For the relational databases, solutions to this problem can be classified into three groups: (1) query-based solutions, which explain the failure in terms of a query graph, (2) data-driven explanations exploring trusted and untrusted data sources in data integration systems and describing which tuples have to be changed in untrusted data sources to deliver some results, and (3) query rewriting approaches, which modify 'failing' queries in such a way that they provide some results.

While this problem is well-investigated in the relational database research [1], [2] and RDF graphs [5], [6], it has received only limited attention in the graph database research on property graphs [7], [8]. Supposing that the data graph is acquired from a trusted data source, two approaches remain: query-based solutions and query rewriting techniques. Applying them on the graph data model, the first approach should describe the problem in terms of a graph query consisting of vertices and edges. Specifically, a graph database can determine the reason of a 'failing' query by detecting which parts of a query are present and which are missing from a data graph [7]. We call this approach a subgraph-based solution. Query rewriting techniques should consider the topology and notation of the query during rewriting and should propose query relaxation techniques to rewrite the query in such a way that it delivers a non-empty result [8]. Both solutions are complementary and can be used together.

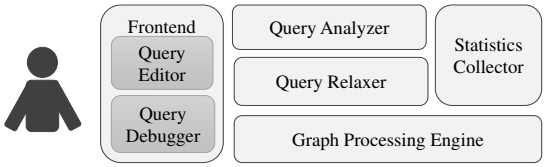


Fig. 1. The system architecture

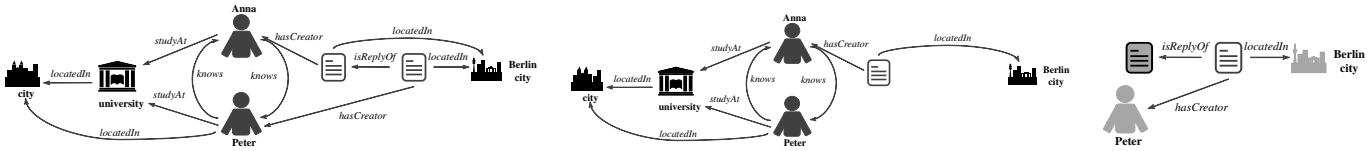
In this demonstration, we propose DebEAQ, a debugging tool, which assists users in analyzing unexpected query answers for pattern matching queries over property graphs by using a subgraph-based solution and query rewriting techniques. In the first case, users will be able to discover a potential reason for a failure of a graph query in terms of its topology. This approach executes an input query in a debugging mode, studies the topology of an input query, and explains a failure as a missing subgraph of an input query. In the second case, the tool utilizes the topology and notation of a query and changes an input query to deliver a non-empty answer. Both approaches can be applied sequentially. Additionally, users will be able to modify a query manually. At the end of the demonstration, users will also have the possibility to compare both approaches and the manual process by investigating statistics that are collected during a debugging session.

II. System Architecture

The main goal of the demonstration is to provide a debugging support for pattern matching queries delivering empty results. We focus on a property graph model [9] since it is a graph data model mostly used in graph databases describing the data in a very natural way. It considers entities as vertices and relationships as edges between them. The same vertices can have multiple connections between each other. Our tool supports pattern matching queries which are a commonly used query type in graph databases.

The architecture of our tool as well as its individual components are presented in Figure 1. Users are communicating with the system via a JavaScript-based *Query Editor* and *Query Debugger*. The query editor proposes a set of queries to users, which are expressed in visual and textual formats. Alternatively, users can create their own query in the editor or change the proposed one. The query debugger assists users

Final edited form was published in "IEEE 32nd International Conference on Data Engineering (ICDE). Helsinki 2016", S. 1402-1405, ISBN 978-1-5090-2020-1
http://dx.doi.org/10.1109/ICDE.2016.7498355



(a) A query example (b) A differential answer (c) A difference subgraph

Fig. 2. An example of a differential query and its result. The query searches for two friends Peter and Anna with a pattern: they study at the same university, which is located in a city, where Peter lives, and Peter replied to Anna's message in Berlin.

during a debugging session by proposing next steps to be done and visualizing intermediate debugging results.

The most important job—debugging—is executed by the engine itself which combines the *Query Analyzer*, the *Query Relaxer*, and the *Statistics Collector*. The query analyzer provides a subgraph-based solution and investigates an empty-answer problem on a query level by traversing an input graph query and showing to users, which parts of a graph query are responsible for the delivery of an empty answer. For taking the decisions, it consults the statistics collector, which gathers all necessary information about an input query and data itself like a size of a data graph, cardinalities of individual edges, vertices, and paths. After discovering the reasons of an empty answer, the query relaxer is activated and it modifies an input query to deliver a non-empty result. For this purpose, the query relaxer investigates an empty-answer problem with the help of query rewriting techniques. It supplies users with query candidates delivering non-empty answers ordered by their similarity scores with respect to an input query.

The tool itself communicates with the *Graph Processing Engine* storing a data graph, executing the queries posed by the tool, and providing their results. It accepts user requests in a form of a pattern to be searched in the data graph. The graph processing engine used in the debugging tool is based on a research prototype GRAPHITE [10] implemented on top of an in-memory column store. The data graph is allocated in two flexible tables separately for vertices and edges allowing to store multiple attributes and making the schema evolve by introducing new attributes on the fly.

III. DISCOVERING REASONS OF A FAILURE

The query analyzer investigates the problem of empty answers on a query level. To support users in getting an understanding of the reasons of an empty answer, the notion of a differential query [7] is used. For this purpose, the system executes an input query as a differential query that seeks for the points of a failure in a graph query in terms of missing edges and vertices. As a result, a differential query delivers two subgraphs: (1) a discovered data subgraph which is isomorphic to the query subgraph and (2) a difference graph describing the remaining part of the query that is missing from the data graph.

To discover the missing part of an input query, the query analyzer detects the maximum common subgraph between a data graph and an input query. For this purpose, the GraphMCS algorithm [7] is used, which selects a first query vertex and edge to traverse and discovers for this starting point a set of maximum common subgraphs. To cover the whole search space of candidates, the algorithm has to traverse all possible combinations of subsequent traversals, which is a task of exponential complexity. The query analyzer avoids this complexity by applying different heuristics and selects only a subset of possible traversals based on the following principles: First, vertices and edges with a zero cardinality are filtered out. Second, a traversal begins from a vertex and edge with the lowest non-zero cardinality. The rationale behind this is to keep a number of discovered maximum common subgraphs minimal.

An absence of a query part from a data graph can split a graph query in several disconnected components. In this case, the maximum common subgraph can be potentially missed, if the search is executed in a smaller query component. For such situations, the GraphMCS algorithm provides a restart strategy that covers all unconnected components and chooses the largest discovered subgraph. After the maximum common subgraph is detected, a difference graph is calculated from a graph query and a maximum common connected data subgraph. A difference graph consists of those query vertices and edges whose instances were not discovered in a data graph and it is also annotated with additional constraints at the vertices, which are adjacent to the discovered subgraph as connecting points.

Assume the graph query in Figure 2(a) delivering an empty result. The query analyzer discovers a maximum subgraph existing in a data graph, which corresponds to the query subgraph in Figure 2(b). There can be multiple instances of this subgraph in a data graph. In addition, the query analyzer determines reasons of a failure, i.e. missing subgraphs with initialized adjacent vertices (Figure 2(c)). The missing subgraph in the example consists of instances for two vertices *Peter* and *Message* annotated by their identifiers, which are adjacent to a discovered data subgraph, and a missing structure from two vertices and three edges. For the demonstration, the second part of the answer is more important, since it represents debugging information that can be used for a manual or automatic relaxation of a query. Our demonstration provides an input query, in which missing parts are drawn by dashed lines, as a subgraph-based solution. To allow this annotation, a query model supports a system property *missing*, which shows whether a vertex or edge are present in a data graph.

IV. QUERY RELAXATION

If users need automatic support in query rewriting, they can trigger a query modification executed by the query relaxer. The query relaxer represents a 'Why Empty?' Engine [8] shown in Figure 4 and aims at relaxing a 'failing' query in such a way that a generated query candidate has a non-empty answer. The query relaxer consists of the following components: The relaxation process is maintained by the *Relaxation Manager*,

Final edited form was published in "IEEE 32nd International Conference on Data Engineering (ICDE). Helsinki 2016", S. 1402-1405, ISBN 978-1-5090-2020-1
http://dx.doi.org/10.1109/ICDE.2016.7498355

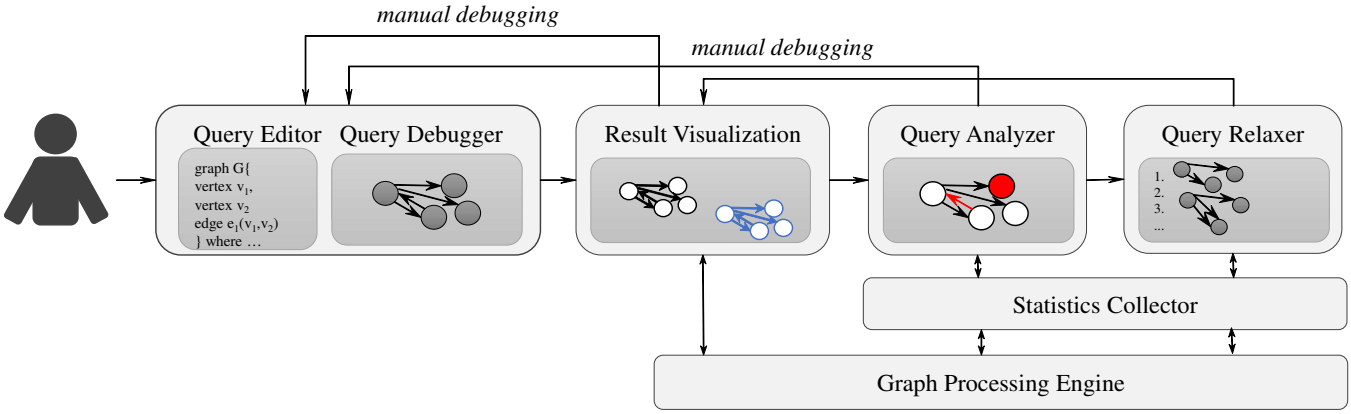


Fig. 3. A debugging process

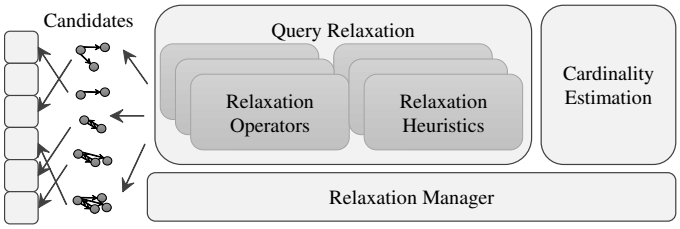


Fig. 4. The architecture of a query relaxer

which redirects queries to the graph processing engine, stores statistics, and takes placement decisions for generated query candidates. The *Query Relaxation* component is responsible for the generation of new query candidates. It consists of relaxation operators and heuristics. The DebEAQ tool supports the following operations to relax the query: predicate and vertex deletion for vertices; predicate, type, direction, and edge deletion for edges. Which edges or vertices to relax is decided by relaxation heuristics. Although the tool supports several heuristics, we use only a maximum impact heuristic, which has been shown to be the most effective one [8]. Those elements are chosen to be relaxed, whose relaxation has the highest cardinality impact on neighboring vertices. By modifying a query, the query relaxer studies cardinalities of its predicates, edge types, vertices, and edges. In taking all these decisions into account, the relaxation manager is supported by the *Cardinality Estimation*.

The relaxation process itself is modeled as an A*-search [8]: the candidates are located in an ordered buffer and the most promising candidates are processed first (see on the left side in Figure 4). To arrange the candidates in a pool, the system calculates an expected cardinality benefit as an average path cardinality of a step with $size = 1$. The candidates with higher promising benefits are preferred. However, this can lead to strong relaxations and, therefore, a final query candidate can be very different from an input query. To postpone the evaluation of strong relaxations to a later point in time, the system compares query candidates also according to their cardinality-based graph edit distances, which allows to choose less relaxed queries first. A cardinality-based graph edit distance expresses how different a query candidate is from an input query in terms of a relative cardinality change caused by a relaxation normalized to a complete relaxation. Candidates with a higher distance have stronger changes and have to be evaluated later.

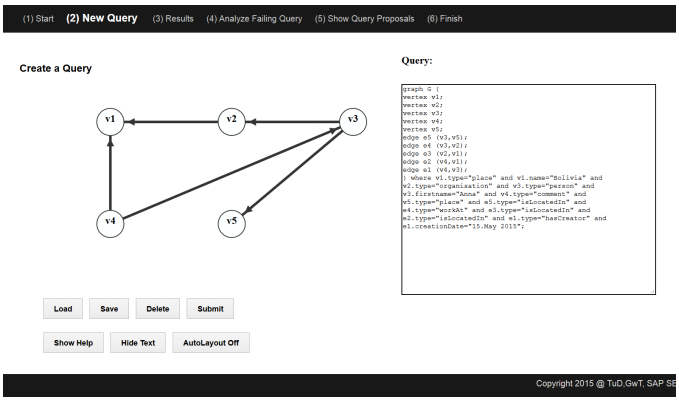
V. Demonstration Scenarios

This demonstration showcases the debugging of pattern matching queries over a property graph delivering empty results by applying a subgraph-based solution and query rewriting techniques. The goal of the demonstration is to bring together both complimentary approaches. While the discovery of reasons for a failure focuses mostly on a topology, query relaxation techniques give an inside whether it is possible to change also the notation of a query to deliver some results. Our demonstration consists of a tool with an interactive frontend and a graph processing engine GRAPHITE [10] that runs on a single server machine with SUSE Linux Enterprise Server 11 (64 bit), an Intel Xeon Processor E5-2643 with 24CPUs and 96 GB RAM. As a data graph we use LDBC SF1, which represents a social network with 3.7M vertices and 21.7M edges, and a property graph generated from the DBPEDIA with 819K edges and 182K vertices. As queries we have chosen several traversal-based queries from the LDBC interactive workload and specified them as pattern matching queries in GRAPHQL language and several randomly generated queries for DBPEDIA graph.

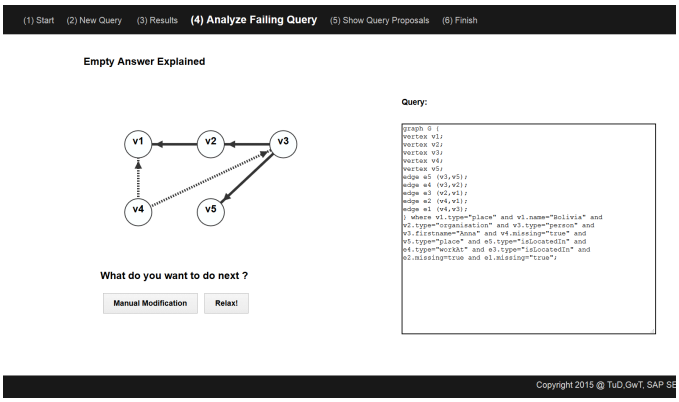
The demonstration flow is presented in Figure 3. At the beginning, users have several options how to start a debugging process: they can either choose a query from a list of available 'failing' queries or they can create a query on their own in the query editor in a textual or in a graph-visualized form (see Figure 5(a)). Both editors maintain the same common graph query model; therefore, all changes done in one panel are immediately reflected in the second panel. The text editor uses the GRAPHQL language [11]. The user interface provides also the following functionality: Users can upload their own queries and hide the textual representation. The visual editors use tooltips to visualize the properties of edges and vertices.

During a demonstration, users are supported by the query debugger, which guides them through a debugging process. After a query has been defined, the graph processing engine executes it. If a query fails to deliver any answer, an error message appears in the visualization panel. In this case, users

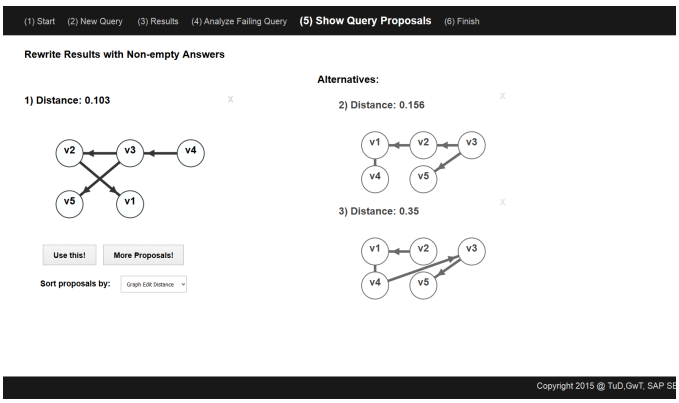
Final edited form was published in "IEEE 32nd International Conference on Data Engineering (ICDE). Helsinki 2016", S. 1402-1405, ISBN 978-1-5090-2020-1
http://dx.doi.org/10.1109/ICDE.2016.7498355



(a) Visual and textual query editors



(b) Visualization of reasons of a failure in a graph query



(c) Query candidates delivering non-empty results

Fig. 5. Demonstration screens

can trigger the query analyzer to discover reasons of a failure, which consults statistics collector and graph processing engine to acquire processed parts of an input query and to calculate difference graphs. As an answer from the query analyzer, users receive an input query with missing parts marked by a dashed line. A query is represented in the visual and textual form (see Figure 5(b)). At this step, users again have a possibility to rewrite a query manually or can trigger its relaxation. To relax a query, the graph relaxer acquires all the necessary statistics from the statistics collector like cardinalities of query vertices, edges, and paths. As a result of the relaxation, users receive a list of query candidates delivering non-empty result sets, which are ordered by their distance to an input query (see Figure 5(c)). To allow this compact representation, users can request more proposals and remove non-interesting candidates. Users can choose a favorite one for investigating its results. After the relaxation is done, users can study the collected runtime statistics about the relaxation process.

To conclude, during the demonstration users will have the possibility to combine both approaches, to create and debug queries manually, and to compare all these techniques with each other. The DebEAQ website features a screencast as well as the description of this demonstration[1].

VI. Conclusion

This paper introduces DebEAQ, a tool for debugging graph pattern queries over property graphs delivering empty results. It combines two approaches for explaining causes of empty answers: a subgraph-based explanation and rewriting techniques. The subgraph-based solution focuses on the topology of a query, discovers a maximum common data subgraph, and calculates a reason of an empty answer—a difference query subgraph. In contrast, query rewriting techniques consider the topology and notation of a graph query and relax a query by discarding predicates, vertices, and edges possibly causing a failure. This demonstration highlights both approaches and illustrates how they can benefit from each other. Finally, it allows to compare them and showcases the usefulness of automatic debugging approaches compared to a manual debugging.

References

[1] D. Mottin, A. Marascu, S. Basu Roy, G. Das, T. Palpanas, and Y. Velegrakis, "IQR: An interactive query relaxation system for the empty-answer problem," in *Proc. SIGMOD*. ACM, 2014, pp. 1095–1098.

[2] D. Mottin, A. Marascu, S. B. Roy, G. Das, T. Palpanas, and Y. Velegrakis, "A probabilistic optimization framework for the empty-answer problem," *Proc. VLDB Endow.*, pp. 1762–1773, 2013.

[3] D. Mottin, F. Bonchi, and F. Gullo, "Graph query reformulation with diversity," in *Proc. SIGKDD*, ser. KDD '15. New York, NY, USA: ACM, 2015, pp. 825–834.

[4] U. Junker, "QUICKXPLAIN: preferred explanations and relaxations for over-constrained problems," in *AAAI*, 2004, pp. 167–172.

[5] A. Poulovassilis and P. T. Wood, "Combining approximation and relaxation in semantic web path queries," in *The Semantic Web–ISWC 2010*. Springer, 2010, pp. 631–646.

[6] H. Huang, C. Liu, and X. Zhou, "Approximating query answering on rdf databases," *World Wide Web*, vol. 15, no. 1, pp. 89–114, 2012.

[7] E. Vasilyeva, M. Thiele, C. Bornhövd, and W. Lehner, "GraphMCS: Discover the unknown in large data graphs," in *Workshops EDBT*, 2014, pp. 200–207.

[8] E. Vasilyeva, M. Thiele, A. Mocan, and W. Lehner, "Relaxation of subgraph queries delivering empty results," in *Proc. SSDBM*, 2015, pp. 28:1–28:12.

[9] M. A. Rodriguez and P. Neubauer, "Constructions from dots and lines," *Bulletin of the American Society for Inf. Science and Technology*, pp. 35–41, 2010.

[10] M. Paradies, W. Lehner, and C. Bornhövd, "GRAPHITE: an extensible graph traversal framework for relational database management systems," in *Proc. SSDBM*, 2015, pp. 29:1–29:12.

[11] H. He and A. K. Singh, "Graphs-at-a-time: Query language and access methods for graph databases," in *Proc. SIGMOD*, ser. SIGMOD '08. ACM, 2008, pp. 405–418.

[1] https://wwwdb.inf.tu-dresden.de/misc/debeaq/