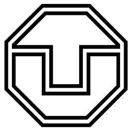


TECHNISCHE UNIVERSITÄT DRESDEN

Faculty of Computer Science
Database Technology Group

Why-Query Support in Graph Databases

DISSERTATION
ZUR ERLANGUNG DES AKADEMISCHEN GRADES DOKTORINGENIEUR (DR.-ING.)

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
Fakultät 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

Tag der Verteidigung: 08. November 2016

Dresden, im August 2016

Abstract

In the last few decades, database management systems became powerful tools for storing large amount of data and executing complex queries over them. In addition to extended functionality, novel types of databases appear like triple stores, distributed databases, etc. Graph databases implementing the property-graph model belong to this development branch and provide a new way for storing and processing data in the form of a graph with nodes representing some entities and edges describing connections between them. This consideration makes them suitable for keeping data without a rigid schema for use cases like social-network processing or data integration. In addition to a flexible storage, graph databases provide new querying possibilities in the form of path queries, detection of connected components, pattern matching, etc.

However, the schema flexibility and graph queries come with additional costs. With limited knowledge about data and little experience in constructing the complex queries, users can create such ones, which deliver unexpected results. Forced to debug queries manually and overwhelmed by the amount of query constraints, users can get frustrated by using graph databases. What is really needed, is to improve usability of graph databases by providing debugging and explaining functionality for such situations. We have to assist users in the discovery of what were the reasons of unexpected results and what can be done in order to fix them.

The unexpectedness of result sets can be expressed in terms of their size or content. In the first case, users have to solve the empty-answer, too-many-, or too-few-answers problems. In the second case, users care about the result content and miss some expected answers or wonder about presence of some unexpected ones. Considering the typical problems of receiving no or too many results by querying graph databases, in this thesis we focus on investigating the problems of the first group, whose solutions are usually represented by why-empty, why-so-few, and why-so-many queries.

Our objective is to extend graph databases with debugging functionality in the form of why-queries for unexpected query results on the example of pattern matching queries, which are one of general graph-query types. We present a comprehensive analysis of existing debugging tools in the state-of-the-art research and identify their common properties. From them, we formulate the following features of why-queries, which we discuss in this thesis, namely: holistic support of different cardinality-based problems, explanation of unexpected results and query reformulation, comprehensive analysis of explanations, and non-intrusive user integration. To support different cardinality-based problems, we develop methods for explaining no, too few, and too many results. To cover different kinds of explanations, we present two types: subgraph- and modification-based explanations. The first type identifies the reasons of unexpectedness in terms of query subgraphs and delivers differential graphs as answers. The second one reformulates queries in such a way that they produce better results. Considering graph queries to be complex structures with multiple constraints,

we investigate different ways of generating explanations starting from the most general one that considers only a query topology through coarse-grained rewriting up to fine-grained modification that allows fine changes of predicates and topology. To provide a comprehensive analysis of explanations, we propose to compare them on three levels including a syntactic description, a content, and a size of a result set. In order to deliver user-aware explanations, we discuss two models for non-intrusive user integration in the generation process.

With the techniques proposed in this thesis, we are able to provide fundamentals for debugging of pattern-matching queries, which deliver no, too few, or too many results, in graph databases implementing the property-graph model.

Acknowledgments

First of all, I would like to thank my advisor Professor Wolfgang Lehner for giving me the opportunity to write my PhD thesis at Technische Universität Dresden in the database technology group, for guiding and supporting me over the years. He also gave me a chance to get to know the HANA Student Campus with its nice working atmosphere and interesting people.

I would like to express my gratitude to Professor Katja Hose for taking over the role as co-referee of this thesis and to all members of the examination committee.

I am also grateful to Gregor Hackenbroich for giving me an opportunity to start my research at SAP Dresden. I owe my special thanks to Arne Schwarz, who gave me a chance to finish my PhD work at SAP Walldorf in the HANA Student Campus.

Furthermore, I would like to thank Maik Thiele for co-advising my research, for reading all my papers, and for supporting me during this time. I am also grateful to Adrian Mocan, who helped me to establish a balance between research and project work. During my research, I had a luck to work in three different teams including SAP Research Dresden, the database technology group, and the SAP HANA Student Campus. My first PhD years I spent at SAP Dresden, where I had a chance to be involved in several projects and get interesting working experience. I had a nice time to work at the database chair twice a week, where I could exchange my ideas with other PhD students, participate in research seminars, and get some feeling of the university life. I finalized my PhD work at SAP HANA Campus in Walldorf, in the very enthusiastic team of PhD students, where I had a freedom of doing my research and luck to be involved in the campus life. I am especially thankful to Marcus Paradies, who gave me a chance to implement and test all the concepts of this thesis in the GRAPHITE prototype. I had the opportunity to work in all these three groups and gain a lot of experience from my colleagues. Thank you!

My special thanks go to Thomas, who supported me during the whole time of my PhD thesis and motivated me in doing this work. Thank you for all our discussions, for all your arguments, for your support, and for your belief in me.

I also would like to thank Maik, Thomas, Ulrike, Alina, Angela, and Katrin for comments on early versions of this thesis. All this work would not be possible without support of my family and friends. Thank you!

Особую благодарность я выражаю своей семье и друзьям, которые поддерживали меня все эти годы, верили в успех, относились с пониманием к тому, что я всё время была занята и работала над диссертацией. Без вашей поддержки я бы не справилась. Спасибо, мама и папа!

Contents

1 Introduction **1**
1.1 Contributions . . . 3
1.2 Outline . . . 5

2 Foundations of Why-Queries **7**
2.1 Why-So Queries . . . 7
2.1.1 Eager Provenance . . . 8
2.1.2 Lazy Provenance . . . 9
2.1.3 Common Properties . . . 12
2.2 Why-Not Queries . . . 13
2.2.1 Reasons of Missing Answers . . . 13
2.2.2 Provenance-Based Explanations . . . 15
2.2.3 Modification-Based Explanations . . . 20
2.2.4 Query-Type Oriented and Model-Specific Why-Not Queries . . . 21
2.2.5 Common Properties . . . 24
2.3 Why-Empty and Why-So-Few Queries . . . 25
2.3.1 Modification-Based Explanations . . . 25
2.3.2 Prevention Methods . . . 29
2.3.3 Query-Type Oriented And Model-Specific Why-Empty Queries . . . 29
2.3.4 Common Properties . . . 30
2.4 Why-So-Many-Queries . . . 31
2.4.1 Result-Based Explanations . . . 31
2.4.2 Modification-Based Explanations . . . 36
2.4.3 Common Properties . . . 38
2.5 Summary . . . 39

3 Why-Queries in Graph Databases **43**
3.1 Properties of Why-Queries in Graph Databases . . . 43
3.1.1 General Graph Model . . . 43
3.1.2 Supported Types of Graph Queries . . . 44
3.1.3 Holistic Support of Different Cardinality-Based Problems . . . 44
3.1.4 Explanation of Unexpected Results and Query Reformulation . . . 45
3.1.5 Comprehensive Comparison of Explanations . . . 45
3.1.6 Non-Intrusive User Integration . . . 46
3.2 Comprehensive Comparison of Explanations . . . 46
3.2.1 Preliminaries . . . 47
3.2.2 Syntactic Level . . . 49
3.2.3 Cardinality Level . . . 54
3.2.4 Result Level . . . 55

3.2.5 Evaluation 57
3.3 Summary 63

4 Explaining Unexpected Results 65
4.1 Preliminaries 65
4.1.1 Detection of Maximum Common Connected Subgraphs 67
4.1.2 Calculation of Differential Graphs 68
4.2 Generation of Subgraph-Based Explanations 68
4.2.1 The DISCOVERMCS Algorithm for Why-Empty Queries 69
4.2.2 The BOUNDEDMCS Algorithm for Why-So-Few and Why-So-Many Queries 70
4.2.3 Calculation of Differential Subgraphs 72
4.3 Optimization 73
4.3.1 Processing of Weakly Connected Components 73
4.3.2 Selection of Single Traversal Path 75
4.3.3 Processing of Unconnected Components 76
4.4 User Integration 77
4.4.1 Definition of User Preferences 77
4.4.2 User-Centric Selection of Traversal Path 78
4.4.3 Rank Calculation 82
4.5 Evaluation 82
4.5.1 DISCOVERMCS Algorithm for Empty-Answer Problems 83
4.5.2 The BOUNDEDMCS Algorithm for the Too-Many-Answers Problem 88
4.5.3 Evaluation Summary 92
4.6 Summary 92

5 Coarse-Grained Why-Empty Query Modification 95
5.1 Predicate- and Topology-Aware Modification Process 95
5.1.1 System Architecture 95
5.1.2 Query Relaxation 96
5.2 Cardinality Estimation 98
5.2.1 Query-Dependent Statistics 98
5.2.2 Querying Statistics for Edges and Vertices 101
5.2.3 Querying Statistics for Paths(n) 102
5.3 Query-Candidate Selection 104
5.3.1 Placement of New Query Candidates 104
5.3.2 Calculation of Induced Cardinality Changes 105
5.4 User Integration 108
5.4.1 User-Preference Model 109
5.4.2 Adaptation of Query Rewriting 110
5.5 Evaluation 111
5.5.1 Priority Functions of Query-Candidate Selector 112
5.5.2 Runtime Convergence 116
5.5.3 Priority Function with Average Path(1) Cardinality and Induced Cardinality Changes 118
5.5.4 User Integration 121
5.5.5 Evaluation Summary 122
5.6 Summary 123

6 Fine-Grained Cardinality-Driven Query Modification 125
6.1 Predicate- and Topology-Aware Modification Process 125
6.1.1 General Modification Process 126
6.1.2 Operational Graph-Query Representation 127
6.1.3 Modification Tree 129
6.2 Generation of Modification-Based Explanations 131
6.2.1 TRAVERSESEARCHTREE Algorithm 133
6.2.2 Generation of New Query Candidates 134
6.3 Adaptation of Modification Tree 137
6.3.1 Guaranteeing of Change Propagation 138
6.3.2 Discarding of Non-Contributing Changes and Search Branches . . 138
6.4 Evaluation 140
6.4.1 Baseline Approaches 140
6.4.2 Baseline Comparison 142
6.4.3 Topology Consideration 147
6.4.4 Evaluation Summary 150
6.5 Summary 150

7 Conclusion and Future Work 153

A Evaluation Setup 157
A.1 System Overview 157
A.2 Data Sets 157
A.2.1 LDBC Data Set and Its Queries 157
A.2.2 DBPEDIA Data Set and Its Queries 159

B Additional Evaluation Results 163
B.1 User Integration in Why-Empty Query Rewriting 163
B.2 Resource Consumption for Why-Empty Query Rewriting 164

Bibliography 167

List of Figures 181

List of Tables 185

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 their querying becomes an issue reducing their usability. This issue has been extensively studied by Jagadish et al. [77], 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 [32], presence of unexpected results [144], empty [108], too few [119], or too many answers [106]. The corresponding why-queries are called why-not, why-so, why-empty, why-so-few, and why-so-many queries.

This debugging functionality is useful in the following scenarios including but not limited to:

- **Debugging unexpected results** is required in daily work of database users who design queries for acquiring information, analyzing stored data, etc.

- **Data integration** systems especially suffer from empty-answer and too-many-answers problems, which are typical for use cases, where only limited knowledge about the underlying data is available, data is extracted from multiple, potentially unreliable data sources, or data-transformation steps are executed during querying.
- **Data generation testing** executes data-consistency checks and can face the problem that queries, which have to deliver the same number of results, provide different answers. Therefore, such systems have to support why-empty, why-so-few, and why-so-many queries.
- **Selection of items with the same characteristics**, whose information is stored in the database, is typically required for a health study or a user survey.

In general, 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.

Content-Based Why-Queries are extensively studied in relational database management systems (RDBMS). Why-so queries typically target data integration and transformation systems, which map, collect, and process data from different sources. These are data-provenance systems, which track how data is transformed in order to answer the query. The research of these systems focuses on studying which information has to be tracked and how provenance has to be calculated: eagerly [23, 43] or lazily [2, 89]. One of the first solutions for why-not queries was introduced by Chapman et al. [32], who detects the reason of unexpectedness in terms of those manipulations which remove items of interest from the result set. There is also multiple work for rewriting queries in order to retrieve missing answers, e.g., in ConQueR [132]. Why-not queries are provided not only for SPJUAG queries, but also for specific requests like for example skyline and reverse skyline queries [39, 76].

Cardinality-Based Why-Queries including why-empty, why-so-few, and why-so-many queries investigate an unexpected size of a result set. This unexpectedness is typical in the systems, whose querying is complicated with the limited knowledge of underlying data or complex queries. In such scenarios, users may construct queries that are too general or too specific and receive too many or no results, respectively. These why-queries are represented by rewriting methods for the empty-answer problem [108], cardinality-assurance approaches [119], and ranking-based methods [69] for why-so-few and why-so-many queries. Solutions for cardinality-based problems typically rewrite queries or improve the representation of results. Only in a few cases, they discover the reason of unexpectedness, which makes cardinality-based why-queries different from the content-based ones. In addition, these methods also concern a cardinality constraint, which impedes the problem investigation. Similar to content-based why-queries, cardinality-based solutions are extensively studied for RDBMS.

The aforementioned 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 studied in the object-relational setup [150], why-not pattern matching is proposed for multiple labeled graphs [72], why-empty queries can be defined for

data in the RDF format (Resource Description Framework) [117], and why-so-many queries consider richly attributed graphs [116].

Referring to the usability study by Jagadish et al. [77], 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 [1], SAP HANA [2], and ORACLE BIG DATA SPATIAL AND GRAPH [3].

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, definition of correct queries becomes even more complicated for graph queries consisting of multiple query constraints. Therefore, the usability issue requiring explanations as described by Jagadisch et al. [77] for query results is of a 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 the 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 by predicates for attribute values and 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 users and should be avoided. Therefore, generation of explanations has to be able to consider user interest in specific query elements 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 of pattern-matching queries in graph databases implementing the property-graph model and how to support user-integration strategies in the explanation generation.

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

[1] https://neo4j.com/
[2] https://www.sap.com/product/technology-platform/hana.html
[3] https://www.oracle.com/database/spatial/index.html

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 users with the most appropriate explanation, we propose their 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 users. 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 [32], 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 user 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.

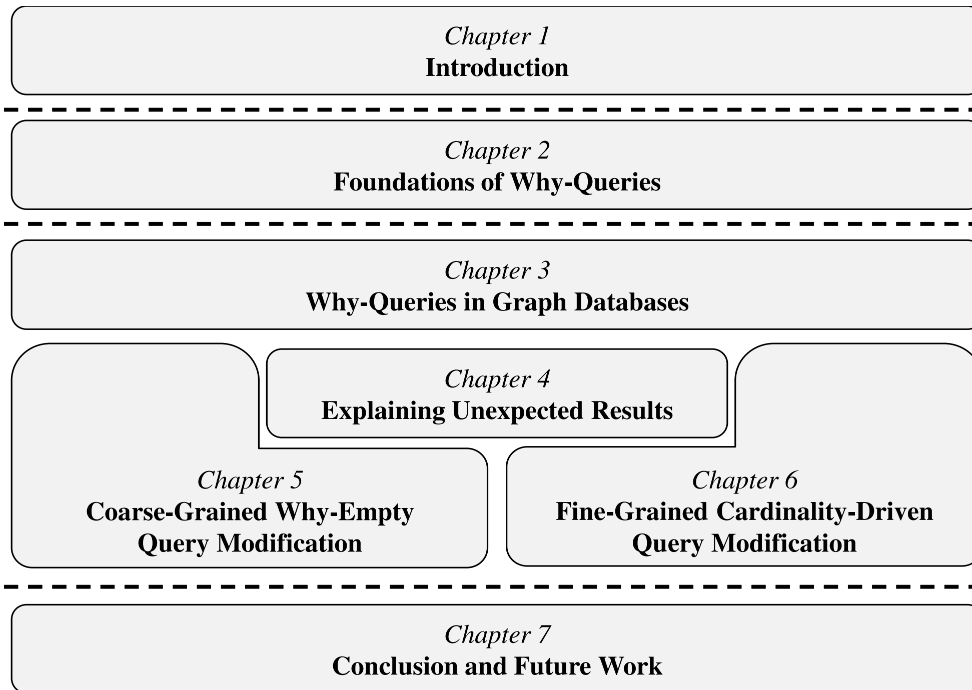


Figure 1.1: Thesis outline

1.2 Outline

Figure 1.1 illustrates the outline for this thesis, with the remaining chapters organized as follows. In Chapter 2, we analyze the state-of-the-art approaches for debugging why-queries and extract a set of general features from them. For each type of why-queries, we provide an overview of used data models and generated types of explanations. The chapter concludes with an overview of properties extracted from the related work, which are common for why-queries and should be implemented in order to provide explanations for pattern-matching queries in graph databases.

In Chapter 3, these properties are discussed in the specific context of the property-graph model, which include, for example, generation of subgraph-based and modification-based explanations and their comprehensive analysis. This chapter also describes in detail and evaluates metrics for comparing explanations, which are used along all subsequent chapters to judge the quality of produced explanations.

The description of core debugging functionality for why-queries starts in Chapter 4 with the generation of subgraph-based explanations that describe the reasons of failures in terms of query subgraphs. These explanations are derived by traversing the graph query and providing the differential subgraphs. These subgraphs can be used as an upper threshold for query rewriting investigated in the following chapters. To generate subgraph-based explanations, we propose two algorithms DiscoverMCS and BoundedMCS along with several optimization techniques and evaluate them on two data sets. In addition, we describe a way to incorporate user interest in the traversal, which is based on the detection of the most relevant traversal path along a query.

In Chapter 5, we continue describing core debugging functionality with generation of coarse-grained modification-based explanations for why-empty queries that are derived by discarding query constraints until a rewritten query delivers at least some results. Similar to subgraph-based explanations, we propose a method for non-intrusive user integration in the generation process, which automatically extracts a

user-preference model from user ratings of already discovered solutions. This modification-based approach is also evaluated on two data sets.

The fine-grained modification-based explanations are discussed afterwards in Chapter 6, where the rewriting process adjusts both topology and predicates (on the value level) such that a generated explanation delivers results with a smaller cardinality distance than the failed query. Finally, in Chapter 7 we summarize thesis contributions and findings and discuss open research challenges.

2

Foundations of Why-Queries

In this chapter, existing types of explanations for unexpected query results in the state-of-the-art systems are presented and their applicability to graph databases implementing the property-graph model is discussed. Unexpectedness of a query result can be expressed in terms of its content or size and can be explained by content-based or cardinality-based why-queries presented in Figure 2.1.

Content-based why-queries study why some specific answers are existing or missing from a result set and are represented by why-so and why-not queries, correspondingly. Cardinality-based why-queries focus on the size of a result set and explain why a result is empty or consists of too many or too few answers. The corresponding why-query types are implemented by why-empty, why-so-many, and why-so-few queries, respectively.

In this chapter, we will first give an overview on existing solutions for content-based why-queries in Sections 2.1 – 2.2. Although this class of why-queries has no direct connection to cardinality problems, which are tackled in this thesis, the analysis of its related work helps to derive features, which are necessary to implement in a debugging tool for an unexpected number of query results. Afterwards, we discuss solutions for cardinality-based why-queries in Sections 2.3 – 2.4. Finally, we introduce debugging features, which are derived from the related work.

2.1 Why-So Queries

Receiving query results, a user may wonder why a specific item exists in the result set, where it comes from, and who else uses this data [144]. These questions aim at studying the origin of a result set and affect its correctness and trust.

Consider a general data flow, in which a system extracts some data, applies a set of operations to it, and maybe visualizes the final result. The origin of results can be described by its processing history called lineage or pedigree [150]. It allows to trace the impact of each processing step or faulty data source on the final result. Explaining the result provenance is especially critical for example by querying web data, where databases can keep transformed data, which is possibly retrieved from data sources with different levels of trust. In addition to trust, a temporal aspect for evolving source databases should be considered. In this case, a provenance report has to include a correct version of the studied data and its location in other versions if required [144].

Generally speaking, all methods for tracing data provenance can be classified as *eager* or *lazy* methods, which will be described as follows.

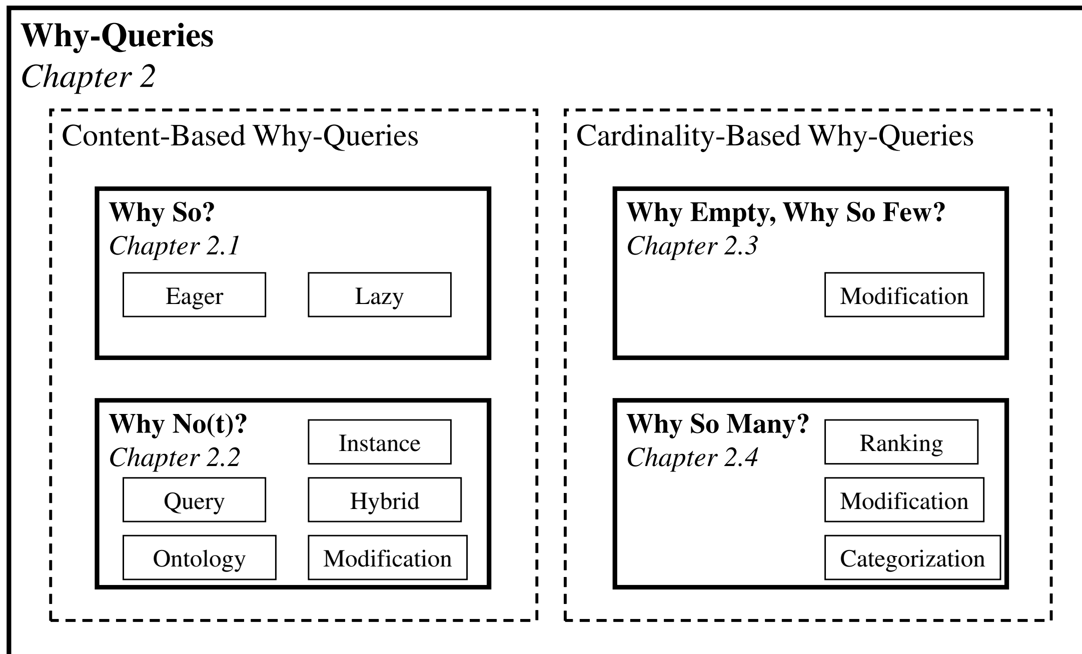


Figure 2.1: Overview of why-queries

2.1.1 Eager Provenance

Eager-provenance methods compute and keep provenance as long as data is transformed. The data is always annotated and querying provenance is exact and fast: the pre-calculated provenance only needs to be retrieved. Clearly, this creates a storage overhead and increases transformation and computation costs. This approach of maintaining provenance is also known as annotations, metadata support, source tagging, and the attribution approach [144].

Lee et al. [89] propose a mediator framework SAQSD for querying semi-structured data on the web. In this framework, each cell of a result is accompanied by its *attribution*, which is a “list of URLs for each Web document that was accessed during the processing of the query and from which the corresponding cell value was extracted” [89]. This system-generated attribute describes the location of a source. In this framework, a Polygon model [147] is used, which aims at resolving data source tagging and intermediate source tagging. According to this model, a user query is split in multiple local queries, which are redirected to particular sources. Each answer to a local query is annotated by its source.

Bernstein et al. [13] implement metadata support (MSDT) in Microsoft Server SQL 7.0 warehousing as a data transformation service with database models and data transformation models. For tracing lineage, a data transformation model is of interest that captures information about data transformations. A data transformation service stores the execution of each package in a repository and marks each generated row in a warehouse by the identifier of the package, which produced it. This metadata information is required for lineage tracing and extracting all necessary provenance information from the repository.

Bhagwat et al. [14] propose to keep data provenance as an annotation system with an SQL extension PSQL. Under annotations, Bhagwat et al. assume provenance, comments, or other types of metadata. PSQL supports three schemes for propagating annotations [14]: default, default-all, and custom schemes. While the *default* scheme distributes annotations about a data origin, the *default-all* scheme propagates also an-

notations for all-equivalent formulations of a given query. The proposed schemes are implemented in DBNOTES [40], which attaches a note to a value and transmits both during query evaluation. It also can derive lineage by request: in this case a reverse query is generated and the provenance is calculated lazily.

Buneman et al. [24] provide a formal study for an annotation placement problem (PDAV): which annotations in a database can cause an annotation to appear in a view with a minimal effect on other attributes. The formalization is provided for selection, projection, join, union, and renaming. Tan [128] studies the query containment problem for queries with annotations and proves that the annotation placement problem, which is known to be NP-hard [24], is DP-hard.

TRIO Conventional RDBMS usually store exact data, are based on the closed-world assumption, and do not support lineage. However, the TRIO project [2, 148] combines data, accuracy, and lineage in a single database, each of them can be queried separately and in conjunction. Lineage is implemented on a tuple level and is enabled by a specific handling of deleted data. No data is removed or overwritten in a database, any deletion and overwriting means a deactivation of a piece of data. Information about deactivated data can be used for querying lineage. The authors propose an SQL extension called TRIQL to query lineage and inexact data. In general, TRIO supports three types of lineage: historical lineage, phantom lineage, and versioning. Lineage for a tuple consists of three components: when and how a tuple was derived and what data was used for it. Five derivation types are supported in TRIO: query-based, program-based, update-based, load-based, and import-based. They describe the general ways how a data tuple can be derived. For example, import-based derivation describes tuples imported from external data sources by source descriptors.

Summary All discussed eager-provenance systems provide a model for storing lineage and generate different kinds of explanations. Only in the TRIO system [2, 148], lineage is considered as a first-class citizen allowing its direct querying and traversing and the type of a generated explanation depends on a used query type. Two systems, the TRIO system and annotations on views [24], consider a relational data model. Metadata support [13] uses an object-oriented data model, attributions [89] are investigated for semi-structural data. Eager-provenance systems constantly create and maintain data lineage for all data and therefore do not require user integration. Only in DBNOTES a user can define a custom scheme for storing lineage and such the storage overhead can be directly controlled by a user.

2.1.2 Lazy Provenance

Lazy-provenance methods compute provenance by request. While a naïve complete storage of lineage is infeasible with an increasing number of processing steps and data sources, Woodruff et al. [150] limit the stored information and propose lazy-lineage computation SFDL for object-relational DBMS. The authors [150] introduce the notion of weak inversion, which is used for calculating lineage with a wide range of weak inverse functions. Woodruff et al. propose also a verification process with a set of guarantees for the quality of generated lineage. To calculate lineage on the fly, a user needs to register used inverse and verification functions in a database system.

WHY-WHERE Provenance In addition to why-provenance, Buneman et al. [23] introduce a new aspect for relational and XML databases: where-provenance (WHY-WHERE). While why-provenance answers the question of existence of contributing data, where-provenance describes the location from which the data was extracted and is used to de-

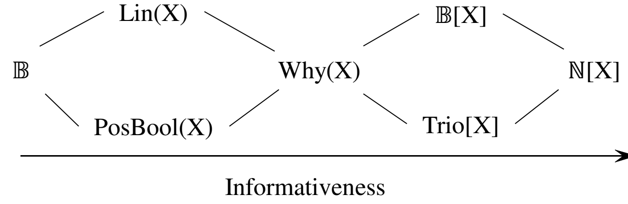


Figure 2.2: Provenance hierarchy (Source: Green et al. [55])

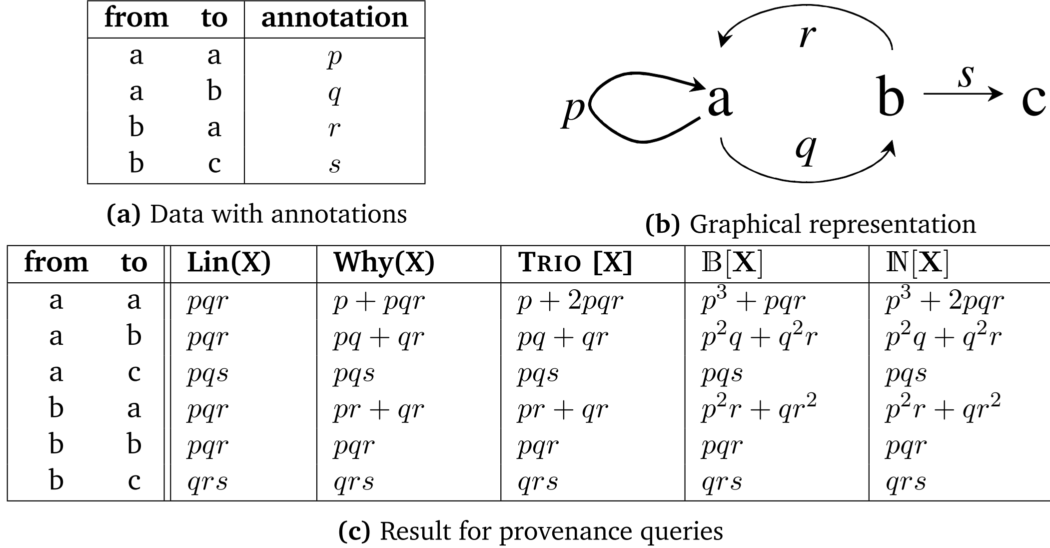
tect the source of failures in a query answering process [150]. The authors prove why-provenance to be invariant under query rewriting for SPJU-queries, while for where-provenance it can differ. The proposed provenance model [22] is deterministic and describes a location of each piece of data with a unique XML path that can be used as its identifier. Based on its nature, a path also stores structural information and therefore can be used to calculate lineage.

The problem of lineage tracing [43, 44] is common for data warehouses that integrate and transform information from different sources: these transformations are combined into an extract-transform-load process (ETL) and include algebraic operations, unit transformations, summarizations, and data cleansing. To query provenance, Cui et al. [43] construct a transformation graph (LTGDWT) from operations applied to a data item and its traversal. All transformations are classified according to how they map input and output, namely [43]: a dispatcher with a subclass filter, an aggregation with context-free and key-preserving aggregators, and a black box. Aggregations and dispatchers have known lineage tracing functions. For a black box, the entire input data set represents its lineage. For dispatchers and aggregations, schema information reduces the number of input items in lineage. Using properties of these transformations [43], lineage tracing can be optimized. Cui et al. [43] study fine-grained (or instance-level) lineage tracing, where original-source data items are returned as a lineage for a given warehouse data item.

ExDB ExDB [26] supports both eager- and lazy-provenance calculations for querying of web text and provides where-, why-, and how-provenance. It explains query answers on two levels: the query lineage like in TRIO [2, 148] and the extraction lineage like proposed by Cui et al. [43] and Buneman et al. [23].

Semiring Provenance (SEMIRINGS) The theoretical study of why-provenance [55] shows that annotated relations like uncertain or provenance-annotated data can be defined as commutative semirings. Following this proposal, Karvounarakis et al. [83] define commutative semirings as "algebraic structures $(K, +, \cdot, 0, 1)$ such that $(K, +, 0)$ and $(K, \cdot, 1)$ are commutative monoids, $\cdot$ is distributive over $+$, and for all a, we have $0 * a = a * 0 = 0$", where K-relation is a commutative semiring, which can be for example Boolean or natural numbers. A semiring can be represented by polynomials, which are used for calculating provenance for why-so queries.

Semirings of different K-relations create the semiring hierarchy illustrated in Figure 2.2 and introduced by Green [54]. Green defines six kinds of provenance over variable X with semiring semantic. The most general representation, polynomial provenance $\mathbb{N}[\mathbf{X}]$ [55], also captures how-provenance, which shows how each input tuple affects calculation of an output tuple (for example, trusted or untrusted data source). By removing coefficients, it derives Boolean provenance polynomial $\mathbb{B}[\mathbf{X}]$. Polynomial provenance $\mathbb{N}[\mathbf{X}]$ contains also TRIO [2, 148] semiring TRIO [X]. Both of



from	to	annotation
a	a	p
a	b	q
b	a	r
b	c	s

(a) Data with annotations

(b) Graphical representation

from	to	Lin(X)	Why(X)	TRIO [X]	$\mathbb{B}[\mathbf{X}]$	$\mathbb{N}[\mathbf{X}]$
a	a	pqr	$p + pqr$	$p + 2pqr$	$p^3 + pqr$	$p^3 + 2pqr$
a	b	pqr	$pq + qr$	$pq + qr$	$p^2q + q^2r$	$p^2q + q^2r$
a	c	pqs	pqs	pqs	pqs	pqs
b	a	pqr	$pr + qr$	$pr + qr$	$p^2r + qr^2$	$p^2r + qr^2$
b	b	pqr	pqr	pqr	pqr	pqr
b	c	qrs	qrs	qrs	qrs	qrs

(c) Result for provenance queries

Figure 2.3: Example of provenance calculation (Source: Karvounarakis et al. [83])

them contain why-provenance Why(X) [23], which is more informative than a positive Boolean expression used in incomplete databases PosBool(X) [70] and lineage semiring Lin(X) [44].

Assume the example provided by Karvounarakis et al. [83] in Figure 2.3. The example database consists of the table in Figure 2.3a, which describes links between nodes in a network. Its graphical representation is provided in Figure 2.3b. The *3hop* query derives such pairs of nodes (x, y) that node y can be reached from node x in three hops. There are six combinations of source and target nodes, which are on a *3hop* distance. They are comprised in the result table in Figure 2.3c. In this result table, provenance reports of five previously introduced provenance models are presented. We can conclude that the polynomial representation $\mathbb{N}[\mathbf{X}]$ [55] consists of all possible paths between two nodes and derives the complete provenance in contrast to other models. For instance, node b is reachable from node a via two paths: qrq and ppq. For further examples of semiring applications, interested readers are referred to the survey [83].

HOW-PROVENANCE While a polynomial representation of provenance is one of the most general models, Meliou et al. [95, 96] open a discussion on applying lineage calculation and its improvement for the use in database systems. The authors emphasize the abundance of how-provenance. What a user really needs is the cause of an answer (and non-answer), which is typically only a part of lineage. Therefore, Meliou et al. [95, 96] give a formal notion of causality that detects why a tuple is (is not) in the result set based on user-defined criteria. A user can distinguish which information is more crucial for him and can influence an answer generation. Meliou et al. also define a responsibility of a cause that describes how strongly it contributed to an answer. As a result, the causes of query results are returned, which are ranked by their degree of responsibility [96]. The authors show a high algorithmic complexity for causality and responsibility: it varies from PTIME and NP-complete for conjunctive queries with and without self-joins.

Summary In Table 2.1, the comparison of why-so methods is presented according to types of generated explanations and used data models. Lazy algorithms consider various data models and therefore they provide different explanations. Most of the

	Explanation	
Model	**Lazy Provenance**	**Eager Provenance**
Object-oriented	–	MSDT [13]
Relational	WHY-WHERE [23, 22], SEMIRINGS [55, 83], HOW-PROVENANCE [95, 96], DBNOTES [14, 40]	PDAV [24], TRIO [2, 148], DBNOTES [14, 40]
Semi-structured	–	SAQSD [89]
Object-relational	SFDL [150]	–
XML	WHY-WHERE [23, 22], SEMIRINGS [55, 83]	–
ETL	LTGDWT [43]	–
Web	EXDB [26]	EXDB [26]

Table 2.1: Overview of why-so methods

related work focuses on the relational data model. Lazy lineage calculation is also proposed for the object-relational model, ETL processes, and the XML model. Graph data models are rarely represented by the XML data model. One of the important aspects, user integration, is investigated in two cases. In the fine-granular lineage calculation proposed by Woodruff et al. [150], a user is required to define weak inverse and verification functions. By considering HOW-PROVENANCE [95], a user is required to define items of interest, which have to be inspected. Table 2.1 comprises also information about two systems which support eager- as well as lazy-provenance calculations. DBNOTES [40] derives annotations and provides custom annotation schemes for relational data allowing to configure which information has to be considered for lineage. In contrast, EXDB [26] does not investigate user integration and generates unique paths to a data source as an explanation for extracted web data.

2.1.3 Common Properties

In this section, an overview of related work for why-so queries have been presented, which comprises various domains like data warehousing, explaining data anomalies, access control, etc. Typically, these systems base on the open-world assumption, include imprecise data, or transform and integrate data from multiple sources. Therefore, such systems require explanations for the existence of some query results. The most used application for provenance querying is data warehousing. The second most used one is information extraction from the web, where the extracted and transformed data is stored with unique paths to original data sources. Only in a few cases, the user is integrated in the debugging process. Although a lot of work is done for querying data provenance, only a limited set of data models has been considered. To conclude, all presented solutions expose the following common properties:

1. *Efficient generation of explanations* is investigated in two forms: eager and lazy provenance. While eager provenance provides fast explanations, it requires additional storage overhead. Therefore, lazy methods are preferred, which do not

keep the provenance of the result in advance, but calculate it by request. For debugging purposes, lazy generation is more appropriate, because it considers provenance only for items of interest.

2. *User integration* is purely presented by why-so queries. A user can specify which explanation is required for which answer or which method should generate an explanation.
3. *Different kinds of explanations* are provided on different lineage levels and include complete and partial explanations, tuple-based and query-based explanations, etc.
4. The *problem discovery* why some elements exist in the result is done by studying the provenance of the data items. The corresponding explanation provides an evaluation path describing from which sources the items of interest are received.

To conclude, the field of why-so queries for relational data is intensively studied. However, the provenance of graph data is poorly investigated and could be the focus of the next generation of research on data provenance.

2.2 Why-Not Queries

While why-so queries explain why particular items are present in the result set, why-not queries investigate the problem of missing answers, which can be done on different levels starting from examining a query tree and data up to modifying the query. In this section, we first discuss possible causes of missing answers and then consider some solutions proposed in the state-of-the-art research projects.

2.2.1 Reasons of Missing Answers

There can be multiple reasons of missing answers such as (1) processing failures, (2) missing or uncertain data, or (3) an incorrect query definition.

Processing Failures

Incomplete answers can be returned because of processing failures like technical issues, schema mismatching in a distributed environment, or failures during data extraction and integration processes.

Technical issues like network delays, link failures, or source unavailability [6] can cause delays in query processing. For short delays, authors [6] propose query scrambling that adapts query execution on-the-fly in such a way that during query delays other query parts are processed. For long delays, a user is supported by partial results. These methods aim rather at hiding delays than at debugging missing answers.

Schema (Ontology) Mismatching can be a reason of imprecise answers in a distributed environment [98]. Mena et al. [98] estimate how much information can be lost by translating queries across multiple ontologies and annotate answers with a level of confidence, which is calculated based on a user-defined upper threshold. If a user is not satisfied with the answer, the tool [98] relaxes a query along multiple ontologies by using synonym, hyponym, and hypernym relationships. Specifically in data integration systems providing a unique access to data via a global-as-view schema [27], queries are automatically reformulated according to local schemas. This reformulation can cause conflicts on integrity constraints (perhaps, local data does not satisfy integrity constraints on a global schema)

and as a consequence a system can provide an incomplete answer. To prevent incompleteness, the system detects such conflicts and performs a minimal repair of data [27], which is implemented as minimal removal of tuples violating constraints.

Data Integration Failures can be caused by confusing questions in a data gathering process or a sensor malfunction. If an a priori known summary of raw data is available, then missing values can be potentially reconstructed in three different ways [151]: ideal-constrained, under-constrained, and over-constrained. In an ideal case, only one exact solution exists, the aggregated values are accurate and can be directly used to derive missing values. In an under-constrained case, aggregated values are not sufficient to infer all exact missing values. Multiple solutions can be found and an optimal one can be chosen by a user. In an over-constrained case, the aggregation values are not accurate or they are estimated. Therefore, no assignment of values to variables satisfies all constraints. The solution for this case is the best compromise.

Missing or Uncertain Data

Missing or uncertain data is a subject of incomplete or probabilistic databases. An incomplete database can have some partial relations, where only a part of each relation is known to be complete [90]. By querying such relations, a result set can be incomplete. Therefore, the research focuses mainly on the answer-completeness problem and determines whether an answer is complete even if a database is incomplete.

```
SELECT LIBRARY.title, LIBRARY.author
FROM LIBRARY, AUTHORS
WHERE LIBRARY.title=AUTHORS.title
```

Listing 2.1: Example query for incomplete databases

Levy [90] considers this problem as query independence from updates. Assume an example database with a set of books, where the table `LIBRARY` is complete only starting from 1950, the relation `AUTHORS` is complete. For the query in Listing 2.1, the answer can be incomplete because if we update the database with information about books published before 1950 the answer changes. An answer is surely complete for a refined query with additional constraints on a year. The identification of answer-completeness can improve query performance by reducing redundant querying of multiple data sources. For this purpose, Levy [90] compares the equivalence of query results before and after updates.

Incompletenesses in data on an attribute level can be produced by an inaccurate extraction of information, a heterogeneous or user-defined schema, or incomplete entries provided by a user and are modeled by null values. QPIAD [149] detects such missing attribute values and rewrites queries appropriately [149]. Two standard solutions include the `ALLReturned` algorithm [149], which in addition to exact answers delivers all tuples with missing values and the `ALLRanked` algorithm [149], which requests all tuples with missing values, predicts them, and generates answers based on this extended database. QPIAD [149] creates multiple rewritten queries for attributes with null values, selects rewritten queries, and orders them according to the number of relevant answers to an original query. Afterwards, the rewritten queries are executed according to their orders and their answers are provided as extensions to the result set of an original query.

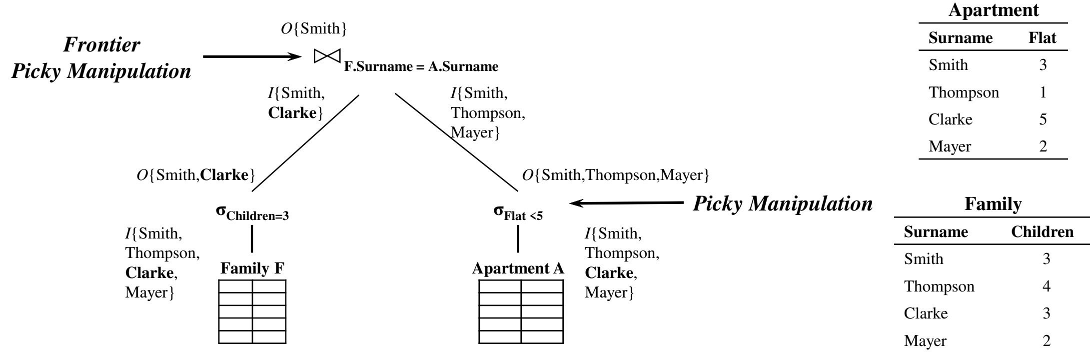


Figure 2.4: Query-based solution for why-not query: Why is family Clarke not in result set? Original query: SELECT Surname FROM Family AS F INNER JOIN Apartments AS A ON F.Surname=A.Surname WHERE F.Children = 3 AND A.Flat < 5

Query Incorrectness

The above presented systems dealing with incomplete or uncertain answers aim at preventing incomplete answers or at hiding side-effects like delays. For these systems, incomplete answers are natural and therefore do not require strong debugging capabilities. If the incomplete and missing answers are unexpected then the reasons of unexpectedness have to be detected and failed assumptions have to be fixed in a query. In this case, a user has formulated a query in such a way that some operations applied for selecting answers discard the items of interest.

Query incorrectnesses belong to failures done by users and should be supported by a debugging tool. Therefore, in this thesis we will focus mostly on this reason. To support users in such scenarios, two kinds of explanations can be generated: provenance-based and modification-based explanations. In the first group, the original query is not changed and the provenance of the missing answer is studied on query or data levels. In the second group, the original query is refined in such way that missing answers are delivered to a user.

In the following, we will discuss provenance-based explanations in Section 2.2.2 and modification-based explanations in Section 2.2.3. Then, several examples for specific query types and models will be presented in Section 2.2.4. Finally, common properties of methods for generating why-not explanations are discussed in Section 2.2.5.

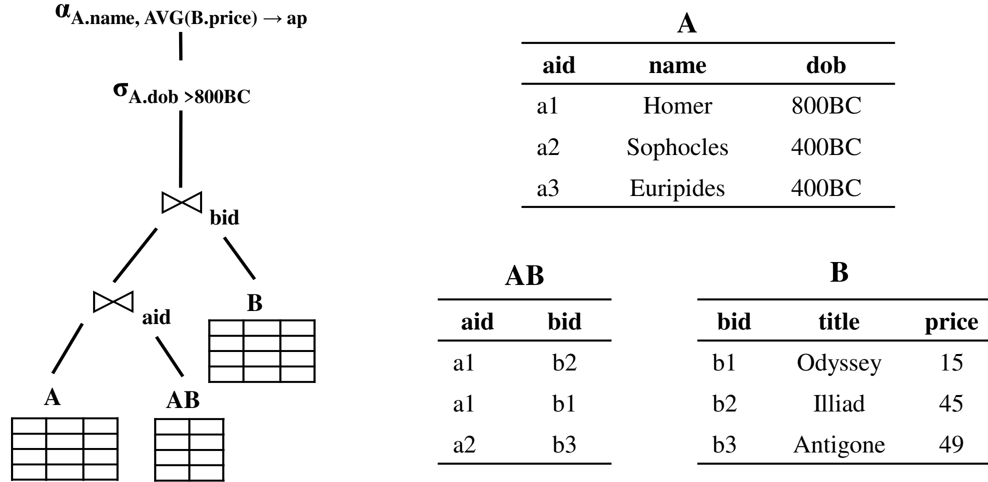
2.2.2 Provenance-Based Explanations

To detect the reasons of missing answers, provenance-based why-not queries study result provenance on query or data levels or consider both of them at the same time. Depending on the used explanation level, query-based, instance-based, or hybrid reports can be generated.

Query-Based Explanations

Query-based explanations describe the reason of a missing item in terms of query processing operations responsible for its elimination from a result set.

Why-Not In original why-not queries [32], a user is interested in debugging non-answers. It is assumed here that a user cannot process the data manually because of its large volume and complexity. Chapman et al. [32] propose a solution Why-Not for answering why-not questions based on a query tree. An answer includes operators of a



A

aid	name	dob
a1	Homer	800BC
a2	Sophocles	400BC
a3	Euripides	400BC

AB

aid	bid
a1	b2
a1	b1
a2	b3

B

bid	title	price
b1	Odyssey	15
b2	Illiad	45
b3	Antigone	49

Figure 2.5: Example for database and query tree (Source: Bidoit et.al [17])

query tree, which are responsible for eliminating tuples of interest from a result set and are called picky manipulations. The highest operator along a tree, which removes interesting tuples from the consideration is called a frontier picky manipulation and is responsible for missing interesting tuples. To detect picky manipulations, each operator is characterized by its lineage, which consists of input tuples contributed to output tuples. Assume an example query, its query tree, and an example database in Figure 2.4. Each operator is annotated with its input I and output sets O. The interesting item *Clarke* is discarded by two picky manipulations $\sigma_{Flat<5}$ and join ($F.Surname = A.Surname$), where the second one is a frontier picky manipulation. This approach works for SPJU-queries.

NEDEXPLAIN Bidoit et al. [17] discover several shortcomings in the WHY-NOT approach [32], which lead to inaccurate or even wrong results, namely: (1) An inaccurate selection of unpicked data, which becomes feasible if a query has self-joins. (2) An insufficient definition of successors, which results in independent tracing of data sources. (3) A restriction to frontier picky manipulation, which strongly depends on the query tree representation. (4) Insufficient answer details, which complicates understanding how each returned manipulation contributed to the rejection of interesting answers.

```
SELECT A.name, AVG(B.price) AS ap
FROM A, AB, B
WHERE A.dob > 800 BC AND A.aid = AB.aid AND B.bid = AB.bid
```

Listing 2.2: Example SQL query [17] to highlight shortcomings of traditional why-not definition [32]

To highlight some of the shortcomings of the WHY-NOT approach [32], Bidoit et al. provide the following example. Assume the database of books in Figure 2.5 with three tables: table A declares authors, table B presents books with prices, and table AB describes a foreign key relation between tables A and B. A user is interested in authors living before *800 BC* and average prices of their books (the corresponding query tree is illustrated in Figure 2.5, the SQL query is in Listing 2.2). The query engine delivers only a single answer *(Sophocles, 49)*. The absence of the tuple *Homer* with a price value *49* can be explained with its rejection by the second join, because there is no book with price *49* for *Homer*. Still, the WHY-NOT approach [32] generates no explanation, because of the used incomplete definition of unpicked data. Referring back to this

example, the output set for the second join includes *(Homer, 800 BC, Illiad, 45)*,*(Homer, 800 BC, Odyssey, 15)*, and *(Sophocles, 400 BC, Antigone, 49)*. All input tuples with *49* and *Homer* are picked, therefore, everything is fine, the algorithm traverses up the tree and misses the correct answer. This incorrect behavior is caused by the assumption that data sources are independent. To overcome shortcomings of the WHY-NOT approach [32], Bidoit et al. [17] formalize the process of answering why-not questions and provide the algorithm NEDEXPLAIN for generating why-not answers. This approach makes use of conditional tuples (c-tuples) to formalize why-not questions, which allows to check whether input-output tuples satisfy conditions in a why-not query. Moreover, they restrict the notion of a successor to valid successors with respect to a set of tuples which allows to detect the cases similar to the one provided in the example above.

Polynomials with TED and TED++ The above presented approaches strongly depend on the query tree representation and can therefore miss some explanations. Considering this drawback, Bidoit et al. [15, 16, 18, 19] propose to model why-provenance via polynomials, which is similar to why-so provenance, considered in Section 2.1. A why-not question is represented as a conjunction of conditions for a missing answer. In the previous example, a why-not query "Why is an item Homer with a price 49 not in the result set?" can be modeled as $name = Homer \wedge price = 49$. The authors propose two algorithms: TED [16] and TED++ [15] to generate why-not explanations. Bidoit et al. introduce compatible tuples that are generated from the data tuples according to why-not questions. Referring to the previous example, first compatible tuples are extracted, which satisfy $name = Homer \wedge price = 49$. Second, the algorithm enumerates them and checks which of them do not satisfy the original query conditions. In such a way, the algorithm generates explanations based on violated query conditions. Afterwards, all of them are put together into a single explanation via disjunctions. If different compatible tuples have the same explanations the polynomial representation can be improved by assigning coefficients to unique explanations, which show the number of tuples having a specific explanation.

The main drawbacks of the TED algorithm [16] are the enumeration of compatible tuples and the iteration over them. To prevent them, the TED++ algorithm [15] performs the following optimizations: (1) Instead of iterating over all compatible tuples, it iterates over possible explanations. This space is assumed to be smaller. (2) It derives partial sets for passing compatible tuples. (3) It calculates how many compatible tuples were eliminated for each explanation. For further details, we recommend the research work of Bidoit et al. [15, 19] to the interested reader.

Declarative Debugging Caballero et al. [25] propose a declarative debugging tool (DDWMA) of missing or unexpected answers for SQL views, which constructs a computational tree for failed views. Its nodes represent tables (relations) on leaves or they can also be computational trees of subviews. The tool [25] navigates the search along this tree by asking a user, whether a particular node is valid. Each invalid (buggy) node corresponds to an erroneous relation. The debugger collects buggy nodes and provides them as an explanation for an unexpected answer.

Instance-Based Explanations

Explaining a reason of a missing answer can also be done on a data level, which is typical for data extracted from external data sources with imperfect extractors that can fail or derive data with mistakes. To debug and fix data, databases have to support lineage [12], which describes a data item's derivation (inside the database of an external

source of information) and is used to generate instance-based explanations. They show how a data source has to be modified in order to deliver missing answers. Like in the previous section, these approaches are relation-oriented and do not consider specifics of a graph.

Missing-Answers (MA) In general, all query results can be classified in two groups: potential answers (they can become answers, if data updates are allowed) and never-answers (there are no updates that could turn a non-answer into an answer) [67]. To distinguish between them, Huang et al. [67] consider two types of sources and attributes: untrusted (their data can be fixed) and trusted. This consideration allows to limit rewriting decisions by the notion of trust such that modifications of less trusted sources are preferred. By varying (un)trust in specific tables and attributes, a user can explore the provenance space of non-answers. To answer a why-not query, the system requires a query, a list of trusted tables and attributes, and a non-answer. First, a user query is completed with implicit predicates and transitivity rules. Then, returned attributes are calculated for a user query (they can be potentially changed to create an answer from a non-answer). Afterwards, predicates on trusted tables and attributes are selected and non-corresponding tuples are filtered out. Finally, trusted predicates are joined with corresponding data sources if they are available. Untrusted tables are populated with null proxy tuples and the constructed provenance query is evaluated. As an explanation for a missing answer, the MA system [67] generates a provenance report that consists of a set of update tuples from which an answer can be potentially derived. If a tuple is a never-answer then its provenance query reports an empty provenance for it. In the example in Figure 2.4, a tuple $Clarke$ can become an answer, if we change the apartment number from *5* to *4* in the table *Apartment*, which was probably extracted incorrectly.

ARTEMIS Herschel et al. [65] propose the tool ARTEMIS for debugging and validating SQL queries using data. In addition to the MA approach [67] described above, ARTEMIS supports result ranking based on the number of tuples to be inserted in a database in order to make a non-answer an answer. It also supports user-defined constraints like immutable views. In further work [64], ARTEMIS was extended by handling aggregations and grouping in queries. In ARTEMIS, explanations are calculated by modeling a missing-answer problem as a set of constraints which are passed to a constraint solver. The solver [64] returns only such explanations which satisfy all constraints also if explanations require nulls, inequalities, or unique constraints. For modeling a constrained problem, conditional tuples (c-tuples [70]) are used. Herschel et al. [64] define a c-tuple as “a tuple $c - tuple =< a_1, a_2, \ldots, a_n, cond >$ such that every value a_i is either a constant or a labeled null. The attribute $cond$ is a Boolean expression”. C-tuples describe missing values which need an explanation. An explanation itself represents a finite set of c-tuples that is sufficient to receive all missing tuples. Still, a user has to decide which subset of explanations should be used to aggregate a final value. An original query is split in SPJ-operators which are pushed down and processed by the original algorithm for creating explanations [67]. Afterwards, their grouping and aggregation takes place. ARTEMIS supports also the generation of explanations for several tuples at the same time.

Provenance Games (PROVGAMES) Considering why-provenance on semirings, i.e., semiring-provenance, introduced in Section 2.1, we can also answer a question, why some tuples exist in the result set. While semiring-provenance is the most general

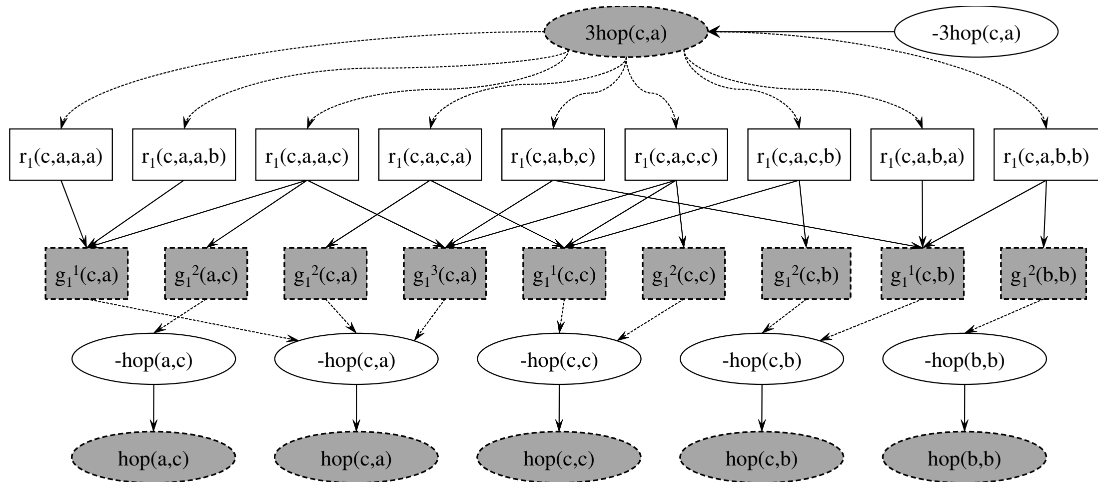


Figure 2.6: Why-not provenance for *3hop(c,a)* using provenance games (Source: Riddle et al. [121])

and suits why-provenance, it cannot model a negation, which is necessary for why-not provenance. Considering this fact, Köhler et al. [85] propose provenance games for modeling why- and why-not provenance. Assume the example in Figure 2.3. A user is interested in a *3hop* connection between nodes c and a, namely *3hop(c,a)*. This situation can be modeled as a game with two players, which can be represented as a graph $G = (V, M)$ with V positions and $M \subseteq V \times V$ edges. Both players move along edges of the graph [85]. The first player asks whether a particular connection exists, the second player has to prove this with instance data. A game is modeled as an iterative process, which terminates when one of the players cannot move anymore and therefore loses the game. The graph is colored according to the movements of the players. If a player position is correct it is filled with white color, otherwise, with gray color. For why-provenance, to show that a path between two data nodes exists (for example, *3hop(c,a)* exists), at least one of the leaves of a provenance graph has to evaluate to true (which would be then highlighted by the white color). To model why-not provenance, the first player starts with a negation of a tuple. An example of why-not provenance for $3hop(c, a)$ queries is taken from the research of Riddle et al. [121] and represented in Figure 2.6. To prove that *3hop(c,a)* is not correct, it is necessary to show that all leaves of a why-not provenance tree fail. For further examples, we recommend the research work of Riddle et al. [121] to the interested reader. Later Glavic et al. [52] revise this approach and propose summarizations for generating explanations. Namely, explanations with the same partial answers can be summarized and represented by an intermediate node in a graph. In other words, instead of traversing all possible connections, they can be generalized to an upper level with the use of inclusion rules and ontologies.

Hybrid Explanations

In some scenarios, above discussed algorithms do not find any solution, for example, if there are multiple inconsistencies, and the change of a table (for example) for a join predicate has to be tested. In this case, a system has to introduce a new tuple into a table, which is impossible if changing the data is not considered. Therefore, sometimes it is advantageous to combine different approaches, which is done in the CONSEIL system [60, 61] that generates hybrid explanations of missing answers.

The presented types of provenance-based explanations, which include query-based, instance-based, and hybrid reports, require user knowledge about underlying data and

schema. Based on them, a user can understand, which query part has to be modified in order to receive missing answers. Still, this can be a difficult task for an inexperienced user. Therefore, in such cases it can be more appropriate to refine a query automatically, where a user does not care about changing a query or data by himself with the help of an explanation. Indeed, a system improves it automatically.

2.2.3 Modification-Based Explanations

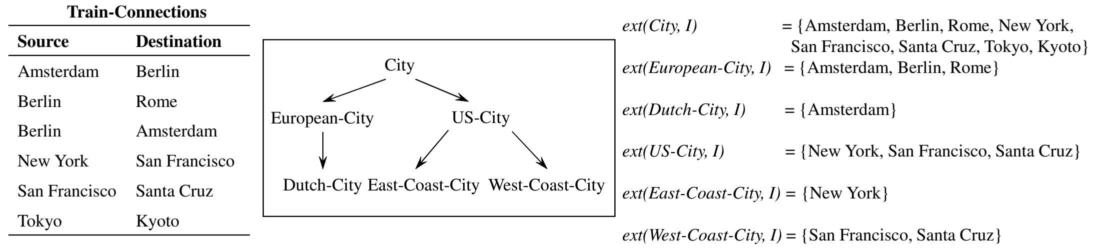
Methods of this group provide explanations in the form of modified queries, which deliver missing results. They can be classified mainly according to used integration strategies: automatic and navigational solutions. In the first case, no user is directly integrated, while in the second case, on each modification step a user feedback is requested. In navigational solutions, the search is steered by a user who can prefer some rewriting decisions or reject inappropriate query candidates.

Constraint-Based Query Refinement (ConQueR) Tran et al. [132] change a failed query in such a way that missing tuples are in the result set together with original answers. This automatic modification-based approach for SPJUA-queries is implemented in the ConQueR system, which generates a set of refined queries and modifies them further to produce skyline queries by applying additional predicates. First, the system modifies a query minimally on the level of selection predicates. If refined queries do not deliver missing answers the schema of a query is changed. The generation of multiple candidates may result in a performance bottleneck. Therefore, an optimization approach enabling simultaneous changes of multiple conditions is proposed. This is a similarity-driven approach, i.e., refined queries are compared based on the following metrics:

- A *dissimilarity metric* that describes how semantically different a refined query is from the input one. It is calculated as an edit distance by transforming an original query into a transformed one.
- An *imprecision metric* that compares the results of both queries. Ideally, a refined query should include only original results with why-not tuples. All other tuples are considered to be irrelevant and should be minimized.
- *Skyline refined queries*. The authors are interested in a set of skylined queries to propose them to a user. Assume two refined queries Q_1 and Q_2. First refined query Q_1 dominates over Q_2 if both its dissimilarity and imprecision metrics are the same or lower of the metrics of Q_2, and at least one of these metrics is much lower than the same metric for Q_2. The dominating refined queries are preferred.

As a reference model, Tran et al. [132] use the TALOS [133] system, which generates a query for a given set of output tuples. Given a database, a set of originally received and missing answers, TALOS generates a set of instance-equivalent queries that deliver the same result, but look semantically different. Therefore, instead of changing an original query, the system can try to generate a new one. Although this is technically a valid solution, a user can be confused by receiving a semantically different query.

FlexIQ Islam et al. consider user feedback during interactive query refinement in the FlexIQ framework [73, 74, 75]. A user feedback provides information of two kinds: expected answers that do not exist in a result set and unexpected results presented in a result set. At each iteration, the framework automatically refines a query in such a way that expected results appear and unexpected results disappear from a result set. For this purpose, the framework exploits a skyline operator for minimizing unexpected



Train-Connections

Source	Destination
Amsterdam	Berlin
Berlin	Rome
Berlin	Amsterdam
New York	San Francisco
San Francisco	Santa Cruz
Tokyo	Kyoto

Figure 2.7: Example of database table with train connections and ontology (Source: ten Cate et al. [131])

and maximizing expected answers. It collects feedback, calculates boundaries between expected and unexpected results, and refines a query inside new query bounds. Islam et al. also provide a pre-processing step making a refined query as semantically similar to the original query as possible.

Language-Level Debugging A language-level debugging is also proposed in HABITAT [56], which implements the basics of observational debugging. According to this paradigm, a user observes whether basic assumptions are valid and evaluated as expected. In HABITAT [56], a user marks any subquery to observe. The marked subqueries are executed by the query engine and their results are exposed to a user, who categorizes them as (in)valid. A user can further narrow the search to localize a reason of an error. At the end of a debugging session, an algebraic compiler generates a refined query based on collected debugging information.

To conclude, we considered provenance-based and modification-based explanations. All of them are complementary and can be used in a debugging or testing framework like for example in NAUTILUS Analyzer [62, 63], which relies on why- and why-not provenance and provides instance-based, query-based, and modification-based explanations. The NAUTILUS Analyzer is able to consider a user feedback for explanations, which can be annotated by a user as (ir)relevant. In addition, it also supports a simple ranking schema to prioritize explanations, which considers the number of data tuples involved into explanations, their similarity, and diversity.

2.2.4 Query-Type Oriented and Model-Specific Why-Not Queries

Until now, we considered general why-not queries in RDBMS for SPJUAG queries. Why-not queries can also appear in other use cases such as explaining failed queries in knowledge bases [31] or missing items of SPARQL queries [153]. In this section, we will have a look at several scenarios and give a short overview for why-not queries which consider characteristics of particular data models and query types like for example top-k and reverse top-k queries [51, 58, 59].

Ontology-Based Explanations (HLWN) Missing query answers can also be explained by high-level and meaningful explanations (HLWN) with the help of ontologies [131]. The main challenge taken here is to compute the most general explanation. Ten Cate et al. [131] provide the following example to make a reader familiar with the ontology-based explanations: Assume a database table with train connections between cities and the ontology presented in Figure 2.7. The database is described by the schema *S={Train-Connections (source, destination), Cities (name, population, country, continent)}*. A user is interested in all pairs of cities connected via a city, namely: *Train-Connections (source, destination)* $\wedge$ *Train-Connections (source, destination)*. A user may

wonder why the connection $(Amsterdam, NewYork)$ is not in the result set. The generated ontology-based explanations are presented in Listing 2.3.

```
E1=(Dutch-City,East-Cost-City)
E2=(Dutch-City,US-City)
E3=(European-City,East-Cost-City)
E4=(European-City,US-City)
```

Listing 2.3: Generated ontology-based explanations

According to generality, they can be sorted as follows: $E_4 >_o E_2 >_o E_1$ and $E_4 >_o E_3 >_o E_1$. E_4 is the most general explanation to be retrieved by the system. Such general explanations reduce the number of explanations and can be understood easier by a user. Ten Cate et al. [131] study the theoretical foundations of this problem and show that it is NP-complete. In some special cases, it can be solved in polynomial time. The question of missing answers (MA-DL) has been studied for querying over ontologies [28, 29]. The used techniques consider reasoning and are less relevant for relational and graph databases.

Answering Why-Not QuestioNs for SpArql (ANNA) The Resource Description Framework (RDF) [100] is a general data model originating from the semantic web community. It represents data as a graph expressed by a set of triples *(subject,predicate,object)* called RDF statements, where the subject and the object are nodes of a data graph connected by predicates introducing edges. This data is usually stored in triple stores implementing SPARQL [57] endpoints for its querying. Triple stores with open data are typically connected in the linked open data cloud and provide functionality for federated querying. This model is very general and allows to store and process different data, which complicates its querying. The problem of missing answers is very common by querying RDF data from triple stores with a SPARQL query interface. To explain them, the ANNA system [153] uses a divide-and-conquer principle and considers two levels of query processing: a basic graph pattern level and an operator level. The basic pattern level represents only the topology of pattern matching queries expressed by a set of RDF triples. The operator level includes SPARQL operators like `FILTER`, `DISTINCT`, `LIMIT`, etc. The reason of a missing answer can be a too restrictive basic graph pattern at the basic pattern level or questionable query operators on the operator level [153]. To generate an explanation, the ANNA system first checks whether a basic graph pattern with a missing item exists. If the pattern is not available, it is refined. Otherwise, the system identifies questionable query operators which are responsible for discarding items of interest from a result set.

Failed Queries in Knowledge Bases (WHYNOT) Chalupsky et al. [31] describe several reasons of failed queries, which are specific for knowledge bases: missing knowledge, limitations of used reasoners to derive knowledge, user misconceptions, or incorrect facts and rules, whose detection is complicated when the size of knowledge bases is large. In such settings, it is infeasible to derive an exact, correct proof. Therefore, Chalupsky et al. [31] propose to generate plausible partial proofs, which only approximate correct proofs and describe what could be a reason for a missing item. If a query fails and delivers *unknown* status, the proposed WHYNOT tool [31] checks which information is known to a knowledge base and which is unknown. Afterwards, it assumes what happens if a failed condition would be true or false, and provides a corresponding answer to a user. Although the proposed strategy delivers only partial proofs, they give sufficient information about missing parts of data or what could be a

reason of a failure. To avoid overloading a user with explanations, partial proofs are ranked according to their plausibility.

Top-K Queries (WHY-K) If some answers are missing for a top-k query, a user can ask [58, 59]: What has been chosen inappropriately? The number K, the weighting function, or both? To answer these questions, He et al. [58, 59] propose a modification-based approach, where an explanation represents a refined query which is characterized by two numbers showing how much K and a weighted function were changed. From two possible explanations, a dominating one is derived for a user. A refined query dominates over another refined query if both its refinements on K and a weighted function are smaller than that of the second candidate. Considering inefficiency of an exact search, He at al. [58, 59] propose a sampling-based method to find the best approximate answer. For this purpose, a randomly-sampled list of weighted vectors is generated from a weighting space. For each vector, a corresponding query is created, which is processed by a progressive top-k algorithm until an item of interest appears in the result set with a calculated rank. The mappings between discovered ranks and weighted vectors are collected. After all queries are executed, the best candidate with the lowest difference to the original query and weighting function is delivered to a user.

While top-k queries deliver K ranking results according to a user-defined weighting function, reverse top-k queries with respect to an item q return those preference functions from a given set of functions, which rate item q in first K items [51]. This type of queries is specifically useful in marketing research and recommendations. Similar to top-k queries, an explanation can be calculated mathematically over the space of preference functions. In addition, to a set of weighting functions and a number K, item q can be changed [51]. Why-not explanations are also proposed for keyword spatial top-k queries [36] that search for K best ranked geo-tagged objects on the web. Chen et al. [36] model the problem of missing geo-tagged objects as a two-dimensional problem, which allows to derive a geometrical model for ranking objects and is used for query refinement.

Sky-Not Queries (SKY-NOT) Why-not queries are also studied for skyline queries, called SKY-NOT queries [39]. By definition, a skyline query returns those data items, over which no other data items dominate across all attributes. This is a query for multidimensional analysis considering particular multidimensional space and returning extremes of this space. By receiving an answer, a user may wonder why particular items (points) are dominated. Therefore, a sky-not explanation changes the bounds of the query such that items of interest appear in the skyline [39]. To generate a sky-not explanation, Chester et al. [39] propose a bounded rectangle algorithm, which calculates a reduced set of candidates which puts an item of interest in the skyline and chooses the closest one to the original bounds.

The reverse skyline queries deliver those data points to a user, which skylines include a query point [71, 76]. Typically, these queries are used in marketing research and discover for example customers who have a given set of products in their skylines. If a user wonders why a product is not in the result set and therefore it is not well-located on the market, he can request a why-not reverse skyline query. A generated explanation includes minimal changes to a query point (a queried product) and a why-not point (a customer) which can include a why-not point in the result set of a reverse skyline query by preserving original answers [76]. Similar to why-not explanations for top-k and reverse top-k queries, sky-not and reverse sky-not explanations can be derived mathematically. This fact makes them different from the relational SPJUAG-queries.

	Explanation				
Model	**Query**	**Instance**	**Modification**	**Hybrid**	**Ontology**
Relational	WHY-NOT [32], NEDEXPLAIN [17], TED [16], TED++ [15, 18, 19], DDWMA [25]	MA [67], ARTEMIS [64, 65], PROVGAMES [52, 85, 121]	CONQUER [132], FLEXIQ [73, 74, 75], WHY-K [36, 51, 58, 59], SKY-NOT [39, 71, 76], HABITAT [56]	CONSEIL [60, 61], NAUTILUS [62, 63]	–
RDF	ANNA [153]	–	–	–	–
Graphs	WHYNOT [31]	–	PM [72]	–	–
Annotated data	–	–	–	–	HLWN [131], MA-DL [28, 29]

Table 2.2: Overview of why-not methods

Pattern Matching Queries (PM) The problem of missing answers has also been studied for pattern matching queries in graph databases, which maintain multiple data graphs. In this setup, an item of interest represents a data graph, which is missing from a result set [72]. Islam et al. [72] provide an approximate solution (PM) for generating why-not explanations for labeled graphs. To calculate a bounded search space, Islam et al. compute an approximate maximum common subgraph between a query graph and missing graphs. For this purpose, a query and missing graphs are decomposed in a set of stars, which then are mutually mapped. Afterwards, individual vertices can be mapped between each pair (a query star, a data graph star) and a maximum common subgraph is derived for each pair. Finally, the system composes a global maximum common subgraph from maximum common subgraphs for pairs (query, data graph), which represents the worst query refinement. Afterwards, an explanation is produced in two steps [72]: First, a system generates candidate edges, which can be added to or removed from a query graph. These edges are derived from the comparison of a query graph, data graphs, and a discovered maximum common subgraph. Second, from the generated set those edges are selected which minimize the distance to the original query based on a distance function.

2.2.5 Common Properties

In this section, we consider content-based why-not queries in RDBMS. A comparison of different methods is presented in Table 2.2, where discussed research projects are classified in five groups according to the type of generated explanations such as query-, instance-, modification-, ontology-based, and hybrid methods. In all these cases, users provide the items of interest, which are missing in the results. In addition, in some systems trust sources or buggy relations can also be defined. Most approaches consider the relational data model and specifics of a debugged query type. There is some initial

work done for graph data models. Still, it considers only RDF and multiple labeled graphs.

Referring to the state-of-the-art systems for processing why-not queries, the following common features can be extracted:

1. *Efficient generation of explanations* is investigated in different forms, for example: bottom-up and top-down approaches for processing relational queries, or rewriting methods focusing on minimal refinements.
2. *User integration* is presented in two general ways: a user provides the tuples of interest to be investigated or decides on refinement steps by generating modification-based explanations.
3. *Different kinds of explanations* investigated by why-not queries depend on a query type and include for example: provenance- and modification-based explanations. Some systems consider user proficiency and generate multiple explanations.
4. The *problem discovery*, why some elements are missing from the result, is done by studying the query tree and determining those query operators, which discard items of interest from the result.
5. *Query refinement* changes a query in such a way that it delivers items of interest to a user. This is a more advanced explanation than just providing manipulations, which are responsible for discarding items of interest.

To conclude, why-not queries aim at efficient discovery of reasons for missing answers and query refinement, which can be further improved by user integration. Modern debugging tools also try to provide various kinds of explanations in order to satisfy different user groups. While why-not debugging technologies are extensively studied for RDBMS, why-not research for graph databases is still in its early phase.

2.3 Why-Empty and Why-So-Few Queries

The problem of over-constrained queries was extensively considered in RDBMS as well as in graph database management systems (GDBMS). Such queries deliver too few or even empty results. A reason for this behavior can be a too strongly constrained query or missing data. This problem becomes a difficult challenge especially for an inexperienced user. As a consequence, solutions presented in the related work aim to discover a cause of an empty answer and to rewrite an input query to deliver any or more results (modification-based explanations) or to prevent the occurrence of such situations (prevention methods).

Relaxation of relational queries can be classified in two groups according to a user-integration strategy: automatic and navigational approaches. The first group comprises automatic methods to generate refined queries. It typically suffers from a large number of candidates that complicates the selection of the best one. Therefore, optimization techniques require the definition of flexible constraints or conflict resolution. The second group integrates a user directly in a relaxation process by asking whether a specific change is allowed, i.e., a user navigates a refinement process.

2.3.1 Modification-Based Explanations

CO-OP The problem of receiving empty answers was first tackled in the CO-OP system [80, 81], which executes queries expressed in a natural language. Let us assume the following query session presented in Listing 2.4.

Figure 2.8: Example of query representation in Meta Query Language

```
Request: Who passed an exam in geography in WS 2014/2015?
System Answer:  Empty answer.
Modified Request: How many students failed?
System Answer: 0.
Modified Request: How many students passed this exam?
System Answer: 0.
```

Listing 2.4: Stonewalling behavior

All these empty answers are contradictory and may mislead a user. This type of database behavior is called *stonewalling*. To deal with this issue, a system has to be able to detect it, to discover its reasons, and to provide meaningful answers to a user or an explanation. There can be two types of empty answers: '*genuine*' or '*fake*' answers [104]. In the first case, the empty answer is correct and there is no data corresponding to the query. In the second case, some of the query presuppositions failed; therefore, the query failed as well. To detect the correct reason of an empty answer, CO-OP evaluates all query presuppositions. In our example, the following pre-suppositions can be derived as presented in Listing 2.5.

```
There are some students.
An exam in geography took place in WS 2014/2015.
```

Listing 2.5: Pre-suppositions for example queries in Listing 2.4

The proposed query failed because of the second erroneous presupposition: no exam in geography took place in WS 2014/2015. This explanation can be given to a user instead of a misleading empty answer. CO-OP [80] supports an intermediate representation in the Meta Query Language (MQL). The query is expressed in MQL as a graph where each subject represents a node and a relationship is a connection between nodes. The example query is illustrated in Figure 2.8. Unsuccessful evaluation of any of its subgraphs leads to the failure of a query. An answer to a fake query includes an explanation constructed from failed presuppositions. A reply to a genuine query incorporates alternative answers of its presuppositions and hereby represents a straightforward query generalization approach, namely: removing failed presuppositions from a query.

SEAVE In addition to the discovery of erroneous presuppositions, SEAVE [103, 104, 105] proposes query generalizations that are query presuppositions on a logical level and are derived by relaxing some of the query constraints. All generalizations are stored in a relaxation lattice, where the less relaxed presuppositions are located on higher levels. The most general presupposition (extreme generalization) is placed on the bottom of the lattice and has all constraints relaxed. The upper part of a relaxation lattice for the example query is provided in Figure 2.9 where relaxed conditions are underlined. For the categorical constraints $*$ represents a relaxed categorical attribute. Its exact value depends on a relaxation strategy, a similarity measure, and a relaxation step. To detect, whether an empty result is genuine or fake and to provide a corresponding answer, three kinds of database knowledge are involved: facts, integrity constraints, and completeness assertions. Facts describe the data (entities

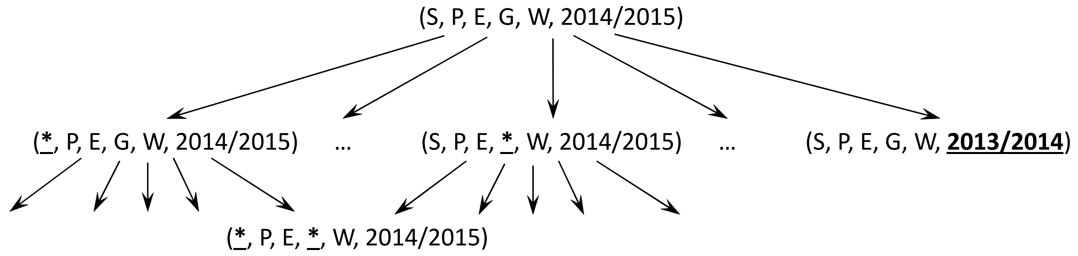


Figure 2.9: Query relaxation lattice

and their attributes stored in a database). Integrity constraints consider the relations between entities in a query. There should be no contradictions inside the query. The completeness concerns the closed-world assumption (the database keeps all the data about requested entities). The query presuppositions are tested against all these three constraints and corresponding explanations are derived to a user. The authors [104] propose an intermediate table representation including all the constraints and allowing their parallel evaluation. To conclude, SEAVE reports the maximally generalized failed queries that are assumed to be more informative answers for a user than just presenting a minimal failed subquery. Chaudhuri [33] defines extended generalized queries (GFQM) for solving the empty-answer problem, which provide additional conditions for the result set and examine some query constraints as flexible ones.

In general, CO-OP and SEAVE consider all the possible query generalizations to discover minimal failed and maximal succeeding queries. These problems are proved to be NP-hard [53], which leads to the necessity of using heuristics to reduce the search space of presuppositions and its complexity.

CoBase This system [41] relaxes a query with the assistance of a Type Abstraction Hierarchy (TAH) which is "a representation for the abstraction of individual types at instance level" [41]. An individual type can represent an atomic type or a tuple expressing multiple attributes at once. For example, an abstraction of a numerical type is an interval. On the bottom, primitive intervals consisting of individual values are placed, which are aggregated on upper levels. Each node of a hierarchy is annotated by a *nearness* value characterizing values below a node. An abstraction of a non-numerical type is a type of conceptual names like "North America" for values "USA" and "Canada". CoBase creates TAH in advance by calculating a relaxation error between the requested values and the returned values. To induce the best correlation between bottom non-numerical values, Pattern-Based Knowledge Induction (PKI) [99] can be used which derives clustering rules from the data. A tree is constructed in a bottom-up manner: First, atomic patterns are generated and their entities are placed on the bottom. Then rules are induced for each pair of attributes, where each rule is characterized by its popularity (frequency of its usage) and confidence (how well it fits the database). Those rules with the biggest clustering correlations (products of their confidences) are chosen from the generated ones and corresponding entities are clustered. To control the depth of a tree, an expert needs to assign a threshold for a clustering coefficient. In addition, weights for different attributes can be assigned in order to vary impact of different attributes on clustering.

LOQR and TOQR The above presented solutions focus on relaxing attributes without considering any dependency between them. In later research, relaxation strategies overcome this drawback. The LOQR algorithm [109] takes the advantages of machine

learning to discover hidden dependencies between attributes for query relaxation. This algorithm consists of three steps: (1) Decision rules describing implicit dependencies between attributes are learned online on a small randomly-chosen subset. (2) A rule that has the most similarities to a failed query is chosen from these detected decision rules. (3) The attribute values of a chosen rule are used to relax a failed query. LOQR works on queries in a normal disjunctive form with discrete and continuous attributes. This algorithm leads to short queries consisting only of a few constraints from a failed query. Considering this drawback, Muslea et al. provide an extended version called TOQR [110] which creates a new query starting from an empty one and adding new constraints from a failed query. The order of considering new constraints is determined by a domain's causal structure which is derived from a data sample.

Mining Functional Dependencies (MAFD) Considering dependencies between attributes of a query to deliver a larger result set is also studied for imprecise queries over web databases [111]. An answer to an imprecise query includes answers to its precise base query and an extended set, which represents results of similar queries. To decide which relaxations have to be used to meet a similarity threshold, approximate functional dependencies are calculated which describe how a change in a value of an attribute affects other attributes. The relaxation approach [111] assumes that tuples which are similar to precise answers have differences in the least important attributes. Therefore, such attributes can be relaxed first. The least important attribute is defined by Nambiar et al. [111] as "the attribute whose binding value, when changed, has minimal effect on values binding other attributes" and is detected by approximate functional dependency analysis [68].

Relaxation of Numerical SPJ-Queries (RNQ) Koudas et al. [87] relax SPJ-queries to solve an empty-answer problem, where selection conditions are defined over numerical attributes (this allows to calculate mathematically a relaxation cost induced by a relaxation). For each data tuple, we can calculate how a query has to be relaxed in order to be included in the result set and how high its relaxation cost is. Relaxation costs are calculated as a difference between existing and relaxed query conditions. A relaxation with equal or smaller costs along all conditions and with a smaller cost at least for one condition dominates over a compared relaxation. To guarantee that at least one tuple appears in the result after a relaxation, a relaxation skyline is calculated whose tuples are not dominated by any other tuple pair. Still, it can lead to many useless results. To prevent strong relaxations, Koudas et al. [87] construct a relaxation lattice, whose nodes describe how specific query conditions are relaxed, and calculate a skyline over a subset of conditions. Koudas et al. [87] discover a minimal relaxation and therefore propose methods to traverse the constructed lattice with a minimum relaxation and to calculate skylines for a subset of conditions.

To overcome the problem of too many query candidates, user interest can be involved in the relaxation [107, 108]. For example, Jannach et al. [78] calculate user-optimal query relaxations (RR) by evaluating subqueries and detecting conflicts for fast relaxation of preferred conflicts on demand. Junker [79] considers user preferences between constraints in the QUICKXPLAIN framework and evaluates preferred relaxations first. The main challenge taken in this research is to detect conflicts rather than a relaxation process itself. This algorithm starts with the most preferred constraints and adds new ones until the query fails.

IQR If a query delivers an empty result, the system generates several query proposals [107, 108] where only the user-defined flexible predicates are relaxed. The proposed relaxation framework builds a query relaxation tree by keeping non-flexible constraints and discarding others. A node of a tree represents a query proposal and is annotated by a probability to be accepted by a user. At each iteration, a user has to select one proposal to be checked on the delivery of a non-empty result. In such systems, a user navigates the search. A further improvement is to integrate user interest into the generation of proposals. In this case, the effectiveness of the proposal with respect to an objective goal and likelihood of its acceptance is considered.

2.3.2 Prevention Methods

This group of methods focuses on preventing the appearance of empty results instead of debugging a query after a problem occurred. To prevent the delivery of empty answers in an information retrieval system, Yom-Tov et al. [154] estimate the difficulty of a query for document retrieval, which describes the overlap of top results between different query terms. For this purpose, the contribution of each query term to a final result is calculated, where each term is expressed by a keyword. Based on the query quality, it can be estimated whether a query requests the missing query content. Therefore, such a query does not have to be evaluated.

To prevent a user from constructing queries delivering empty answers, flexible query answering on graph-modeled data [92] is proposed where approximate queries remove the burden in constructing correct queries from a user. The proposed approximate queries include approximation for both topology and vocabularies. In this framework, notational relatedness between vertices is calculated to control whether a connection between them is meaningful. Afterwards, incorrect annotations are fixed.

Schemaless and structureless graph querying [152] can prevent the problem of too few answers by automatically connecting user-provided keywords in a meaningful graph query. The transformation rules are similar to the ones used in approximate queries [92].

Another approach for flexible query answering is realized by a keyword search approach. Originally, a keyword search does not provide a way to specify the topology for subgraph isomorphism queries, but the structure is derived by the search of substructures matching the provided keywords. For example, Tran et al. [134] compute query proposals from the keywords. Afterwards, a user chooses a candidate to be processed by a query processor. These methods help a user to create a meaningful query and prevent him from defining a query delivering an empty answer.

2.3.3 Query-Type Oriented And Model-Specific Why-Empty Queries

In this section, we will give a short overview for relaxing a query in RDF stores and in a keyword search over RDBMS.

Query Relaxation in RDF (RELAXRDF) Query rewriting for empty-answer and too-few-answers problems over RDF data [49, 66, 117] focuses mainly on providing SPARQL language functionality for relaxed and approximate querying. SPARQL supports the RELAX clause defining flexible constraints for relaxation or removal, which is a part of the language standard. The APPROX clause, which is not standardized yet, changes constraints and facilitates the discovery of approximate results. Both clauses are used in path queries [117], where the rewriting is based on the RDFS inference rules and is introduced in a query language for semantic relaxation [49]. In both cases, a user can

	Explanation
Model	Modification
Relational	CO-OP [80, 81], GFQM [33], RNQ [87], SEAVE [103, 104, 105], CoBase [41], LOQR [109], TOQR [110], QUICKXPLAIN [79], IQR [107, 108], DNKSS [8], RR [78]
Web	MAFD [111]
RDF	RelaxRDF [49, 66, 117]

Table 2.3: Overview of why-empty methods

get additional answers to his query. These approaches require domain knowledge for approximations.

Keyword Search The problem of empty answers in keyword search can be caused by the data itself like missing keywords or synonyms, by the schema, or by joining several results [8]. To deliver at least some results, Baid et al. [8] search for the maximal partial matches and deliver them as partially correct results to a user. Before going into the detail of the solution proposed by Baid et al., first, we will recall the basic principles of keyword search. In RDBMS, a user poses a query consisting of at least one keyword. The system maps these keywords to relational tables and extracts corresponding information. Also if all keywords can be found in the tables, joining their corresponding tuples can lead to an empty answer. To provide a user at least with some results, authors propose a method to deliver to him those maximal subqueries, which deliver at least one item (DNKSS). To efficiently explain non-answers, Baid et al. propose a four-step approach. During the first online-phase a lattice is generated, whose nodes represent uninstantiated SQL queries. Next, the lattice is pruned by a given keyword query and the left nodes are instantiated according to provided keywords. At the third stage, the lattice is further optimized by keeping only answer and non-answer queries and their descendants. Finally, the lattice is traversed to collect non-answer queries.

2.3.4 Common Properties

As an explanation to the empty-answer problem, a user typically receives a refined query, which delivers at least some results. Therefore, this section mainly focuses on differentiation between automatic and navigational solutions for query modification. In addition, we also had a short look at prevention methods, which help a user to define a query. Still, these methods do not have a direct connection to debugging and therefore are not further considered in this thesis. As in the previous sections, we compare why-empty algorithms according to common properties:

1. *Efficient generation of explanations* is investigated as a reduction of relaxation space by considering some heuristics or integrating a user.
2. *User integration* is used in multiple methods. A user can navigate relaxation or relaxation decisions are taken automatically from the estimation of user intention and investigation of constraint flexibility in a query.
3. *Different kinds of explanations* include minimal failed queries and refined queries.
4. The *problem discovery*, why some elements are missing from the result, is done in cooperative systems, which can deduce a minimal failed query.
5. *Query refinement* changes a query in such a way that it delivers some results.

6. *Relaxation lattice* is a typical structure used for query relaxation, which represents the search space of query candidates. Its nodes correspond to refined queries describing minimal changes.

In addition to debugging properties we have seen for previously described why-queries, the why-empty queries are also characterized by a problem-specific feature—a relaxation lattice, which has to be efficiently traversed with optional consideration of user interest. However, this specific property is difficult to apply to graph queries, which consist of multiple dependent properties.

2.4 Why-So-Many-Queries

The too-many-answers problem can be solved by two kinds of methods: result-based and modification-based methods. The first group of techniques, result-based methods, improves representation of results without changing the number of answers with the help of ranking or categorization. The second group of techniques, modification-based methods, modifies (restricts) a query in such a way that the size of a result set is reduced and/or approximately corresponds to a cardinality threshold, if provided.

2.4.1 Result-Based Explanations

Result-based methods change the result representation for its better perception by a user. These approaches can be further classified according to used techniques such as ranking and categorization. In the first case, the results are ranked based on a scoring function, which treats different properties of the results differently: some properties can get higher weights than others. In the second case, the results are distributed across multiple buckets according to some properties, which can be explored by a user on his own.

Result Ranking

In this section, we will discuss the related work only for ranking the results of exact queries. Ranking of approximate queries usually considers how much information exists in the result set by comparing it with a query and therefore is not appropriate for solving the problem of too many answers.

Top-k queries are extensively used in areas which potentially face the problem of too many answers. According to the survey of top-k ranking techniques in RDBMS [69], they can be classified according to five criteria such as a query model, an implementation level, data and query uncertainty, query access, and a ranking function (see Figure 2.10). For more detailed information about this classification and corresponding related work, we recommend the survey [69] to the interested reader. In this section, we consider several examples of ranking-based explanations for different data models.

Attribute-Based Ranking Query results can be sorted according to user preferences, which describe how important different predicates are for a user. A preference model and a preference algebra are proposed by Kissling [86]. The preferences can be defined directly in a query using the PREFERRING clause proposed as an SQL-extension or can be derived indirectly from a partial order of predicate definitions in a query.

Agrawal et al. [4] propose an automatic ranking of query results (ARDQR) with *IDF Similarity* function, which is inspired by a term frequency-inverse document frequency (TF-IDF) from information retrieval: more frequent attribute values are ranked higher.

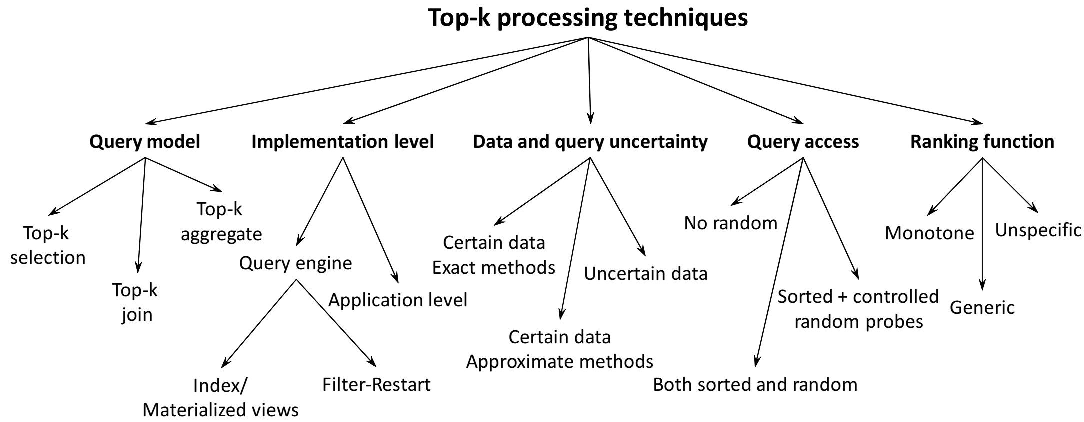


Figure 2.10: Classification of top-k solutions for querying RDBMS (Source: Ilyas et al. [69])

Agrawal et al. also suggest to use a workload for deriving automatic ranking of query results and the corresponding scoring function *QF Similarity*.

Su et al. [127] consider ranking query results for e-commerce web-databases with a ranking schema QRRE (Query Result Ranking over E-commerce). It considers user intention automatically by speculating the relevance of each attribute to a user. To allow ranking, the attribute values for all result tuples are quantified by their relevance to a user called *desirableness*. As a scoring function, the sum of relevancies of attribute values multiplied by the attribute scores is used. The main challenge of this proposal is to speculate relevance for attributes. Without a user feedback, all the relevant information can be derived from the data and the query. QRRE assigns higher weights to those attributes which correlate higher with attributes mentioned in a query. To assign relevancies to attribute values, Su et al. [127] make two assumptions, which are specific to e-commerce web databases: (1) a product with a lower price is more desirable and (2) a non-price higher relevant attribute has a higher price.

STAR [82] (System for Tuple and Attribute Ranking) explores a two-level result ranking scheme: on tuple and attribute levels, i.e., only top-k results are returned and only top-m most relevant attributes are presented to a user. As ranking methods, STAR explores a similarity-based ranking as an aggregated weighted similarity score and probabilistic ranking [35].

A ranking algorithm can consider not only the relevance of an item to a user and query, but it also can incorporate diversity of a result set in the scoring function [145]. A diversified result improves perception and understanding of the data and helps to formulate better queries. Relevance and diversity are opposite properties. To gain both of them, a trade-off has to be achieved, which can be modeled as a bi-optimization problem. Vieira et al. [145] give an overview on existing techniques for creating a diversified ranking result. As their own solution QRD, Vieira et al. [145] use a maximum marginal relevance to compute a maximum contribution of an answer to a result set.

Probabilistic Ranking (PIR) Chaudhuri et al. [34, 35] adapt probabilistic ranking models from information retrieval to rank database query results by automatically extracting user preferences and a scoring function from unspecified attributes. Here, a ranking function is considered in relation to global and conditional scores. While a global score describes a relevance of unspecified attribute values, a conditional score represents dependencies between specified and unspecified attributes. To estimate these scores, the authors adapt probabilistic information retrieval (PIR) ranking models.

In information retrieval, a ranking problem can be described as follows. Assuming a set of documents D, an answer for a query C consists of R relevant documents. Therefore, $\bar{R} = D - R$ documents are irrelevant. Ranking any document $d \in D$ according to a feature f is represented via conditional probabilities as follows

$$score(d) = \frac{p(R|f))}{p(\bar{R}|f)} \tag{2.1}$$

In RDBMS, a tuple is treated as a document from a result set $d \in D$ and score computation considers correlation scores and global scores derived from a workload and data analysis. The ranking is derived from two processing parts [35]: pre-processing and query runtime. At the pre-processing step, atomic conditional probabilities on data and workload are calculated for all distinct values similar to Equation 2.1 and stored in the intermediate representation layer. Afterwards, all ranked lists are pre-computed for all possible *atomic* queries (an atomic query specifies a query with a single value). At query runtime, all atomic ranked queries are merged in a threshold-algorithm fashion [35].

Best-K Queries (BEST-K) Tao et al. [129] propose a general framework for processing BEST-K queries that are a special kind of top-k queries, which consider result tuples as dependent and aim at minimizing redundancy and increasing diversity of answers. While a naïve approach supposed to be NP-hard, Tao et al. [129] propose two methods based on breadth-first and depth-first search with a branch-cutting pruning strategy for the discovery of exact answers and an approximate solution based on a depth-first search. To consider multiple constraints, Tao et al. [129] optimize a primary preference function under constraint functions, i.e. a constrained optimization problem. This approach reduces the search space and allows to terminate search earlier.

Context-Aware Ranking User preferences can evolve over time and be high-dependent on a context. Therefore, considering context information in a ranking function can be advantageous.

Agrawal et al. [3] integrate context preferences in a query answering process. A context preference describes in which context which subset of tuples is preferred to other specified tuples. A context can be modeled for example as a value of an attribute in a database. Different preferences belong to the same preference class, if they are defined in the same context. Agrawal et al. [3] propose a two-phase process (CSR) to calculate ranking. During an offline pre-processing step, for each context class an ordered list of tuples is produced. To reduce a storage overhead, the pairs of a context and an order are classified in several groups, where each group represents a disjunction of contexts and orders which are assumed to be a *good* representation of an order for each context in a group. The *goodness* of an order is defined in terms of a similarity with an original order for a specific context. During an online step, prepared orders have to be combined to provide appropriate ordering based on a query context. A context can also be integrated in database systems via special-purpose attributes called *context parameters* [126] (CAP), whose initialization with some values describes a context state. A user assigns basic context preferences to attributes in a database. At query runtime, several context parameters can be aggregated into a single preference score. Preference parameters are stored into data cubes and can be queried by OLAP techniques. Aggregated scores are stored in a context tree only for already executed queries, they are used to improve performance of new queries.

To derive user preferences, the context model can also consider uncertainty [136]. Therefore, van Bunningen et al. [136] define a probabilistic model of a context for document retrieval from a history of user choices and preliminary evaluated it on database views. Telang et al. [130] consider the problem of query result ranking in web databases, which store a large amount of information and ranking query results cannot be done by manual setting attribute relevance weights. The authors propose a solution (OSDNFA) which integrates both aspects into the ranking: user and query awareness. For this purpose, the system automatically learns user preferences from a browsing history of query results based on probabilistic data distribution difference, defines a similarity model from user and query similarities, and calculates rankings for new users and new queries based on this model.

Ranking RDF Graphs Querying of RDF graphs typically suffers from the too-many-answers problem, which especially concerns federated querying of the linked open data cloud with multiple data sources.

Ramakrishnan et al. [120] investigate a problem of most relevant paths between two entities in an RDF graph, which is modeled as a data subgraph, whose relevance depends on the amount of useful information it provides to a user [120]. The main contribution of this research (RELPATHS) concerns a ranking schema which includes (1) a class and property specificity, whose relevance is derived from its position in a type hierarchy (more specific resources are on the lower levels of a tree and they contain more information), (2) an instance participation selectivity, which depends on a number of type instances (rare facts are more informative), and (3) a span heuristic [120], which considers the involvement of multiple schemas to be more informative than the use of less schemas. After an RDF graph is annotated by these weights, the system calculates more informative paths between two given resources.

Aleman et al. [5] also investigate the problem of the most relevant paths between two resources by using the following metrics: a context, a subsumption (which is similar to a class and property specificity), a trust, a rarity (which is similar to an instance participation selectivity), and a popularity. The overall ranking (RCRSW) is considered to be a weighted sum of these metrics.

Anyanwu et al. [7] propose a SEMRANK algorithm for ranking complex semantic associations, which exposes two main features with a user integration: customizability (a user can choose a ranking schema) and flexibility (a user can compare results of different schemes).

Ning et al. [112] describe RSS framework for ranking queries over semantic web which consider a semantic web graph as a weighted directed graph, where a weight describes the strength of a property relationship. A ranking method considers a global resource importance which is derived from edge weights and a resource's relationships. For this purpose, RSS conducts a spreading process to determine the importance of nodes in a data graph. As starting nodes, those nodes are chosen which have the highest activation values computed from their global importances and query relevance of data nodes. These selected nodes are stored in an activation pool and extracted during a spreading process. After a query was processed, its final results are ordered according to activation values derived during spreading.

Ranking Subgraphs in a Large Attributed Data Graph Zou et al. [155] investigate pattern matching queries on a large labeled graph. A scoring function is defined as a sum of pairwise similarity between each vertex of a query graph and its matched data vertex. Zou et al. do not evaluate a similarity function, which is assumed to be provided

by a use case, and focus on efficiency of the ranked matching algorithm [155]. For this purpose, a G-TREE index is proposed, which is a height-balanced tree, consisting of node areas (sets of data vertices). Node areas of the same level belong to the same partition of a data graph. The tree-based index is constructed offline; it is used to prune node areas with low similarity scores from the search.

Pienta et al. [116] eliminate the index construction for a large data graph and propose the MAGE subgraph matching system, which supports multiple attributes on edges and vertices and provides a scalable multicore matching algorithm. In MAGE, edges are modeled by edge nodes describing their properties. The search routine first discovers matching initial nodes and performs a local search around them to detect neighboring nodes matching query criteria. Afterwards, edges are approximated between them. The ranking is a critical point of MAGE [116]. It is based on the proximity scores derived by the random restart algorithm. MAGE [116] focuses on approximate matching and does not consider ranking of the subgraphs exactly matching to a query.

Fan et al. [48] introduce a notion of an *output node* (OUTPUTNODE), which reveals the most important vertex in a query graph. An answer to a top-k pattern matching query represents a set of K vertices best matching to a query node by considering a query graph. For ranking results, two scoring functions are proposed [48]: a relevance function to measure the relevance of matches and a distance function to characterize the diversity of matches. Considering top-k diversified matching to be NP-complete, Fan et al. [48] propose an approximate solution with early termination conditions based on a diversification function, which aims at maximizing relevance and diversity of a result. In general, first discovered matches are located in a heap and a diversification function is calculated. If new matches appear, the function is recalculated. If it has a higher value, then matches are substituted by new matches.

Summary In contrast to why-empty queries, methods to solve too-many-answers problems have also been investigated for graph-structured data and represent labeled, attributed, and RDF graphs. In these methods, user interest is typically derived from a query or its context automatically and is typically incorporated in the scoring function. In several cases, a user is directly asked to provide feedback.

Categorization of Results

Information overload or the too-many-answers problem can be solved by categorization of query results and navigation of a user through them.

Chakrabarti et al. [30] propose an automatic navigation scheme (ACQR), which represents a tree of categories constructed from result tuples based on data attributes and their values. Each category (attribute) appears only at one level of a hierarchy. To create this tree on the fly, first the categorization space is estimated and an analytical cost model is defined that describes how well a particular categorization can satisfy a user. A cost, i.e., information overload cost, describes how many items (category labels and data tuples) will be exploited by a user and is estimated probabilistically. For this purpose, typical user behavior is extracted from a historical workload. A user can choose the most relevant category for him and traverse down the tree along the most relevant categories.

To improve a categorization, preferences of a particular user can be taken into consideration [37]. This observation faces two challenges: (1) how to summarize diverse user preferences from a workload for all users (2) and how to extract preferences for a specific user. To tackle these challenges, Chen et al. [37] split a query history in non-overlapping clusters offline, where each of them represents a specific preference. All clusters in PREFCLUS [37] are annotated with the probabilities of relevance for particu-

lar users. At runtime, by querying the system, a navigational tree is constructed, based on the queried attributes and in order of a cluster relevance to a user.

In further work [38], Chen et al. combine ranking and categorization techniques in PREFSKY. First, the results of a query are split into several groups. After a user has chosen one of them, its results are ranked. Ranking of SQL results is learned from training examples and includes the following information: skyline operators derive common preferences and navigational patterns exhibit dynamic and diverse preferences [38].

Clustering MUSIQLENS framework [91] takes three challenges in order to solve the too-many-answers problem: representation modeling (which and how many examples have to be shown to a user), a representative finding challenge (for the discovery of the best representatives for a model), and a query-refinement challenge. As a query result, the framework provides a list of best representatives selected from a result set, which exhibits its diversity. A user can navigate any of the provided items down and explore similar items. The main idea implemented in MUSIQLENS is to produce several clusters from a sample of a query result with the help of a cover tree data structure. As a result, on the first page a user receives a list of medoids. If a user chooses one of them to explore its results, a cover tree is consulted, which stores a set of neighboring results. As a final step, a query can be improved according to a chosen medoid and its neighbors. For this purpose, the system refines selection conditions and projections of a query.

Benchi [11] focuses on solving the too-many-answers problem based on clustering results for queries over distributed RDBMS. While a distributed query processing is not in the core focus, we will discuss a clustering algorithm (CAAQR) proposed by Benchi [11] in a single-machine environment. Clusters of result data are derived from the data summaries, which keep information about similar tuples and are organized in a hierarchical structure. If a user asks a query, relevant summaries of tuple clusters are extracted by a depth-first search along a hierarchy and derived to a user.

Faceted Navigation Navigation through query results can also be implemented as a faceted search like in DYNACET [10, 123]. This middleware between a user and a database navigates a user during a query answering process by asking questions about user interest. DYNACET aims at asking a minimal set of questions and builds a decision tree for this case, which exhibits an order, in which facets have to be proposed to a user. Each question corresponds to a facet and is represented by an attribute and its value. After a user has chosen a facet, the search continues inside this facet. A cost-driven faceted navigation approach is also proposed by Kashyap et al. [84] in FACETOR framework, where a user chooses value conditions for rejection of particular facets in order to reduce the number of answers.

Summary The discussed methods focus mainly on relational data and intensively incorporate a user in the categorization process. Typically, this integration is done at runtime: A user has to judge the proposed categorization, which is further refined by the system. As an explanation, these why-so-many methods provide result categorization, implemented according to the user interest.

2.4.2 Modification-Based Explanations

As a second type of explanations for the too-many-answers problem, a user can receive a modified query, which delivers an approximately required number of results.

Query Refinement in SQL (QRS) Ortega-Binderberger et al. [113] refine SQL similarity queries based on user feedback: a user can judge result quality and mark some tuples (or attributes) as good, neutral, or bad examples, which are stored in a feedback temporal table. This feedback is integrated in a query refinement process and a new query is generated by modifying a scoring rule and similarity predicates. Afterwards, answers to a newly refined query are proposed to a user. In general, the system supports two types of modifications [113]: inter-predicate and intra-predicate refinements. In the first case, scoring functions are modified by removing and adding new predicates and changing their weights. Inter-predicate refinement [113] means to re-calculate weights of each predicate in a scoring rule according to a user feedback in order to converge them to optimal weights. If no tuples were judged, no re-calculation takes place. If the same attribute has been differently judged for different tuples, then adding a new predicate can be important. A new predicate can be retrieved by studying judgments of ranked tuples. A deletion of a predicate is conducted if its weight in a scoring rule becomes lower than a threshold. In the second case, domain-specific similarity predicate functions are revised. Therefore, intra-predicate changes [113] are domain-specific, for example: query expansion or a query point movement for geospatial querying.

Keyword Search (MDKQE) Sarkas et al. [125] propose a solution for refining keyword queries over a set of documents. According to this proposal, a user is supplied with top-k expansions, i.e., word-sets, each of them contains additional search terms, which can be interesting to a user. To generate expansions, the notion of *surprise* is used as a measure of interestingness, which shows how a re-occurrence of specific terms differs from their independent re-occurrence. To rank expansions, Sarkas et al. [125] suggest three scoring functions considering a word co-occurrence pattern, an available extreme, and consistent user ratings.

Diversified Query Refinement Over Multiple Data Graphs (GQRD) Query reformulation can also be used for exploration of graph data: instead of overwhelming a user with too many results, he is supported with a set of query specializations [106]. A query specification in this case represents a modified input query, which includes one or two new aspects, which make it more special. By calculating specializations, two aspects are considered [106]: the result coverage for an input query and diversification of specializations. They are combined via a linear combination in an optimization function, which has to be maximized during the query reformulation. Mottin et al. [106] prove this optimization problem to be NP-hard, and propose a greedy algorithm with $\frac{1}{2}$ approximation guarantee.

Queries with Cardinality Assurance The problems of too few or too many results can be also seen as the problems of cardinality assurance, where a user is interested in a specific number of results. This user requirement is commonly used in two use cases: query testing and candidate selection, which are shortly discussed in Section 1. Cardinality assurance is tightly-coupled with the research about cardinality estimation, which estimates the size of an intermediate result to adapt a query plan execution in DBMS.

QRELX [137] aims at refining queries to meet cardinality constraints by considering *closeness* of a refined query to an original query in RDBMS. The framework consists of two key components: a transformer and an explorer. QRELX iteratively transforms a search space for SPJ-queries and explores this to discover refined queries, which meet

closeness and cardinality constraints. Discovered queries are reported to a user. If queries are not found, a new iteration begins with increasing a query search space. To enable efficient search for refined queries, cardinality is estimated incrementally and only for newly added data tuples.

Qarabaqi et al. [119] consider user queries as imprecise ones and propose a probabilistic-based framework (UDRIQ) to explicitly model constraint uncertainty. For each constraint in a query, a user can specify how confident he is about it in terms of a probability distribution. In addition to the probability distribution, a constraint can be characterized by its *sensitivity* and *benefit*. A sensitivity shows how strongly a result of a query can change if a constraint was modified only slightly. Inclusion of such conditions in a query should be avoided. A benefit of a constraint shows how a query result would benefit from its inclusion in a query. Based on these three properties, an exploratory search is realized as an interactive procedure with the following steps: (1) For each entity, its probability to be liked by a user is calculated. As a result, at this step a user receives a list of top-k entities ordered by their probabilities. (2) Sensitivities of all attributes are calculated and proposed to a user. (3) Benefits of all attributes, which are not included in a query, are calculated. After a user received this information, he decides how a query is further refined.

The satisfaction of cardinality constraints for conjunctive SPJ-queries [101, 102] incorporates a user feedback to capture user preferences in the STRETCH 'N' SHRINK framework, which enables stretching and shrinking the range of selection predicates to meet a cardinality constraint. For this purpose, a framework estimates maximum query transformations for each predicate and collects the necessary information for cardinality estimation. Afterwards, a query is refined such that all its constraints are maximally refined and the cardinality of its result is estimated with the help of a sampling technique. Finally, a user can choose relaxations of specific conditions and refine a query in such a way that it meets cardinality constraints. The process terminates when a wished query is constructed.

Summary Most of the discussed methods for generating modification-based explanations integrate a user in the refinement process. This integration allows to reduce the refinement space and to deliver tolerable modifications. Most of the work is done for relational data, which highlights the necessity of extensive investigation of modification-based explanations also for graph data.

2.4.3 Common Properties

All discussed solutions for why-so-many queries are analyzed in Table 2.4, according to the used data model and types of generated explanations. In general, five data models and three types of explanations are used. Why-so-many queries provide an additional type of explanations, result-based explanations, which is specific only for this query type and represented by ranking- and categorization-based reports. They focus on optimal representation of results for their better perception by a user. Modification-based explanations are also generated to solve the too-many-answers problem and represented for three out of five considered data models.

Why-so-many queries are intensively studied for the relational data model, where all kinds of explanations are provided. The solutions for graph models cover approximate matching for attributed graphs, diversified matching with output nodes, and rewriting labeled graphs. The area of rewriting of exact patterns for graphs with multiple attributes is not represented in the related work.

To conclude, why-so-many queries generate result-based and modification-based explanations and exhibit the following common features:

	Explanation		
Model	**Ranking**	**Categorization**	**Modification**
Relational	PREFERRING [86], ARDQR [4], QRRE [127], STAR [82], QRD [145], PIR [34, 35], BEST-K [129], CSR [3], CAP [126]	ACQR [30], PREFCLUS [37], PREFSKY [38], MUSIQLENS [91], CAAQR [11], DYNACET [10, 123], FACETOR [84]	QRS [113], UDRIQ [119], STRETCH 'N' SHRINK [101, 102], QRELX [137]
Web	OSDNFA [130]	–	–
RDF	RCRSW [5], RELPATHS [120], RSS [112], SEMRANK [7]	–	–
Documents	RQR [136]	–	MDKQE [125]
Graph	G-TREE [155], OUTPUTNODE [48], MAGE [116]	–	GQRD [106]

Table 2.4: Overview of why-so-many methods

1. *Efficient generation of explanations* is achieved by limiting the search space for example with user-defined flexible constraints or thresholds (like in ranking methods).
2. *User integration* is an important component of why-so-many queries, which is extensively studied. It facilitates generation of explanations and incorporates user interest in why-so-many queries. Ideally, user interest is automatically derived based on a use case or extracted from the context.
3. *Different kinds of explanations* include result-based and modification-based methods.
4. *Query refinement* changes a query in such a way that it delivers less results or a required number of results.
5. *Diversification of a result set* is a unique property of why-so-many queries, which is used by result-based methods, where the best diverse answers are delivered to a user to give him a deep overview on a result heterogeneity and to reduce redundancy in the results.

From already studied debugging properties in the previous sections, the user-integration feature is of utmost interest for why-so-many queries, which is mostly used to reduce the search space. In addition, why-so-many queries also focus on their unique property, the result diversity, which is not represented in other types of why-queries.

2.5 Summary

In this chapter, we had a deep look at debugging methods for unexpected answers in the state-of-the-art systems and extract the common features from them as presented in Table 2.5. In general, five out of seven features are provided by most of the systems, two last features are unique for some why-queries. While construction of a relaxation lattice is typical for why-empty queries, improvement of result representation is specific

Property	Why So?	Why Not?	Why Empty? Why So Few?	Why So Many?
Efficient generation of explanations	+	+	+	+
User integration	+	+	+	+
Different kinds of explanations	+	+	+	+
Problem discovery	+	+	+	–
Query refinement	–	+	+	+
Relaxation lattice	–	–	+	–
Result representation	–	–	–	+

Table 2.5: Common properties of why-queries

for why-so-many queries. This thesis proposes a general framework for solving three types of why-queries, therefore, we aim at providing only that functionality which is common among most of them. Therefore, only the first five commonly-used features are investigated in this thesis, namely efficient generation of explanations, user integration, different kinds of explanations, problem discovery, and query refinement.

According to the comparison in Table 2.5, most of the why-queries discover reasons of unexpectedness and/or refine the original query. In the first case, such an explanation is generated which describes an operation responsible for an unexpected result like query-based explanations produced by why-not queries or generates a responsible polynomial in why-so queries. In the second case, a failed query is modified to derive an expected result like in modification-based methods for why-not, why-empty, and why-so-many queries. In total, this thesis focuses on producing two kinds of explanations: query- and modification-based explanations, which discover the reasons of unexpectedness and refine the query, respectively.

We will also shortly discuss possible user-integration techniques, which are used to reduce the amount of modified query versions and thus improve the performance of generating explanations, and to provide user-relevant refinements.

In this chapter, we also considered data models, for which why-not queries have been investigated. Most of the research efforts intensively investigates debugging of unexpected answers for the relational data model. The recent work focuses also on graph models, which are mainly represented by RDF data and labeled graphs. Graph databases, especially those which model property graphs, are a new trend in database research. They allow storing and processing complex relationships, which are easy to represent in the form of a graph. However, only a few research groups inspect cardinality problems on a large graph with multiple attributes on edges and vertices, which is a novel data model used in modern graph processing systems for processing for example social networks.

To process such graphs is a challenging task which involves investigation of multiple constraints. Therefore, it is very likely for a user to construct a query delivering unexpectedly empty, too few, or too many results in this setup. Considering missing research about debugging queries for property graphs and difficulty of their processing, in this thesis we focus on investigating the basic debugging functionality extracted from

the related work on the example of property graphs. Specifically, we study cardinality-based why-queries for failed pattern matching queries, which are a fundamental type of graph queries.

In the next chapter, we will provide a definition for the property-graph model and propose similarity measures for judging explanations for why-queries in graph databases, which are later used in this thesis. Afterwards, we will cover the debugging features derived in this chapter, namely: the discovery of reasons for a failure and query refinement, which produce query- and modification-based explanations with the focus on their efficient generation and possible ways of integrating a user in the generation process by considering his feedback.

3

Why-Queries in Graph Databases

In the previous chapter, we gave an overview of the existing related work for explaining unexpected results retrieved by querying databases. To generate explanations, five types of why-queries exist such as why-so, why-not, why-empty, why-so-few, and why-so-many queries. Why-so and why-not queries investigate the presence of unexpected results and absence of expected ones and belong to content-based why-queries. Why-empty, why-so-few, and why-so-many queries focus on an unexpected size of the retrieved result set and are called cardinality-based why-queries.

For each type of why-queries, we presented existing solutions considering different data models and described which explanations they deliver to users. We also compared specific approaches for why-queries and extracted a set of debugging features they propose. While some of the features are common among all why-queries, others are unique for individual query types. In detail, most of the existing approaches focus on efficient generation of explanations and user integration, provide multiple kinds of explanations, discover reasons of unexpectedness, and rewrite a query such that it delivers better results.

To provide basic functionality for cardinality-based why-queries in graph databases, all these features have to be considered. In this chapter, we will describe their meanings for graph databases and give some basic definitions including the used property-graph model in Section 3.1. Then in Section 3.2, one of the presented debugging features, comprehensive comparison of explanations, is discussed in detail and evaluated.

3.1 Properties of Why-Queries in Graph Databases

In this section, we will describe four main debugging features, which correspond to the list of extracted debugging functionality from the related work and have to be realized in order to provide support for 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 going into detail, we first define the property-graph model as well as the supported graph queries in Sections 3.1.1 – 3.1.2.

3.1.1 General Graph Model

As an underlying graph model, we use a property-graph model [122] 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. 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. 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.

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) $u : E \to V^2$ is a mapping between edges and vertices; (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 spaces.*

In the following, we denote a data graph as G_d with M_d edges and N_d vertices.

3.1.2 Supported Types of Graph Queries

In addition to queries which are typical for RDBMS, graph databases support complex graph queries which implement graph algorithms like reachability queries, detection of a shortest path between two vertices, etc. Each of these queries can potentially derive some unexpected results. In general, graph queries can be classified in two groups according to the type of their results.

The first group delivers only a Boolean answer or a number like reachability and shortest-path queries. Therefore, the unexpected results can be described in terms of the result content. For example, two vertices are unreachable, although a user is sure that a path between two queried vertices exists. A shortest path can deliver a distance that is unexpectedly too short or too long. This thesis focuses only on cardinality-based why-queries, therefore, the graph queries of the first group are not considered.

The graph queries of the second group deliver a set of data subgraphs as results and include the community detection algorithms, pattern matching, or graph traversal queries. These queries represent property graphs whose properties are described by predicates for attribute values. They provide the most naïve and abstract representation of graph queries, which we extensively use in this thesis. For these queries, both content-based and cardinality-based why-queries can be defined and therefore this group of graph queries is in the focus of this research.

Specifically, we consider a pattern-matching query that is a fundamental graph query of the second group. It describes a pattern to be discovered in a large data graph and retrieves data subgraphs matching the pattern. A pattern itself represents a property graph, where edges and vertices are defined with predicates for their properties. In the following, we denote a query graph as G_q with M_q edges and N_q vertices. An answer to a pattern-matching query represents a set of data subgraphs matching a query graph, which is characterized by its cardinality.

3.1.3 Holistic Support of Different Cardinality-Based Problems

To provide explanation functionality for an unexpected size of the result for a specific query, we first specify cardinality problems, which are investigated in this thesis and their relations to the result cardinality.

Definition 2 (Result Cardinality). *We define result cardinality $C(G_q)$ of query G_q as the number of data subgraphs it matches.*

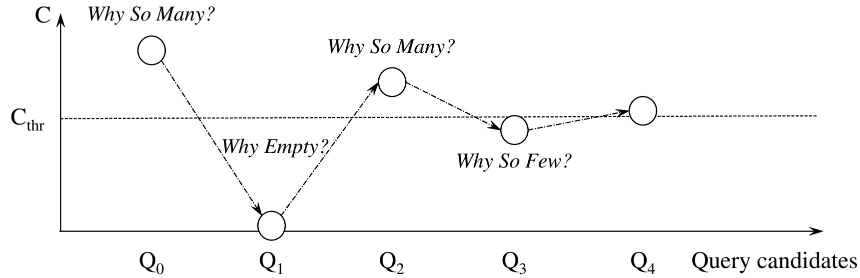


Figure 3.1: Holistic support of different cardinality-based problems

If a result set is empty, its cardinality is $C(G_q) = 0$. If too many results were delivered to a user, the result cardinality exceeds the threshold $C(G_q) > C_{thr}(G_q)$ (for too few results, $C(G_q) < C_{thr}(G_q)$). In general, a cardinality threshold can represent a cardinality interval with lower and upper cardinality bounds.

During the debugging and query rewriting, the size of the result can oscillate around the cardinality threshold as presented in Figure 3.1, where the original query delivers too many results. Its first refinement Q_1 delivers no result, the 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.

3.1.4 Explanation of Unexpected Results and Query Reformulation

One of the core functionalities extracted from the state-of-the-art systems summarized in Table 2.5 describes the discovery of 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 part of the processed (query) graph 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 solution is a subgraph-based solution 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, produces a refined query that was generated from an original query in such a way that it delivers results closer to the cardinality threshold compared with the results of an 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. We describe approaches for efficient generation of subgraph-based explanations in Chapter 4 and modification-based explanations in Chapters 5 – 6.

3.1.5 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 one, a similarity function has to be proposed, which would consider specifics of the property-graph model and integrate two aspects into

the same function such as topology and predicates. Moreover, explanations can be compared according to three characteristics: the content and the size of the results and the syntactic difference between explanations.

The comparison of the result content should show how the results of a compared explanation differ from the results of an original query, which can be derived from their overlapping parts. The cardinality distance should describe which explanation is closer to the cardinality threshold. The syntactic comparison should show how familiar an explanation appears to a user. This comprehensive comparison of explanations is used throughout the thesis to analyze the quality of explanations derived by different approaches. Therefore, it is described in advance in Section 3.2.

3.1.6 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, a 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 thesis, we present two models for user integration, which are described in Section 4.4 for subgraph-based explanations and in Section 5.4 for modification-based solutions.

In this section, we defined the property-graph model and described considered query types. We also gave a short overview on debugging features, which are studied in this thesis. In the following section, we will describe in detail one of them, comprehensive comparison of explanations, which compares two explanations based on three characteristics including syntactic, result, and cardinality distances.

3.2 Comprehensive Comparison of Explanations

In this section, we will discuss in detail one of the debugging properties presented above, comprehensive comparison of explanations, which is used across subsequent chapters for evaluating the quality of generated explanations. In general, any explanation for cardinality-based why-queries is a query graph, which describes a subgraph of an original query in subgraph-based algorithms or a refined query in modification-based methods. A subgraph-based explanation represents a query part that satisfies a given cardinality constraint. A modification-based explanation describes a query that delivers a required number of results and is generated from an originally failed query. In this thesis, we call a query, for which an explanation has to be generated, an *original query*.

Comprehensive comparison of explanations should consider three important aspects such as syntactic, result, and cardinality difference. The syntactic difference should describe how familiar an explanation appears to a user in respect to an original query. The cardinality difference should explain how the result size of an explanation differs from an expected cardinality and therefore it should show how well the goal of a receiving a specific cardinality is achieved. The result difference should focus on which part of original results is delivered to a user with an explanation. By considering this metric, we aim at producing not a completely new query, but at refining an existing one

and re-using some of its answers. In total, we will present three similarity measures for comparing explanations on these three levels and propose syntactic, cardinality, and result distances.

Before going into detail, a general graph edit distance approach for calculating dissimilarity between two graphs and its adaptation to the property-graph model are presented in Section 3.2.1. Then, we propose how to calculate syntactic, cardinality, and result distances for comparing pattern-matching queries in Sections 3.2.2 – 3.2.4.

3.2.1 Preliminaries

Modeling an explanation as a graph gives us a possibility to compare explanations based on the graph edit distance to an original query that shows how syntactically different they are.

A graph edit distance (shortly, GED) represents costs of transforming one graph into another one. Transformation operations usually include addition, removal, and substitution of edges and vertices and are characterized by transformation costs. One graph can be transformed into another one in multiple ways. Each transformation sequence is characterized by its own edit distance, which is derived by summing costs of executed transformations. The least expensive transformation sequence is used as GED between two graphs. The research on GED focuses mainly on learning costs of transformation operations and discovering the cheapest transformation sequence.

In the survey on GED [50], algorithms for calculating GED are classified according to used graph models: non-attributed and attributed graphs. Non-attributed graphs exhibit only the connectivity structure of a graph and can be encoded as strings. Therefore, a GED between two non-attributed graphs can be calculated as a string distance. In attributed graphs, GED is computed as an attribute distance retrieved from multidimensional label space. Gao et al. [50] emphasize the fact that GED is application-dependent and there is no general measure, which can be used across all applications.

In comparison to these data models, a property graph has both characteristics topology and predicates, which have to be considered during calculating GED. Therefore, in the following, we revise existing graph edit operations in order to use GED for property graphs.

Graph Edit Operations for Property Graphs A property graph has a twofold nature: both topology and predicates are important for the description of a graph. In addition to standard graph edit operations for a vertex and an edge, for a property graph it is necessary to consider several new operations for edge direction, types, and predicates. Any substitution can be modeled as subsequently executed deletion and insertion, therefore, this transformation is not considered in a further discussion.

A graph can be transformed by *basic* or *complex* operations. Each of them can belong to one of two classes: concretization and extension. If a transformation introduces new elements in the description of a graph, it is a *concretization* operation. Such operations are represented by insertions. If an edit operation reduces the description by deleting some graph content, it is a *relaxation* operation. Each relaxation operation has an inverse concretization operation. In Table 3.1, basic edit operations are classified according to how they change the description of a graph, where a target describes a modified graph element. The basic operations describe minimal modifications that can be applied to a graph query. The total number of applied basic operations can be used as a distance between an original query graph and an explanation.

Several basic modifications can be used together as a complex operation, which represents a more sophisticated modification and executes several changes at once.

Type	Target	Relaxation Operation	Concretization Operation
Topological modification	Edge	Edge deletion	Edge insertion
	Vertex	Vertex deletion	Vertex insertion
	Edge	Direction deletion	Direction insertion
Predicate modification	Edge, vertex	Predicate deletion	Predicate insertion
	Edge	Type deletion	Type insertion

Table 3.1: Basic modification operations

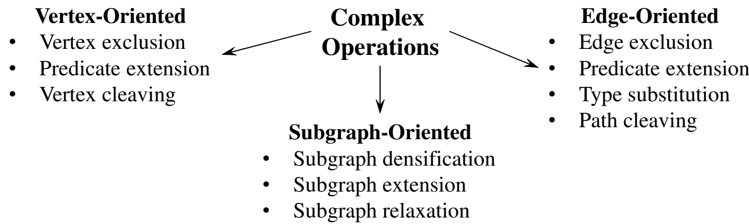


Figure 3.2: Classification of complex modification operations

Complex operations can be classified according to their targets as vertex-oriented, edge-oriented, and subgraph-oriented operations. Several examples of complex operations are provided in Figure 3.2. For instance, a subquery can be transformed by *subgraph densification* or *subgraph extension* operations. The first one increases the density of a subgraph: the number of vertices does not change, while the amount of edges increases. A subgraph extension increases both numbers of vertices and edges. Changing a predicate interval belongs to complex operations, because it includes two actions: a predicate deletion and an insertion of a new one.

The extended set of graph edit operations, which is suitable for property graphs, can be used for comparing explanations syntactically and calculating GED as a similarity metric. However, such a comparison would be coarse-grained because it is based only on the number of applied changes and does not consider fine-granular predicate modifications, which are an important aspect of the property-graph model. Therefore, a finer approach is required, which would consider predicate modifications by keeping the graph representation. Considering this fact, we propose to model a query and an explanation as a set of vertices and edges, which are subsets on their own. This proposal, which we present in Section 3.2.2, allows to keep a graph representation, consider fine-granular changes of predicates, and use well-known set distances from the related work.

The GED approach can be used only on a syntactic level, because it studies only a query graph and does not qualify how well an explanation achieves a cardinality threshold and which part of original results it delivers. However, for a comprehensive analysis of explanations all three aspects are important. Therefore, in this section, we also discuss these missing aspects and compare explanations according to the size and content of their result sets. We first consider the syntactic distance between explanations in Section 3.2.2 and then move to cardinality and result distances in Section 3.2.3 – 3.2.4.

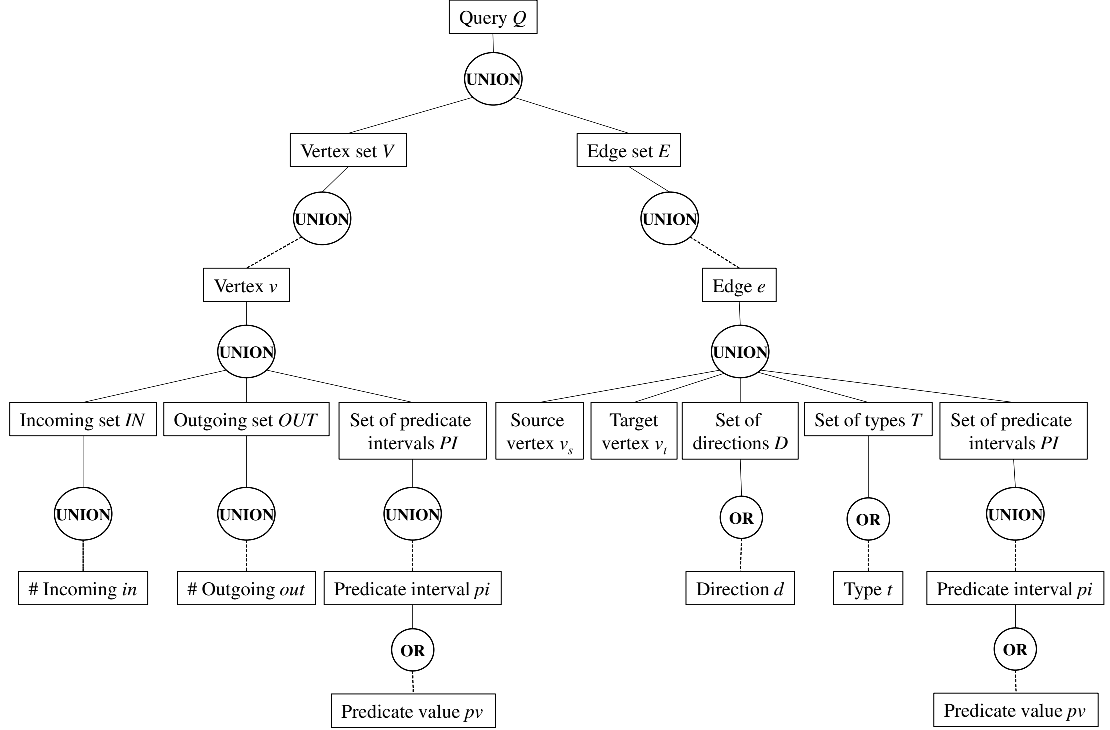


Figure 3.3: Set-based query model, where ovals define operations such as union and disjunctions of values and rectangles describe graph elements and their properties. For reasons of simplicity, all elements are drawn only once and dashed lines represent multiplicity of relations, for example: vertex set can consist of multiple vertices

3.2.2 Syntactic Level

As we discussed in Section 3.2.1, GED derives only a coarse-grained comparison, which does not consider fine-granular changes on predicates. Therefore, in this section, we provide a way for syntactic comparison of explanations, which overcomes this drawback. We study how different the explanations appear to a user in respect to an original query. For this purpose, we model an explanation and an original query as sets, which gives us an advantage to use well-known set distances as a syntactic similarity metric.

Set-Based Graph Representation

To support a fine-grained query comparison, in this thesis we define query Q as a set of vertices V_q and edges E_q (see Figure 3.3), where all vertices and edges are sets on their own:

$$Q = V_q \cup E_q \text{ , where } V_q = \bigcup_{i=1}^{n} v_q^i \text{ and } E_q = \bigcup_{j=1}^{m} e_q^j \tag{3.1}$$

A vertex as well as an edge can be annotated with predicate intervals for attribute values. For example, a query vertex representing a person, can be characterized by predicates for attributes like age, a name, or a profession. A predicate interval describes a set of values with upper and lower bounds, which an attribute can have. These attribute values pv are combined by their disjunction, because a data vertex can take only one value from this set:

$$pi = pv_1 \vee pv_2 \vee \cdots \vee pv_n \tag{3.2}$$

For instance, a query vertex is annotated with predicate $1 < age < 4$, which can be represented by predicate interval $age \in (1;4)$. This interval comprises two values, which age can have: 2 and 3.

By modeling a vertex as a set, in addition to predicate intervals $PI(v_q^i)$, we also consider sets of identifiers for its incoming $IN(v_q^i)$ and outgoing $OUT(v_q^i)$ edges. The union of these three sets fully describes a vertex:

$$v_q^i = PI(v_q^i) \cup IN(v_q^i) \cup OUT(v_q^i) \text{ , where} \tag{3.3}$$

$$PI(v_q^i) = \bigcup_{k=1}^{|PI|} pi_k(v_q^i) \text{ , } IN(v_q^i) = \bigcup_{t=1}^{|IN|} in_t(v_q^i)\text{, and } OUT(v_q^i) = \bigcup_{l=1}^{|OUT|} out_l(v_q^i) \tag{3.4}$$

Edge e_q^j is defined by the identifiers of its source v_q^s and target vertices v_q^t, a set of directions $D(e_q^j)$, a set of its types $T(e_q^j)$, and a set of its predicate intervals $PI(e_q^j)$:

$$e_q^j = T(e_q^j) \cup v_q^s \cup v_q^t \cup PI(e_q^j) \cup D(e_q^j) \text{ , where} \tag{3.5}$$

$$PI(e_q^j) = \bigcup_{k=1}^{|PI|} pi_k(e_q^j) \text{ , } D(e_q^j) = \bigcup_{h=1}^{|D|} d_h(e_q^j) \tag{3.6}$$

Each query vertex and edge has a numerical identifier $\in \mathbb{N}$ which is uniquely defined in an original query. In this thesis, we use these identifiers as indices i, j on query vertices and edges, respectively. For example, v_q^1 defines the query vertex with identifier $i = 1$. Vertex identifiers are used to specify source $v_q^s(e_q^j)$ and target vertices $v_q^t(e_q^j)$ for edge e_q^j. Edge identifiers are used to define incoming $in(v_q^i)$ and outgoing $out(v_q^i)$ edges for vertex v_q^i.

Similar to predicate intervals, a direction of an edge can be represented as a set, which can contain at most two values: backward or forward direction.

A type represents a special kind of an attribute, whose predicate can take only one value or a disjunction of values. Similar to predicate values, a data edge can have only one edge type, therefore, type values are combined by their disjunctions:

$$T(e_q^j) = t_1(e_q^j) \vee t_2(e_q^j) \vee \cdots \vee t_z(e_q^j) \tag{3.7}$$

Usually the number of types existing in a data graph is much smaller than the total number of data edges ($|T_d| \ll |E_d|$).

In general, the presented set-based definition of a graph query is illustrated in Figure 3.3. To define a query, its vertices and edges are represented as sets as well, which can be iterated down the tree to values of predicate intervals, types, etc. The proposed representation facilitates flexible modification of a query by easily removing or inserting disjunctions.

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 [45].

In general, a set represents a collection of data points. Therefore, to calculate the distance between two sets A and B, distances between their points $\forall a \in A$ and $\forall b \in B$ have to be derived, which are called point-point distances. They can be expressed by any similarity measure from the related work, for example: a string or synonym distance between categorical attributes, a numerical distance between two numerical values, or a Boolean distance:

$$d(a,b) = \begin{cases} 0\text{, if } a == b \\ 1\text{, otherwise} \end{cases} \tag{3.8}$$

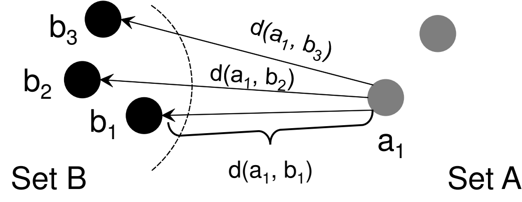


Figure 3.4: Point-set distance $d(a_i, B)$

where $a \in A$ and $b \in B$.

In this thesis, a point is presented by a small letter, a corresponding set is denoted by a capital letter. In Figure 3.4, two data sets A and B are presented, where $d(a_1, b_1)$ is a point-point distance between $a_1 \in A$ and $b_1 \in B$.

In order to calculate a distance between two sets, we also need to introduce the notion of a point-set distance.

Definition 3 (Point-Set Distance). *We define a point-set distance $d(a, B)$ between point $a \in A$ and set B as a minimal point-point distance $d(a, b)$ between a and any point $b \in B$.*

To calculate a point-set distance $d(a_1, B)$ between point a_1 and set B, all point-point distances between point a_1 and points $\forall b \in B$ have to be derived. Following the example in Figure 3.4, distance $d(a_1, b_1)$ is the minimal one among distances $\{d(a_1, b_1), d(a_1, b_2), d(a_1, b_3)\}$ and is considered to be a point-set distance $d(a_1, B)$.

If a point-point distance is modeled as the Boolean function as described in Equation 3.8, then a point-set distance based on it can be calculated as following:

$$d(a, B) = \begin{cases} 0, \text{ if } a \in B \\ 1, \text{ otherwise} \end{cases} \tag{3.9}$$

After all point-set distances are calculated, the set-set dissimilarity $d(A, B)$ can be derived as the commonly used modified Hausdorff distance [45]. The evaluation conducted by Dubuisson et al. [45] shows that the value of this distance function "increases monotonically as the amount of difference between two sets increases, and it is robust to outlier points".

Definition 4 (Set Distance). *Given two sets $A = \{a_1, a_2, \dots, a_m\}$ and $B = \{b_1, b_2, \dots, b_n\}$, where $m, n \in \mathbb{N}$, set distance $d(A, B)$ is calculated as a modified Hausdorff distance:*

$$d(A, B) = MHD(A, B) = max(\frac{1}{|A|} \sum_{a_i \in A} d(a_i, B), \frac{1}{|B|} \sum_{b_j \in B} d(b_j, A)) \tag{3.10}$$

The modified Hausdorff distance [45] is an asymmetric measure and hereby it is evaluated for both directions: $A \rightarrow B$ and $B \rightarrow A$.

Syntactic Query Comparison

To syntactically compare two explanations, we calculate their set distances to an original query. For this purpose, the set distances of their subsets such as predicate intervals PI, in-sets IN, out-sets OUT, types T, and directions D are derived as distances between two sets. Then, they are aggregated in the distances of their vertices and edges. Afterwards, the distance between an explanation and its original query can be determined. An explanation with a lower syntactic distance to its original query is better than an explanation with a larger syntactic distance.

To calculate vertex and edge set dissimilarities, we propose to apply the well-known modified Hausdorff distance [45] to above mentioned subsets, which are then aggregated into the distances of individual vertices and edges. A distance between two vertices $v^i_{q_1}, v^j_{q_2}$ is an average set-set distance for predicate intervals and sets of identifiers for incoming and outgoing edges:

$$d(v^i_{q_1}, v^j_{q_2}) = \frac{\begin{array}{c}\sum_{pi}^{|PI(v^i_{q_1}) \cup PI(v^i_{q_2})|} d(pi(v^i_{q_1}), pi(v^i_{q_2})) \\ + d(IN(v^i_{q_1}), IN(v^i_{q_2})) + d(OUT(v^i_{q_1}), OUT(v^i_{q_2}))\end{array}}{|PI(v^i_{q_1}) \cup PI(v^i_{q_2})| + 2} \tag{3.11}$$

A distance between two edges is derived as an average set-set distance for predicate intervals, types, directions, and source and target vertices:

$$d(e^i_{q_1}, e^j_{q_2}) = \frac{\begin{array}{l}\sum_{pi}^{|PI(e^i_{q_1}) \cup PI(e^j_{q_2})} |d(pi(e^i_{q_1}), pi(e^j_{q_2})) + \\ \quad d(T(e^i_{q_1}), T(e^j_{q_2})) + \\ \qquad d(D(e^i_{q_1}), D(e^j_{q_2})) + \\ \qquad\quad d(v_s(e^i_{q_1}), v_s(e^j_{q_2})) + d(v_t(e^i_{q_1}), v_t(e^j_{q_2}))\end{array}}{|PI(e^i_{q_1}) \cup PI(e^j_{q_2})| + 4} \tag{3.12}$$

After calculating distances for all vertices and edges, they can be aggregated into a set dissimilarity for sets of vertices and edges and dissimilarity of two queries.

$$d(Q_1, Q_2) = \frac{\sum_{i=1}^{|V_{q_1} \cup V_{q_2}|} d(v^i_{q_1}, v^i_{q_2}) + \sum_{j=1}^{|E_{q_1} \cup E_{q_2}|} d(e^j_{q_1}, e^j_{q_2})}{|V_{q_1} \cup V_{q_2}| + |E_{q_1} \cup E_{q_2}|} \tag{3.13}$$

To summarize, the syntactic distance between an original query and its explanation is derived according to Algorithm 1. First, modified Hausdorff distances are calculated for each subset of properties for vertices and aggregated in distances for individual vertices at lines 3 – 16. In a similar way, distances for edges are derived at lines 17 – 38. Finally, the syntactic distance between an original query and its explanation is calculated from all pre-computed distances at line 39.

This calculation model can be further generalized by assigning weights to specific edges, vertices, and their properties. These weights could express their relevance for a user. Since a property graph is an underlying data model considering the equal importance of topology and attribute descriptions, all subsets are considered as equally important.

Example Assume original query Q_1 in Figure 3.5a and its refined version Q_2 sketched in Figure 3.5b. Here, we present the detailed distance calculation only for vertex v_2 and edge e_1 in order to derive the syntactic distance between original query Q_1 and its explanation Q_2. Since the original predicate interval $pi(type, (university))$ of size 1 was relaxed to $pi(type, (university, college))$, it has an effect on the distance. According to Equation 3.10, predicate interval distance for vertex v^2 is derived as follows:

$$d(pi_{type}(v^2_{q_1}), pi_{type}(v^2_{q_2})) = max\left(\frac{0+1}{2}, \frac{0}{1}\right) = \frac{1}{2} \tag{3.14}$$

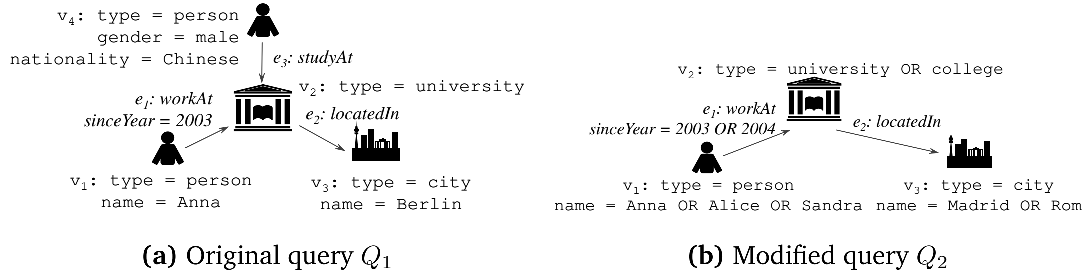
Similar, the set of incoming edges has also been changed by removing edge e_3, which gives us

$$d(IN(v^2_{q_1}), IN(v^2_{q_2})) = max\left(\frac{0}{1}, \frac{1+0}{2}\right) = \frac{1}{2} \tag{3.15}$$

The distance $d(OUT(v^2_{q_1}), OUT(v^2_{q_2})) = 0$ because the set of outgoing edges remains the same. Having calculated all the subcomponents of v^2, its aggregated dissimilarity is derived according to Equation 3.11: $d(v^2_{q_1}, v^2_{q_2}) = \frac{1}{3}$ (3.16).

Algorithm 1 Syntactic-distance calculation between original query and its explanation

Input: original failed query Q_1, explanation Q_2
Output: syntactic distance between original query and explanation $d(Q_1, Q_2)$
1: $V \leftarrow$ create vertex union
2: $E \leftarrow$ create edge union
3: **for all** $v \in V$ **do**
4: $PI \leftarrow$ create union of vertex predicate intervals
5: **if** $v \notin Q_1$ **then**
6: $d(v_{q_1}, v_{q_2}) \leftarrow 1$
7: **else if** $v \notin Q_2$ **then**
8: $d(v_{q_1}, v_{q_2}) \leftarrow 1$
9: **else**
10: **for all** $pi \in PI$ **do**
11: $d(pi_{q_1}, pi_{q_2}) \leftarrow$calculate MHD ▷ Equation 3.10
12: $d(PI_{q_1}, PI_{q_2}) \leftarrow d(PI_{q_1}, PI_{q_2}) + d(pi_{q_1}, pi_{q_2})$
13: $d(IN_{q_1}, IN_{q_2}) \leftarrow$calculate MHD ▷ Equation 3.10
14: $d(OUT_{q_1}, OUT_{q_2}) \leftarrow$calculate MHD ▷ Equation 3.10
15: $d(v_{q_1}, v_{q_2}) \leftarrow$calculate distance between two vertices from $d(PI_{q_1}, PI_{q_2})$, $d(IN_{q_1}, IN_{q_2})$, $d(OUT_{q_1}, OUT_{q_2})$ ▷ Equation 3.11
16: $vertices[\,] \leftarrow$ insert $d(v_{q_1}, v_{q_2})$
17: **for all** $e \in E$ **do**
18: $PI \leftarrow$ create union of edge predicate intervals
19: **if** $e \notin Q_1$ **then**
20: $d(e_{q_1}, e_{q_2}) \leftarrow 1$
21: **else if** $e \notin Q_2$ **then**
22: $d(e_{q_1}, e_{q_2}) \leftarrow 1$
23: **else**
24: **for all** $pi \in PI$ **do**
25: $d(pi_{q_1}, pi_{q_2}) \leftarrow$calculate MHD ▷ Equation 3.10
26: $d(PI_{q_1}, PI_{q_2}) \leftarrow d(PI_{q_1}, PI_{q_2}) + d(pi_{q_1}, pi_{q_2})$
27: **if** $v_s(e_{q_1}) == v_s(e_{q_2})$ **then**
28: $d(v_s(e_{q_1}), v_s(e_{q_2})) \leftarrow 0$
29: **else**
30: $d(v_s(e_{q_1}), v_s(e_{q_2})) \leftarrow 1$
31: **if** $v_t(e_{q_1}) == v_t(e_{q_2})$ **then**
32: $d(v_t(e_{q_1}), v_t(e_{q_2})) \leftarrow 0$
33: **else**
34: $d(v_t(e_{q_1}), v_t(e_{q_2})) \leftarrow 1$
35: $d(T_{q_1}, T_{q_2}) \leftarrow$calculate MHD ▷ Equation 3.10
36: $d(D_{q_1}, D_{q_2}) \leftarrow$calculate MHD ▷ Equation 3.10
37: $d(e_{q_1}, e_{q_2}) \leftarrow$calculate distance between edges from $d(PI_{q_1}, PI_{q_2})$, $d(D_{q_1}, D_{q_2})$, $d(T_{q_1}, T_{q_2})$, $d(v_s(e_{q_1})$,$v_s(e_{q_2})$, $d(v_t(e_{q_1}), v_t(e_{q_2}))$ ▷ Equation 3.12
38: $edges[\,] \leftarrow$ insert $d(e_{q_1}, e_{q_2})$
39: $d(Q_1, Q_2) \leftarrow$calculate average for $vertices[\,], edges[\,]$ ▷ Equation 3.13 .
40: **return** $d(Q_1, Q_2)$



(a) Original query Q_1 **(b)** Modified query Q_2

Figure 3.5: Example query and its modification-based explanation

In a similar way, the distances for the remaining vertices are calculated: $d(v^1_{q_1}, v^1_{q_2}) = 0.16$, $d(v^3_{q_1}, v^3_{q_2}) = 0.33$, and $d(v^4_{q_1}, v^4_{q_2}) = 1$ (because vertex v^4 does not exist in Q_2).

Since edges $e^1_{q_1}$ and $e^1_{q_2}$ have the same type, source, and target nodes, their distances equal 0. Only the predicate interval for the attribute *sinceYear* has been changed:

$$d(pi_{sinceYear}(e^1_{q_1}), pi_{sinceYear}(e^1_{q_2})) = max\left(\frac{1+0}{2}, \frac{0}{1}\right) = \frac{1}{2} \tag{3.17}$$

The aggregated distance for edge e^1 is $d(e^1_{q_1}, e^1_{q_2}) = \frac{0+0+0.5+0+0}{5} = 0.1$. In the same way, the distances for the remaining edges are derived $d(e^2_{q_1}, e^2_{q_2}) = 0$ and $d(e^3_{q_1}, e^3_{q_2}) = 1$ (edge e^3 does not exist in Q_2).

The final distance between Q_1 and Q_2 is derived according to Equation 3.13 as

$$D(Q_1, Q_2) = \frac{0.16 + 0.33 + 0.33 + 1 + 0.1 + 0 + 1}{4 + 3} = 0.42 \tag{3.18}$$

In this section, we considered the syntactic distance between an original query and its explanation, which shows how different they look like for users and quantify this difference. 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. Therefore, in the following section, we describe a second comparison criterion, cardinality distance.

3.2.3 Cardinality Level

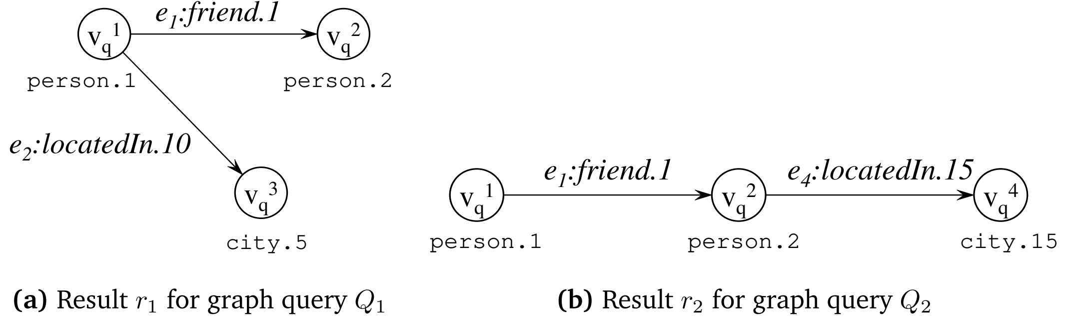
The main objective of this thesis is to explain to a user why a query delivered an unexpected result in terms of its cardinality and how to refine a query to deliver results of a required size. Therefore, the second level of comparing two explanations is to consider the sizes of their results. For too-many- and too-few-answers problems, this investigation can be easily done by calculating the deviation of the result size from the cardinality threshold.

Definition 5 (Cardinality Distance)**.** *Given a cardinality threshold C_{thr}, we define a cardinality distance $\Delta c(Q_1, Q_2)$ between two explanations Q_1 and Q_2 as an absolute difference between their result sizes C_1 and C_2 and cardinality threshold C_{thr} as follows:*

$$\Delta c(Q_1, Q_2) = ||C_{thr} - C(Q_1)| - |C_{thr} - C(Q_2)|| \tag{3.19}$$

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. A query with an empty result has an undefined cardinality distance. Therefore, we compare only the queries delivering non-empty results and calculate the cardinality distance between them as

$$\Delta c(Q_1, Q_2) = |C(Q_1) - C(Q_2)| \tag{3.20}$$



(a) Result r_1 for graph query Q_1 **(b)** Result r_2 for graph query Q_2

Figure 3.6: Example of two result subgraphs for calculating result distance

3.2.4 Result Level

In addition to syntactic and cardinality-based comparisons, the content of both result sets can be evaluated on removal and inclusion of new results during query refinement. This measure allows to compare how many new results are introduced or how many answers are removed from the original result. For a fair judgment, the results of an explanation and original query results are compared. Therefore, this measure can be defined only for problems, where an original query delivers at least some answers.

Definition 6 (Result Graph). *Given a result set R, a result graph $r \in R$ describes data subgraph G_d as a mapping between query vertices $V_q \in G_q$ and data vertices $V_d \in G_d$, query edges $E_q \in V_q$ and data edges $E_d \in G_d$ such that $v_d \in V_d$ and $e_d \in E_d$ are data vertex and edge unique identifiers.*

A result set R represents a collection of result graphs. To compare two results sets, a distance for each pair of their result graphs has to be derived.

Definition 7 (Distance between Two Result Graphs). *We define a distance between two result graphs $d(r_1, r_2)$ as a graph edit distance (GED) normalized to a total number of elements in both query graphs, which considers equally-weighted vertex and edge deletion, insertion, and relabeling:*

$$d(r_1, r_2) = \frac{GED(r_1 \rightarrow r_2)}{|E_{Q_1} \cup E_{Q_2}| + |V_{Q_1} \cup V_{Q_2}|}$$

This measure can compare graphs of different sizes. The distance calculation between two result graphs depends only on result sizes and therefore its complexity is $\mathcal{O}(k)$, where $k = |E_{Q_1} \cup E_{Q_2}| + |V_{Q_1} \cup V_{Q_2}|$. To calculate a distance between these two results r_1 and r_2, each data edge and vertex with the identical query identifiers are compared. If vertices or edges with the same query identifiers have different data identifiers, they have to be relabeled. Single relabeling has the cost of 1. If a vertex or an edge is missing, it has to be inserted in a result. A single insertion increments the total transformation cost.

Example Consider two results r_1 and r_2 for two explanations in Figure 3.6. Each vertex and edge has a query identifier in the form of v_q^i and e_q^j, respectively, and a data identifier from a data graph like for example $person.1$, etc. Both result graphs have the same data identifiers for the following common vertices and edge: v_q^1, e_q^1, and v_q^2, therefore, the distances between these vertices and edge equal 0. Vertex v_q^3 and edge e_q^2 exist only in r_1. Therefore, their deletion during the transformation $r_1 \rightarrow r_2$ increases the cost by 2. In other words, to transform r_1 into r_2, v_q^3 and e_q^2 have to be removed. In

Algorithm 2 Hungarian Assignment Algorithm

Input: matrix of result-result distances for two result sets D of $m \times n$ elements
Output: assignment costs $costs$
1: **if** $m > n$ **then**
2: Step 0. $D \leftarrow$ add $m - n$ columns D_{new}, where $d_{i,j} = 1$
3: Step 1. Subtract row minima from each row $\in D$
4: Step 2. Subtract column minima from each column $\in D$
5: Step 3. Count minimal number of columns and rows $k, \forall d_{i,j} = 0 \in k$
6: **if** $k < m$ **then**
7: Step 4. $s \leftarrow$ find minimal $(d_{i,j} \in D) \wedge (d_{i,j} \notin k)$
8: Step 4. Subtract $min(d_{i,j} \in k) \forall d_{i,j} \in k$
9: Step 4. Increase all $d_{i,j} \in k$ twice by $min(d_{i,j} \in k)$
10: Go to Step 3.
11: Step 5. Calculate assignment
12: Step 6. Calculate assignment costs $costs$
13: **return** $costs$

addition, e_q^4 and v_q^4 have to be inserted, which increases the cost by 2. To conclude, the graph edit distance is

$$d(r_1, r_2) = \frac{VD(v_{q1}^3) + ED(e_{q1}^2) + VI(v_{q2}^4) + EI(e_{q2}^4)}{|N_{r_1} \cup N_{r_2}| + |M_{r_1} \cup M_{r_2}|} = \frac{4}{7},$$

where VD is a vertex deletion, ED is an edge deletion, VI is a vertex insertion, EI is an edge insertion, N is a number of vertices, and M is a number of edges.

After calculating the distance for each pair of result graphs $(r_1^i, r_2^j), r_1^i \in R_1, r_2^j \in R_2$, a total distance between two result sets can be derived. We model the comparison of two result sets as the maximum generalized assignment problem [93], 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 costs, which are 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.

Definition 8 (Generalized Assignment Problem)**.** *Given two result sets R_1 with N graphs and R_2 with M graphs, assign each result graph $r_i \in R_1$ to exactly one result graph $r_j \in R_2$ with distance $d(r_i, r_j)$ so as to minimize the total distance of the assignment, i.e.*

$$\textit{mimimize } z = \sum_{i=1}^{N} \sum_{j=1}^{M} d(r_i, r_j) * r_{i,j}, \tag{3.21}$$

$$\textit{subject to } \sum_{j=1}^{M} r_{i,j} = 1, i \in R_1, \sum_{i=1}^{N} r_{i,j} = 1, j \in R_2, \tag{3.22}$$

$$\textit{where } r_{i,j} = \begin{cases} 1 \textit{ if result item } r_i \textit{ is assigned to result item } r_j \\ 0, \textit{ otherwise} \end{cases} \tag{3.23}$$

The assignment of result graphs can be modeled by the Hungarian-based algorithm [88], which is sketched in Algorithm 2. As an input, the algorithm requires a

matrix D of size $m * n$ with m tasks and n workers. To guarantee that each worker executes only a single task, $n \geq m$. For $m > n$, additional $m - n$ columns with $d(r_i, r_j) = 1$ have to be inserted. Each matrix element $d_{i,j} \in D$ represents the costs of executing a task t_i by a worker w_j. In our setup, these costs are a result distance $d(r_i, r_j)$ between two compared results. The algorithm produces an assignment, which describes the mapping between result graphs of an original query and answers to an explanation, which has the minimal total $costs$. To be able to compare different explanations, we normalize $costs$ to the number of answers to an original query R_1 (this step is not illustrated in Algorithm 2).

Initial Matrix

$$\begin{pmatrix} 0.15 & 0.21 & 0.18 & 0.16 \\ 0.10 & 0.17 & 0.60 & 0.48 \\ 0.12 & 0.29 & 0.10 & 0.15 \\ 0.23 & 0.44 & 0.13 & 0.25 \end{pmatrix}$$

Step 1

$$\begin{pmatrix} 0.00 & 0.06 & 0.03 & 0.01 \\ 0.00 & 0.07 & 0.50 & 0.38 \\ 0.02 & 0.19 & 0.00 & 0.05 \\ 0.10 & 0.31 & 0.00 & 0.12 \end{pmatrix}$$

Step 2

$$\begin{pmatrix} 0.00 & 0.00 & 0.03 & 0.00 \\ 0.00 & 0.01 & 0.50 & 0.37 \\ 0.02 & 0.13 & 0.00 & 0.04 \\ 0.10 & 0.25 & 0.00 & 0.11 \end{pmatrix}$$

Step 3

$$\begin{pmatrix} 0.00 & 0.00 & 0.03 & 0.00 \\ 0.00 & 0.01 & 0.50 & 0.37 \\ 0.02 & 0.13 & 0.00 & 0.04 \\ 0.10 & 0.25 & 0.00 & 0.11 \end{pmatrix}$$

Step 4

$$\begin{pmatrix} 0.00 & 0.00 & 0.05 & 0.00 \\ 0.00 & 0.01 & 0.52 & 0.37 \\ 0.00 & 0.11 & 0.00 & 0.02 \\ 0.08 & 0.23 & 0.00 & 0.09 \end{pmatrix}$$

Final Assignment

$$\begin{pmatrix} 0.01 & 0.00 & 0.06 & 0.00 \\ 0.00 & 0.00 & 0.52 & 0.36 \\ 0.00 & 0.10 & 0.00 & 0.01 \\ 0.08 & 0.22 & 0.00 & 0.08 \end{pmatrix}$$

Example We explain the algorithm based on the following matrix of distances D. The number of rows correspond to the number of result graphs $|R_1|$ of the original query. The number of columns equals to the number of result graphs $|R_2|$ of the explanation. Each element $d_{i,j}$ of this matrix describes the distance between result graphs $r_i \in R_1$ and $r_j \in R_2$. At Step 1 and Step 2, the minimal row values and column numbers are subtracted from the corresponding rows and columns. The derived zeros show the minimal assignments between R_1 and R_2, i.e., the most similar result graphs between two sets. At Step 3, we check zero coverage for the matrix generated at Step 2, which shows whether we uniquely assigned all result graphs $\in R_1$ to graphs $\in R_2$. Only three lines are covered, namely: row 1, row 2, and column 3. This is not enough to cover the matrix D with four rows, therefore, additional zeros (minimal assignments) have to be introduced. The smallest uncovered number (not highlighted numbers) is 0.02. It is substituted from all uncovered numbers and added to those elements, which are covered twice at Step 4. Zeros in this matrix are covered only by three lines, therefore not all result graphs can be assigned and additional zeros have to be introduced again. The minimal uncovered number is 0.01. It is substituted from all uncovered numbers and added to those elements, which are covered twice. In the final matrix, all zeros are covered and the optimal assignment can be derived. Elements $d_{i,j}$ with zero values describe the mapping between R_1 and R_2 with the minimal total assignment costs. The derived optimal assignment includes the following distances: $d_{3,1}, d_{2,2}, d_{4,3}$, and $d_{1,4}$. The corresponding minimal costs are $costs = 0.58$ derived at Step 6. The result distance between two result sets is $d = \frac{costs}{|R_1|} = \frac{0.58}{4} = 0.145$.

3.2.5 Evaluation

In the previous sections, we discussed three similarity metrics for comparing explanations: syntactic, cardinality, and result distances, which consider their syntactic descrip-

tions, sizes, and contents of their result sets. In this evaluation, we will characterize their behavior and reveal dependencies between them.

Evaluation Setup

In order to discover dependencies between the proposed similarity metrics, we should compare such explanations and original queries, which deliver some results. Otherwise, a result distance cannot be calculated. Therefore, in this evaluation we compare randomly generated modification-based explanations for the too-few- and too-many-answers problems and multiple cardinality thresholds.

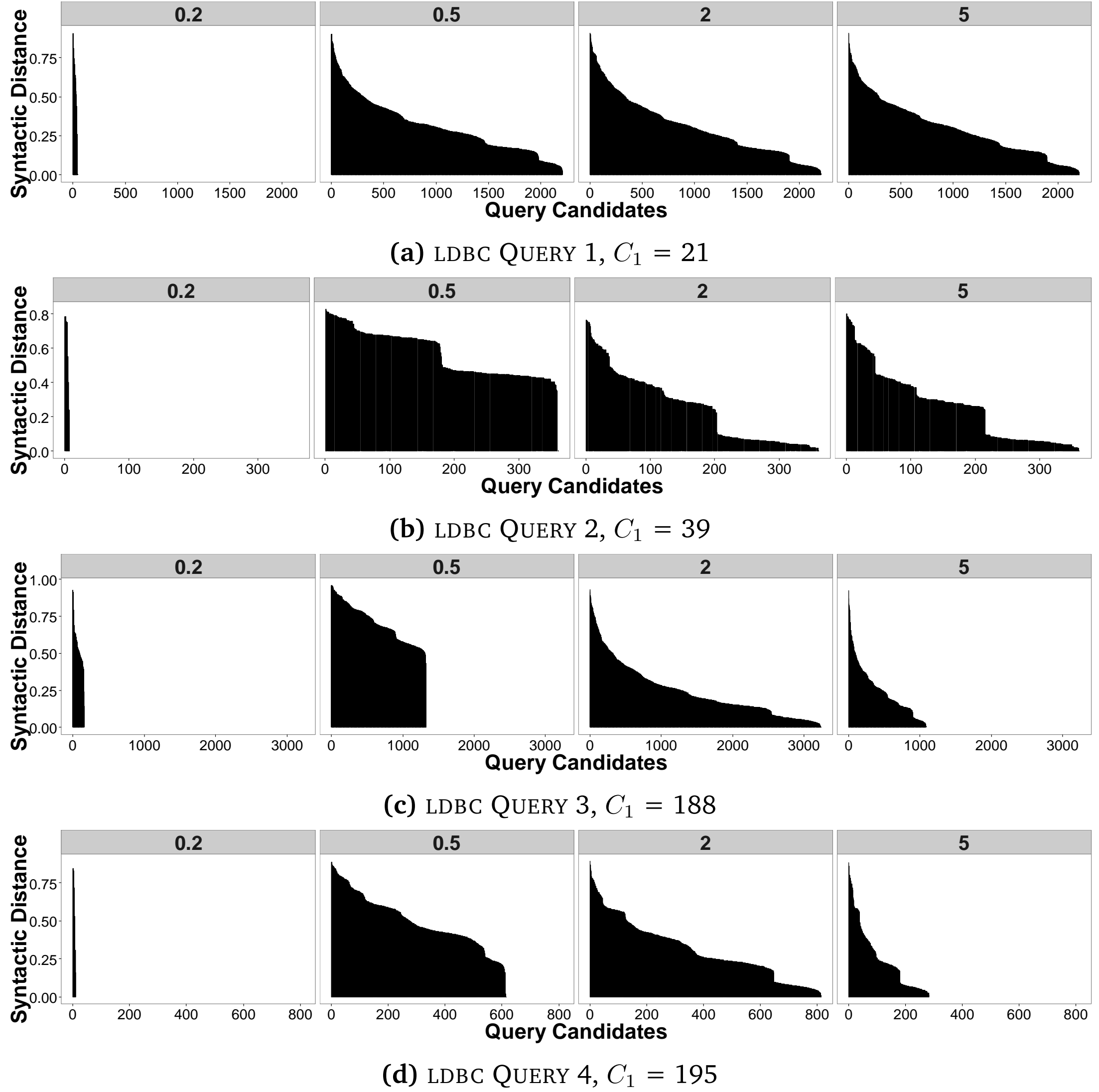
As original queries, we refine four LDBC queries on the LDBC SF1 data set, which are described in detail in Appendix A.1 and include LDBC QUERY 1 – 4 with original cardinalities defined as C_1 in Table A.1. LDBC is the Linked Data Benchmark Council, a non-profit organization dedicated to establishing benchmarks, benchmark practices, and benchmark results for graph data management software. For each query, we consider four cardinality factors $C = \{0.2, 0.5, 2, 5\}$, which are used to derive the cardinality thresholds in respect to the original cardinalities. For example, cardinality factor $C = 0.2$ means that the cardinality threshold equals to the 20% of the original result size. Cardinality factor $C = 2$ means that the cardinality threshold is twice so big as the original cardinality. The use of cardinality factors allows us to compare similarity metrics among different queries. The cardinality factors less than 1 model the too-many-answers problem. Otherwise, the too-few-answers problem is evaluated. In all charts, evaluation results are presented according to the used cardinality factors, which annotate subfigures on the top.

The generation of query variants is implemented as an iterative procedure. First, we execute an original query and store its result for further comparisons. Then, we randomly choose modification operators and query elements from an original query to be modified. Based on them, we generate multiple modified queries as explanations. At each iteration, we pick up one query from this set, execute it, and compare it against the original query, original result set, and cardinality threshold. We terminate the process, after no candidates exist in the pool or 5% of candidates for a three-level modification are processed. For all candidates, we calculate all three similarity measures.

This evaluation is executed on a single server machine equipped with SUSE Linux Enterprise Server 11 (64 bit) with an Intel Xeon Processor E5-2643 (24 CPUs and 96 GB RAM). This setup is used in all experiments described in this thesis and fully explained in Appendix A. In the following sections, the evaluation results are presented according to the distance measure used.

Syntactic Distance

First, we consider the syntactic distance of generated explanations, which describes the y axis in Figure 3.7. The generated explanations are ordered according to their syntactic distances in a descending order and correspond to the x axis. From this evaluation, we can conclude that the proposed syntactic distance is a monotonic function, which increases if a new change was introduced in an explanation. A few explanations are characterized by the same syntactic distance, which means that these queries have the same amount of changes. For example, two explanations have been generated by extending the same predicate intervals. The behavior of the syntactic distance is similar among different queries. Larger magnitudes correspond to strong query changes, which are typically caused by topological changes.



(a) LDBC QUERY 1, $C_1 = 21$

(b) LDBC QUERY 2, $C_1 = 39$

(c) LDBC QUERY 3, $C_1 = 188$

(d) LDBC QUERY 4, $C_1 = 195$

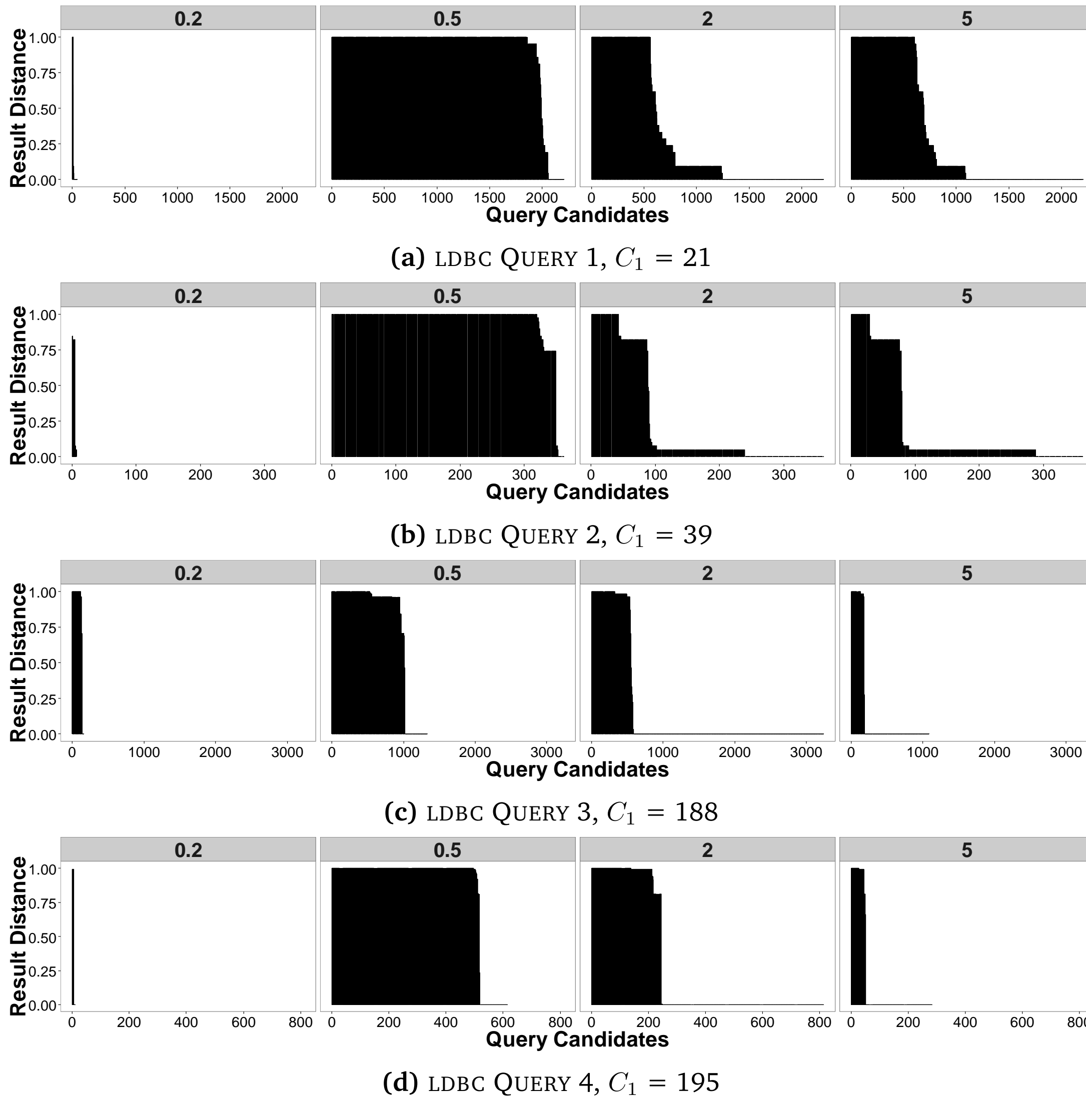
Figure 3.7: Ordered syntactic distances for randomly generated explanations

Result Distance

Second, we consider the result distances for generated explanations that show how the data subgraphs delivered by an explanation differ from the data subgraphs of an original query. This metric equals 1 if a result is completely different from an original result or the rewritten query produces an empty answer. The evaluation charts for the result distributions are constructed similarly to the syntactic-distance charts and presented in Figure 3.8.

The evaluation shows that the result distance distribution is graduated. The candidates exhibiting the same result distance can have different syntactic distances and probably similar cardinality distances. This behavior is dependent on an original query and shows how strong query elements are interconnected with each other. Stronger connections require simultaneous modifications of multiple query elements in order to change a result set. From explanations delivering the same result distance, those have to be preferred, which are less syntactically different from the original query.

Considering charts for the too-many-answers problem ($C < 1$) in Figure 3.8, in most cases the result distance approaches 1, because only a few original results are



(a) LDBC QUERY 1, $C_1 = 21$

(b) LDBC QUERY 2, $C_1 = 39$

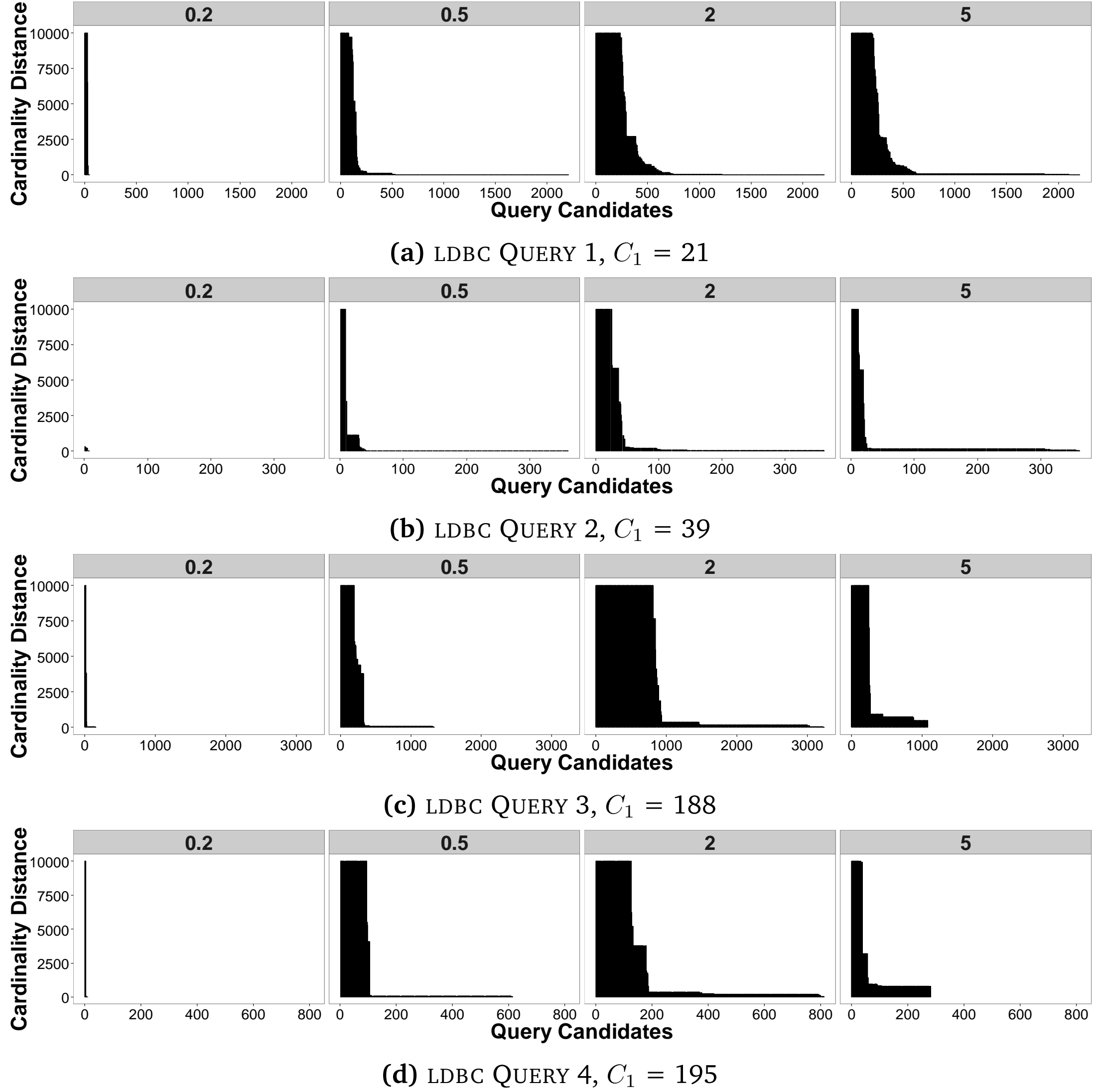
(c) LDBC QUERY 3, $C_1 = 188$

(d) LDBC QUERY 4, $C_1 = 195$

Figure 3.8: Ordered result distances for randomly generated explanations

included in the result set of an explanation. In addition, each answer can contain less information than before the modification.

In contrast, for the too-few-answers problem, we extend predicates with new values, which introduce additional answers without removing original ones and result in a low result distance. However, we also allow to change the topology by removing vertices and edges, which increases the result distance. In addition, those queries, which deliver no results, lead to the maximal result distance = 1. This effect can happen if the transformed query was created by several relaxations and concretizations with different values for predicates. Considering Figures 3.8c – 3.8d, LDBC QUERY 3 and LDBC QUERY 4 are difficult to modify, because in most cases they have result distances with the maximum value, where some of them are induced by empty result sets. The most easy-to-judge result distances can be calculated in scenarios, where the cardinality of an original query is close to a cardinality threshold.



(a) LDBC QUERY 1, $C_1 = 21$

(b) LDBC QUERY 2, $C_1 = 39$

(c) LDBC QUERY 3, $C_1 = 188$

(d) LDBC QUERY 4, $C_1 = 195$

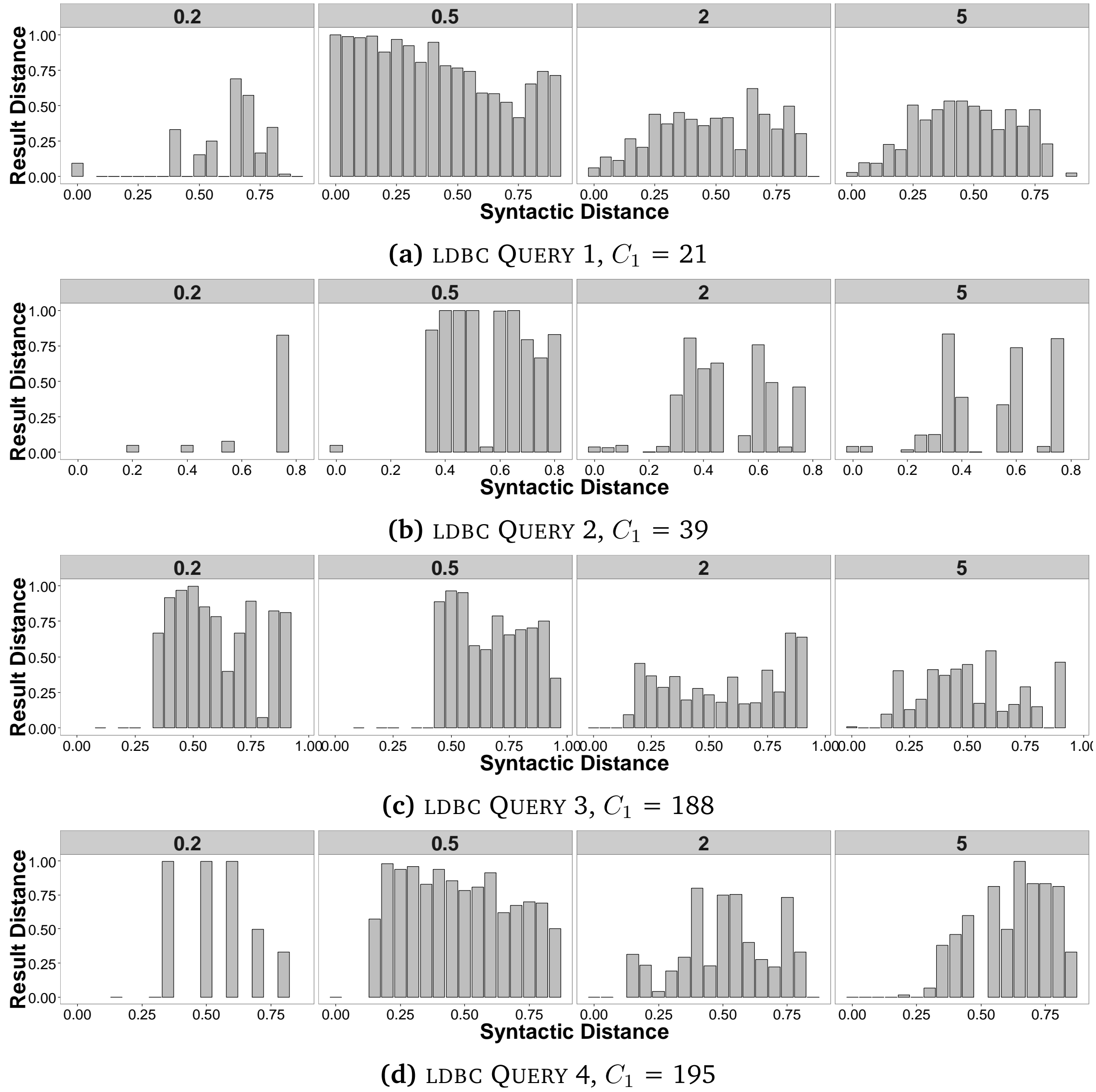
Figure 3.9: Ordered cardinality distances for randomly generated explanations

Cardinality Distance

Third, we evaluate cardinality distances for generated explanations, which are illustrated in Figure 3.9. In this evaluation, we express a cardinality distance as the difference between a cardinality threshold and a cardinality of a generated explanation. As in the previous figures, generated explanations are sorted by their cardinality distances in a descending order. Similar to the result-distance charts, multiple explanations can have the same cardinality distance, which is explained by existing query dependencies and necessity to simultaneously execute multiple changes in order to modify the result cardinality. If not all required dependent elements are modified in the query, an executed modification will not result in a cardinality change. This behavior can be observed for explanations with the same cardinality distances in Figure 3.9.

Average Result Distance vs. Syntactic-Interval Distance

Until now, we have considered all three similarity measures independently. In this set of experiments, we would like to test whether some dependencies exist between similarity metrics.



(a) LDBC QUERY 1, $C_1 = 21$

(b) LDBC QUERY 2, $C_1 = 39$

(c) LDBC QUERY 3, $C_1 = 188$

(d) LDBC QUERY 4, $C_1 = 195$

Figure 3.10: Distribution of average result distances for randomly generated explanations

The most interesting dependency can be revealed between result and syntactic distances, because the result distance considers the size of an explanation. In this experiment, we group generated query candidates according to their syntactic distances in several groups with step 0.05 from interval $[0.0, 1.0]$. For each group, we calculate the average result distance, which is illustrated in Figure 3.10.

In all charts in Figure 3.10, we can see the trend that with an increasing syntactic distance the result distance increases as well. Small syntactic distances usually consider only predicate changes, which lead to smaller cardinality improvements, and as a consequence, smaller result distances than in scenarios with topological changes.

If we compare the results between too-many- and too-few-answers problems which are described by cardinality factors $C < 1$ and $C > 1$, respectively, we can observe that the higher result distances are derived for the too-many-answers problems, because in these scenarios we need to reduce the amount of data subgraphs and therefore some information is removed from the result, which leads to higher result distances.

Evaluation Summary

In this section, we evaluate three similarity measures to compare explanations including syntactic, result, and cardinality differences. From the experimental results, we can conclude that the syntactic distance represents a monotonic feature, which increases by introducing new modifications. Larger syntactic distances are derived if the topology of a query is changed. The evaluation of result distances shows that for the too-many-answers problem the result distance is higher than for the too-few-answers problem, because the modification process aims at reducing the number of answers and therefore not all of them are included in the result set of an explanation. Cardinality and result distances exhibit dependencies between multiple changes via graduated distributions, which show that multiple changes have to be applied to an original query in order to deliver a different cardinality. We also observe a dependency between result and syntactic distances. In most cases, an increasing syntactic distance leads to a higher result distance and can even approach to one.

3.3 Summary

In this chapter, 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, which consider three important aspects, 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 result distance slightly depends on the syntactic one such that stronger differences in a query description lead to larger result distances. The result distance is also influenced by the number of results of an 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 chapters, 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. In Chapter 4, we start this description with proposing a way to generate subgraph-based explanations, which reveals why a query delivered an unexpected result.

4

Explaining Unexpected Results

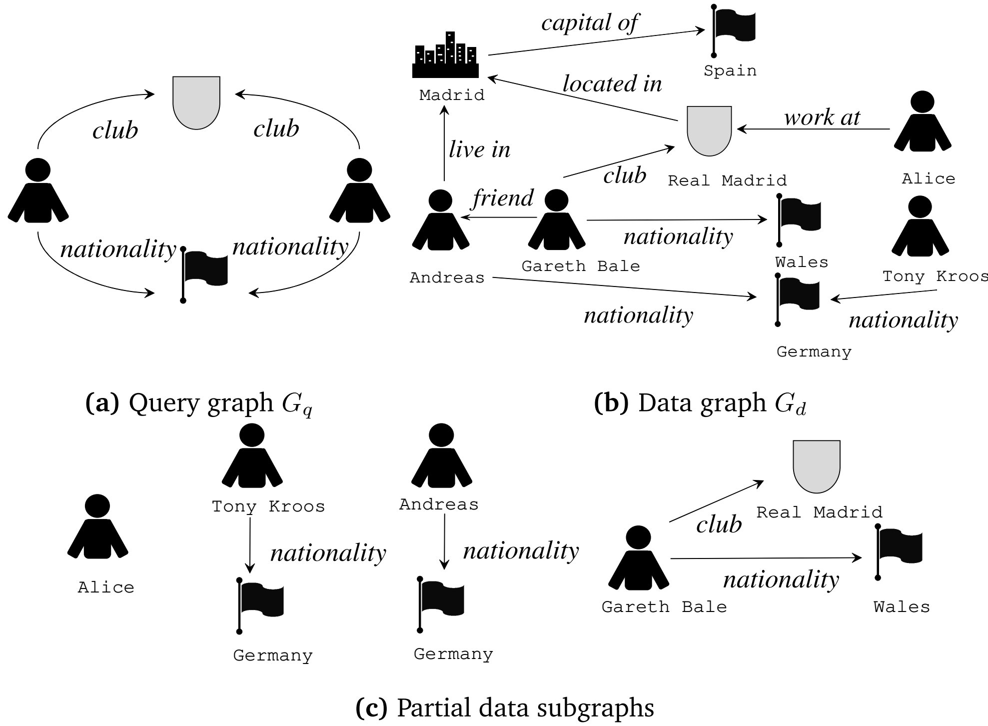
In the previous chapter, we introduced the definition of the property-graph model and discussed a set of general debugging features for why-queries, which was derived from the state-of-the-art systems in Chapter 2 and includes holistic support of different cardinality-based problems, explanation of unexpected results and query reformulation, comprehensive comparison of explanations, and non-intrusive user integration in the generation of explanations. In addition, we had a deep look at one of the properties, comprehensive comparison of explanations, which allows to qualify generated explanations. For this purpose, we proposed and evaluated three similarity measures including syntactic, cardinality, and result distances.

One of the important properties from this set refers to the core debugging functionality, explanation of unexpected results and query reformulation. In this chapter, 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 discuss several optimization techniques, which allow to increase quality of explanations and their performance. We also consider a model for integrating user preferences in generation of explanations, which allows to derive preference-aware explanations. Parts of the subgraph-based approach presented in this chapter have been published in [138, 139, 140, 141, 142].

Before introducing our concept of discovering a query subgraph, which is responsible for an unexpected result, some basic definitions are provided in Section 4.1. Afterwards, two solutions are introduced for empty-answer, too-few-, and too-many-answers problems in Section 4.2, whose optimizations are provided in Section 4.3. In Section 4.4, a strategy to consider user interest during generation of subgraph-based explanations is discussed. Finally, the proposed solutions are evaluated in Section 4.5.

4.1 Preliminaries

As presented in Chapter 3, a query to a graph database can be represented by a graph itself, which is annotated with the attribute predicates on vertices and edges. Therefore, if a query delivers an unexpected result, a reason of unexpectedness can be expressed



(a) Query graph G_q **(b)** Data graph G_d

(c) Partial data subgraphs

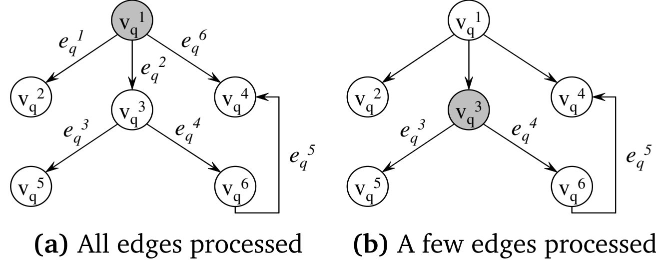
Figure 4.1: Example of why-empty query, data graph, and corresponding partial answers

as a part of the query graph, which violates a given cardinality constraint. In the state-of-the-art systems (see Section 2.3), this part of the query is called a minimal failed query. The remaining part of a query, which satisfies the cardinality constraint, can be described in terms of a maximum common subgraph and therefore, it can be discovered by existing state-of-the-art maximum common subgraph algorithms. Before introducing these algorithms, we illustrate the principle of generating subgraph-based explanations on the example in Figure 4.1.

Example Assuming the example database in Figure 4.1b, a user is interested in data subgraphs, which consist of four vertices and four edges of types *club* and *nationality* like presented in Figure 4.1a. The discovery of this pattern in the data graph in Figure 4.1b delivers an empty result, because there is no data subgraph, which would match the entire queried graph. However, there are multiple partial results in the data graph, which vary from the subgraph containing a single vertex like *Alice* to a subgraph consisting of up to three vertices (some partial results are illustrated in Figure 4.1c). The number of such partial answers can be very large and therefore to explain why each of them does not become an answer can be a non-feasible task. Based on this observation, an explanation should be generated only for the largest discovered subgraph like the right one in Figure 4.1c. This answer holds the maximum available information in the data graph matching the query. The same holds for debugging of too many and too few results, but instead of an empty result we have to consider the number of discovered data subgraphs.

To understand, which part of query G_q, which satisfies a cardinality constraint, can be found in data graph G_d and which part violates it, the maximum common connected subgraphs between data and query graphs must be found and their differential graphs must be calculated.

Definition 9 (Maximum Common Connected Subgraph (MCS))**.** *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) All edges processed **(b)** A few edges processed

Figure 4.2: Depth-first search

*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 $V' \subsetneq V''$ or $E' \subsetneq E''$.*

For too-many- and too-few-answers problems, a maximum common connected subgraph is represented by a *cardinality-bounded* MCS, which shows that part of a query graph, whose number of data instances does not violate a cardinality threshold. It is defined as follows.

Definition 10 (Cardinality-Bounded Maximum Common Connected Subgraph (BMCS))**.** *Let G_d be a data graph, G_q be a query graph, and C_{thr} be a cardinality threshold of a result set. A **cardinality-bounded maximum common connected subgraph** G_d^{bnd} for G_d, G_q, and C_{thr} is a common connected subgraph of G_d and G_q such that $\forall$ common connected subgraph G''_d for G_d, G_q such as G''_d is isomorphic to G_d^{bnd} and a total number of such data subgraphs $\sum G''_d \leq C_{thr}$ for the too-many-answers problem or $\sum G''_d \geq C_{thr}$ for the too-few-answers problem.*

Cardinality-bounded maximum common connected subgraphs (BMCSs) show discovered partial results, before the resulting cardinality has exceeded or dropped below the cardinality threshold.

4.1.1 Detection of Maximum Common Connected Subgraphs

The MCSs between a graph query and a data graph can be discovered by maximum common connected subgraph algorithms [94, 135]. 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 [42]. For example, a matrix A 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. The MCS 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.

To discover MCSs, Ullmann's [135] and McGregor's [94] 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.

Assume that the depth-first procedure starts from vertex v_q^1 in the example shown in Figure 4.2a, and explores all edges of the query as follows: $e_q^1, e_q^2, e_q^3, e_q^4, e_q^5, e_q^6$. If it would start from vertex v_q^3 like in Figure 4.2b only edges e_q^3, e_q^4, e_q^5 would be traversed. To ensure the discovery of all MCSs, the depth-first search is conducted for each vertex of a query, and the data graph is treated as undirected. Ullmann's [135]

and McGregor's [94] 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 the property-graph model [122]. To be used for property graphs, these algorithms have to be adapted to work with properties on edges and vertices. Additionally, only those edges and vertices have to be considered whose descriptions match the predicates and types given in a query graph. The discovery of an MCS can also be modeled as search of a maximum clique like in the Durand-Pasari [46] and in the Balas Yu [9] algorithms, which are also tree-search algorithms. All these methods show diverse performance results among different graphs.

Most of the above presented algorithms implement the depth-first search with sophisticated heuristics for discarding some search branches, which are suitable for labeled graphs. This thesis considers the property-graph model. For this model, a most promising optimization technique differs from the used heuristics in the state-of-the-art methods and implements filtering of data subgraphs based on query predicates. Therefore, before executing a depth-first search, the data graph is filtered such that only those vertices and edges are considered by the search, which match the query predicates.

4.1.2 Calculation of Differential Graphs

The MCS algorithms introduced above can be applied to detect those query parts in a data graph, which satisfy a cardinality constraint. To determine which structural part fails, differential graphs have to be derived, which are calculated as the differences between the discovered MCSs (BMCS) and the query graph.

A differential graph includes those query vertices and edges, which were not visited during the discovery of MCSs (BMCSs), and the instances of query vertices adjacent to an MCS.

Definition 11 (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 the maximum common connected graph or cardinality-bounded maximum common connected subgraph 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 $\mathcal{O}(k)$, where $k = m + n + l + s$, m is the number of edges, n is the number of vertices, l is the number of attributes, and s is the number of types in a query.

4.2 Generation of Subgraph-Based Explanations

In the previous section, we gave general definitions of MCS and BMCS and shortly discussed the state-of-the-art methods for their discovery. The presented algorithms mostly implement a depth-first search with optimizations, which are specific for labeled graphs. In this section, we propose two algorithm for the discovery of MCSs and BMCSs for property graphs and reveal several model- and algorithm-specific optimizations.

In general, the generation of the subgraph-based explanations consists of two steps: (1) The detection of maximum common connected subgraphs (MCSs – BMCSs), which correspond to the cardinality constraint, by a maximum common subgraph algorithm. (2) The calculation of differential graphs from MCSs (BMCSs) and the query graph, which show why discovered data subgraphs fail to deliver a complete answer.

Algorithm 3 DISCOVERMCS: Traversal-based discovery of MCSS

Input: query graph $G_q = (V_q, E_q)$, data graph $G_d = (V_d, E_d)$
Output: set mcs of discovered data MCSS
1: $\forall v_q^i \in V_q : v_d^i \leftarrow$ extract data vertices from data graph G_d
2: $\forall e_q^i \in E_q : e_d^i \leftarrow$ extract data edges from data graph G_d
3: $\forall e \in e_d$: filter out edges with non-matching vertices from e_d and v_d
4: **for all** $startV_q \in V_q$ **do**
5: $startV_d \leftarrow$ extract start data vertices for $startV_q$
6: $subgraphs \leftarrow$ initialize data subgraphs with $(startV_q, startV_d)$
7: **while** $subgraphs \neq \varnothing$ **do**
8: $\forall graph \in subgraphs$: extend data subgraph $graph$ with edge from e_d
9: $mcs \leftarrow$ insert completed MCS from $subgraphs$
10: **return** mcs

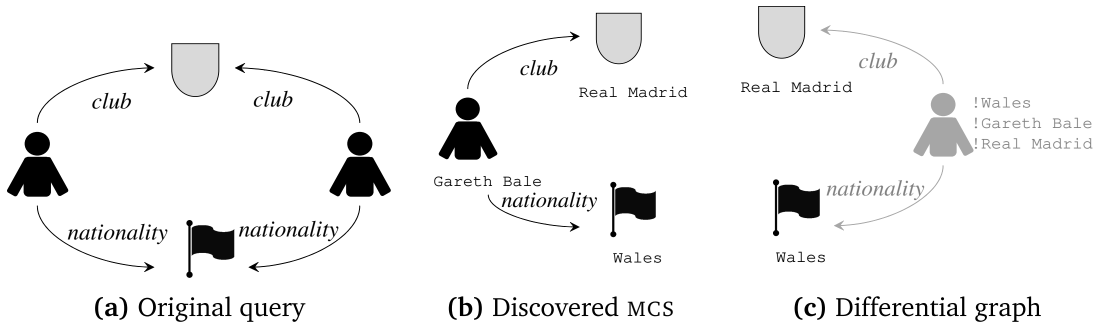
This process slightly differs between why-empty, why-so-few, and why-so-many queries. While why-empty queries have to search for a missing query subgraph from the data graph, why-so-few and why-so-many queries have to consider the number of discovered data subgraphs by preventing the violation of a cardinality threshold. Therefore, as follows, two algorithms are proposed, which discover MCSS for the empty-answer problem and BMCSS for too-few- and too-many-answers problems in Section 4.2.1 and 4.2.2, correspondingly.

4.2.1 The DISCOVERMCS Algorithm for Why-Empty Queries

To discover the reason of an 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 conduct the depth-first search along the query like presented in Algorithm 3.

To leverage an MCS algorithm for property graphs, only those data edges and vertices have to be considered, which match the query predicates. For this purpose, such vertices and edges are indexed at lines 1 and 3, correspondingly. To ensure that the algorithm finds all MCSS, it should be launched from all query vertices as multiple starting points and traverse all possible edge sequences. In Algorithm 3, the first starting vertex is chosen at line 4 and the initial set of solutions is produced at line 6, which consists only of pairs *(start query vertex, data vertex)*. All these initial data subgraphs are stored in vector $subgraphs$ that maintains all incomplete solutions. The core of Algorithm 3 consists of two function calls at lines 7 – 9. First, for each incomplete subgraph, a next query edge is derived by the depth-first search algorithm, which is then used to extend the subgraph with new data edges at line 8 extracted from the indexed data. New subgraphs created at this step substitute their predecessors in this vector. Second, those discovered MCSS are marked as complete and stored in the result vector, which cannot be further extended at line 9. The procedure continues until all subgraphs are moved to result vector mcs. After all data $subgraphs$ are marked as complete and the search is terminated, 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 4.3a. A possible answer to this why-empty query would consist of MCS G'_d as shown in Figure 4.3b, and differential graph G'_q as in Figure 4.3c, 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*



(a) Original query **(b)** Discovered MCS **(c)** Differential graph

Figure 4.3: Original query delivering empty result and its subgraph-based explanation: which two vertices are from same country and play in same club?

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).

The proposed algorithm searches for all MCSs in the data graph and therefore is executed multiple times for each start vertex. As a consequence, the same MCSs may be discovered multiple times and the result may include duplicated MCSs, which also increases the search time. Therefore, we optimize this algorithm with several heuristics as described in Section 4.3.

To conclude, this 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 it for why-so-few and why-so-many queries, which have to consider the number of output data subgraphs isomorphic to the same query subgraph. Therefore, in the following section we present a join-based algorithm, which overcomes these drawbacks and delivers BMCS with respect to the given cardinality threshold.

4.2.2 The BoundedMCS Algorithm for Why-So-Few and Why-So-Many Queries

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 (see Definition 10) 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.

As we have already mentioned above, DiscoverMCS presented in Algorithm 3 can potentially be used to discover BMCSs and to generate subgraph-based explanations for why-so-few and why-so-many queries. However, it does not consider a cardinality threshold and does not track the size of a result set. To account for them, all discovered data subgraphs have to be extended at the same time with the same query edge. To generate subgraph-based explanations for the too-few- and too-many-answers problems, we propose the BoundedMCS algorithm described in Algorithm 4 that considers these observations and discovers BMCSs.

To detect which part of a query matches the cardinality threshold C_{thr}, such BMCSs have to be found in data graph G_d for query graph G_q, which total number is lower or higher than cardinality threshold C_{thr} for the too-many- or too-few-answers problem. Similar to DiscoverMCS, data vertices and edges are indexed at lines 1 and 2. To

Algorithm 4 BOUNDEDMCS: Join-based discovery of BMCSs

Input: query graph $G_q = (V_q, E_q)$, data graph $G_d = (V_d, E_d)$, cardinality threshold C_{thr}

Output: set $bmcs$ of discovered BMCSs

1: $\forall v_q^i \in V_q : v_d^i \leftarrow$ extract data vertices from data graph G_d
2: $\forall e_q^i \in E_q : e_d^i \leftarrow$ extract data edges from data graph G_d
3: $\forall e \in e_d$: filter out edges with non-matching vertices from e_d and v_d
4: **for all** $e_q^i \in E_q$ **do**
5:     **if** e_q^i satisfy C_{thr} **then**
6:         $subgraphs \leftarrow$ initialize data subgraphs with (e_q^i, e_d^i)
7:         $accepted \leftarrow$ insert e_q^i
8:         extendSubgraphs$(G_q, e_d, rejected, accepted, subgraphs, C_{thr})$
9:         **if** syntactic distance$(subgraphs) \leq$ syntactic distance$(bmcs)$ **then**
10:             $bmcs \leftarrow subgraphs$
11: **return** $bmcs$

Algorithm 5 EXTENDSUBGRAPHS: Extension of data subgraphs with new edge

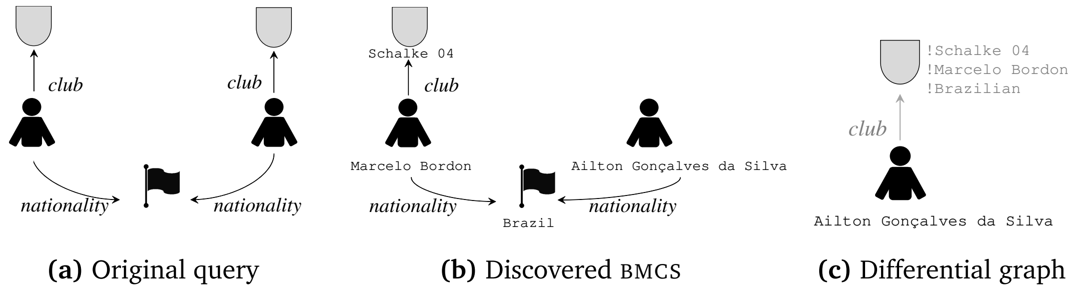
Input: query graph $G_q = (V_q, E_q)$, extracted data edges e_d, rejected query edges $rejected$, accepted data edges $accepted$, discovered data subgraphs $subgraphs$, cardinality threshold C_{thr}

Output: set $subgraphs$ of discovered data subgraphs

1: **while** $e \leftarrow$ get new edge for $(G_q, accepted, rejected)$ **do**
2:     $\forall graph \in subgraphs$: extend with e from e_d
3:     **if** $subgraphs$ satisfy C_{thr} **then**
4:         $accepted \leftarrow$ insert e
5:         $subgraphs =$ EXTENDSUBGRAPHS $(G_q, e_d, rejected, accepted, subgraphs, C_{thr})$
6:     **else**
7:         $\forall graph \in subgraphs$: remove e
8:         $rejected \leftarrow$ insert e
9: **return** $subgraphs$

ensure that the algorithm finds BMCSs, it should be launched from all query edges and traverse all possible edge sequences at lines 4 – 10. The core of the algorithm EXTENDSUBGRAPHS described in Algorithm 5 extends each discovered data subgraph with a new edge at lines 1 – 2. If the size of extended set $subgraphs$ satisfies the cardinality constraint, then the edge is accepted and the traversal process continues at lines 3 – 5. Otherwise, we can assume that the subgraph cardinality changes monotonically and any further traversal step will not make the subgraph satisfy the cardinality threshold. Therefore, the edge can be rejected and the last extension of data subgraphs can be undone at lines 6 – 8. After conducting the search for each start vertex in Algorithm 4, the largest BMCSs are stored and delivered to a user at lines 9 – 11.

Example Assume a modified version of our running example in Figure 4.4a, where two soccer players from the same country have to be discovered who play in different clubs. We want to get not more than ten answers ($C_{thr} = 10$). However, too many answers have been returned: number of discovered data subgraphs $K = 84$ exceeds cardinality threshold $C_{thr} = 10$. By processing the query graph, the number of answers grows, when we search for *club*, because the same player can play at different times during his career in several clubs. While *Marcelo Bordon* played in four clubs, *Aílton Gonçalves da Silva* played in 21 soccer teams. To explain, why the query delivers too



(a) Original query **(b)** Discovered BMCS **(c)** Differential graph

Figure 4.4: Why-so-many query and its answer: which two players of same nationality play in different clubs?

many results, a why-so-many query can be executed, which discovers maximum existing data subgraphs, which correspond to the cardinality threshold. A possible answer to this why-so-many query would consist of BMCS G_d^{bnd} as shown in Figure 4.4b, and differential graph G_q^{ubnd} as in Figure 4.4c, which is an unbounded part of the query with constraints. The first part includes, for example, all discovered instances of vertices and edges like *Aílton Gonçalves da Silva*, *Schalke 04*, *Brazil*, and *Marcelo Bordon*. The second part consists of discovered adjacent vertex *Aílton Gonçalves da Silva*, unbounded query vertex and edges (gray), and constraints for vertices (gray).

Similar to DISCOVERMCS, the above presented BOUNDEDMCS algorithm considers all query vertices as start vertices and therefore can produce duplicated results, whose processing increases the response time. Therefore, several optimization techniques are provided in Section 4.3, which reduce the processing overhead.

After MCSs or BMCSs have been discovered we know which query part exists or satisfies the cardinality threshold, correspondingly. This is the first part of the query answer which is delivered to a user. Based on these subgraphs, we calculate, which part of a query is missing or violates a cardinality constraint called a differential graph as presented below.

4.2.3 Calculation of Differential Subgraphs

To compute a failed subquery that is a missing part of a query for the empty-answer problem or violates a cardinality constraint for the too-many- and too-few-answers problems, a query graph and a discovered MCS (BMCS) are required. The computation consists of two steps: (1) the split of discovered and undiscovered vertices and edges and (2) the annotation of an undiscovered query part with attribute or vertex conditions.

In the first step, the mapping between data edges and query edges as well as data vertices and query vertices is stored in temporary tables during query processing. Query vertices and edges, which are not represented in this mapping, comprise the differential graph. In the second step, this differential graph is annotated with several constraints as follows. (1) If a query edge is not discovered, but at least one of its end vertices has already been found, then this discovered end vertex has to be included into the differential graph. In the example presented above, vertex *Aílton Gonçalves da Silva* represents such a constraint (see Figure 4.4c) and is included in a differential graph. (2) If a query vertex is discovered in the data graph, then its instance has to be excluded from the non-discovered query vertices, which the differential graph consists of. In the example in Figure 4.3c, we exclude vertices *Schalke 04*, *Brazil*, *Marcelo Bordon*, and *Aílton Gonçalves da Silva* from a differential graph.

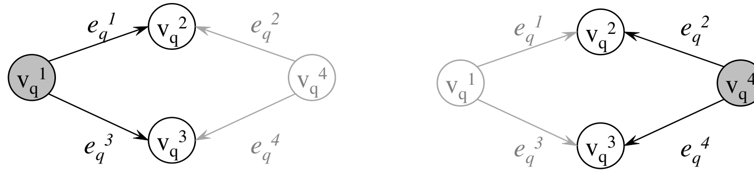


Figure 4.5: Weakly connected graphs

4.3 Optimization

The DISCOVERMCS and BOUNDEDMCS algorithms exhibit several common features: They are based on a depth-first search and require multiple runs from different starting vertices (edges) in order to guarantee the delivery of the best results. Therefore, the same optimization techniques can be applied to both algorithms.

We identified several drawbacks, whose solutions can increase the quality and the performance of the algorithms, i.e., to find larger subgraphs and to reduce the number of runs. First, both algorithms work only with connected graphs, therefore, query edges are traversed only in a forward direction and the query graph can be split in several unconnected components, which can remain unprocessed. To discover larger subgraphs, queries have to be traversed as weakly connected graphs. Such graphs are connected if the direction of edges is not considered. Second, traversing the query graph multiple times from different query vertices creates large intermediate results and duplicates. To reduce this overhead, the search can be done just for a single starting vertex according to a single traversal path. Third, if a failed subquery splits a query graph in several unconnected components, the largest MCSs (BMCSs) can be missed. To prevent this situation, the search can be restarted for non-processed query edges. These improvements are applied to both algorithms DISCOVERMCS and BOUNDEDMCS and described in this section.

4.3.1 Processing of Weakly Connected Components

The first drawback of DISCOVERMCS and BOUNDEDMCS algorithms implies traversal of the query graph only in a forward direction, according to the depth-first search. This can limit the size of discovered subgraphs and deliver subgraphs of potentially smaller size than could be determined if edges are traversed in both directions. To ensure the discovery of a maximum subgraph, the query traversal has to be started with a vertex, from which all vertices are reachable. Because the algorithms work only in a forward direction, it is not always possible to find the best start vertex.

Example Assume an example query in Figure 4.5, which highlights two traversal orders for the same query. A black part corresponds to a traversed subquery, a gray part models a non-traversed one. This query does not have an ideal starting vertex, because there are always non-processed query parts. 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 the maximum subgraphs only with two edges and three vertices (v_q^1, v_q^2, v_q^3 or v_q^4, v_q^2, v_q^3).

To process queries with unreachable components, we propose to construct an all-covering spanning tree, which describes the order, in which query vertices and edges are processed.

Definition 12. *Let $G_q = (V_q, E_q, u_q, f_q, g_q, A_{V_q}, A_{E_q})$ be a graph query. An **all-covering spanning tree** is a directed tree for G_q whose nodes represent query vertices and arrows*

Algorithm 6 GETNEXTEDGESTOTRAVERSE: Extraction of candidate edges for traversal from all-covering spanning tree

Input: all-covering spanning tree $tree$, query graph G_q, set of rejected edges $rejected$
Output: set $edges$ of non-traversed edges

```
1: cursor ← get cursor for tree
2: v_c ← extract query vertex pointed by cursor
3: edges ← get adjacent edges for v_c from G_q
4: edges ← filter out traversed edges from edges
5: edges ← filter out rejected from edges
6: if edges == ∅ then
7:     v_c ← mark as complete
8:     parent ← get parent for v_c
9:     if parent == ∅ then
10:        return ∅
11:    else
12:        cursor ← parent
13:        edges ←GETNEXTEDGESTOTRAVERSE (tree, G_q, rejected)
14: return edges
```

describe traversed query edges. The ***root*** *of the tree is represented by the start query vertex and the* ***cursor*** *describes the position in the tree, which is currently used to extend it. The all-covering spanning tree is* ***complete****, if it cannot be further extended and the cursor points to the root of the tree.*

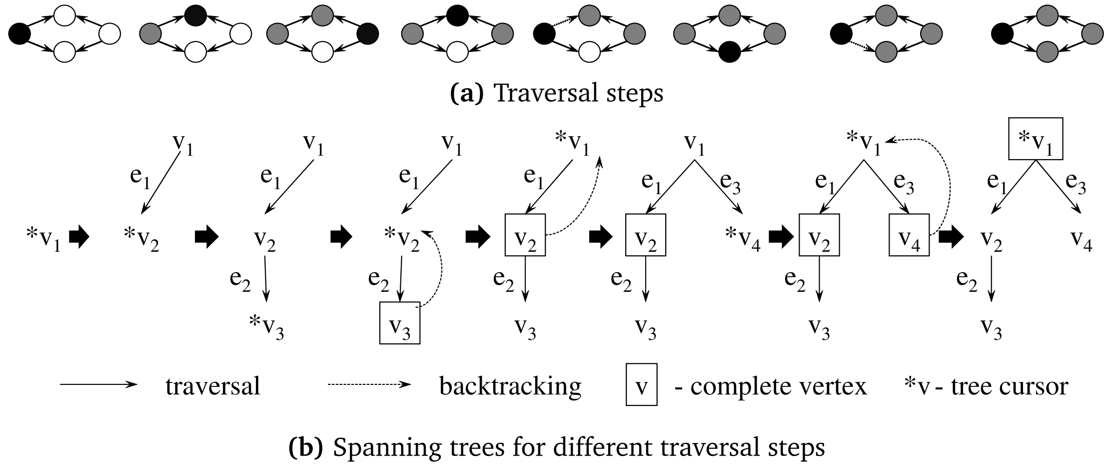
By walking a tree top-down from left to right, a traversal order can be collected, which is used to process a query graph. If a query does not violate a cardinality constraint, the whole query graph can be traversed and an all-covering spanning tree covers all query vertices and edges. At initialization, the all-covering spanning tree consists only of a single vertex as its root, which is marked as incomplete and pointed by the tree cursor.

By extending the tree, a new edge is inserted at the current cursor position with its second incident vertex. The extension is finalized after the cursor is moved to this second vertex. The exact traversal direction (backward or forward) can be easily retrieved from the spanning tree by providing the queried edge. If the source of an edge is on a higher level of the tree then this edge is forward-traversed. Otherwise, it is traversed backwards.

We adapt Algorithms 3, 4, and 5 to work with the all-covering spanning tree. From now on, we consider both incoming and outgoing edges for each query vertex at line 8 in Algorithm 3 and line 1 in Algorithm 5.

To get the candidate edges for traversal, the all-covering spanning tree is processed according to Algorithm 6. First, the last processed vertex pointed by the tree cursor is retrieved at lines 1 – 2. Then non-processed non-rejected adjacent edges are extracted for this vertex at lines 3 – 5. If the filtering removed all edges from the set, then the tree is backtracked and the procedure is repeated again at lines 6 – 13. Otherwise, the left edges are returned at line 14.

Example Referring to our example in Figure 4.5, assume that the search begins from vertex v_q^1. In Figure 4.6, we present the stepwise changes applied to the spanning tree. In Figure 4.6a, the sketches of the query tree for consequent processing steps are illustrated, where the black vertices describe the current traversal position pointed by the tree cursor, the gray vertices represent already traversed query vertices, and the



(a) Traversal steps

(b) Spanning trees for different traversal steps

Figure 4.6: All-covering spanning tree and backtracking procedure

white ones are unvisited vertices. In Figure 4.6b, the corresponding sketches for the spanning tree are provided, where the nodes are processed query vertices and edges are traversed query edges. The dashed arrows denote backtracking. The star in each tree represents the position of the tree cursor. If a vertex is arranged by a square, it and its subtrees are complete and cannot be further extended. By launching the discovery process, first vertex v_q^1 is inserted into the tree and is pointed by the cursor. This vertex has two outgoing edges. Assuming the next edge to be traversed is edge e_q^1, it is inserted into the tree, the cursor moves to its target and points to vertex v_q^2 as illustrated in Figure 4.6b. This vertex has only one non-traversed edge e_q^2 in Figure 4.6a, which is processed to vertex v_q^3. Both of them are added to the tree and the cursors points to the last processed node in Figure 4.6b. Following the same procedure, the next edge to be processed is edge e_q^4, which represents a failed edge. No other non-traversed edges exist, therefore, the system marks vertex v_q^3 as complete, and backtracks to vertex v_q^2, which becomes pointed by the cursor. Similar to v_q^3, vertex v_q^2 also does not have further edges to be traversed. It is marked as complete in Figure 4.6b, and the search backtracks to root v_q^1, which has non-traversed edge e_q^3. Therefore, it is traversed, and last non-processed vertex v_q^4 is reached by the search. It is marked as complete and the search backtracks to root v_q^1, which is finally marked as complete and the discovery process terminates.

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 and why-so-few queries, which includes all vertices and edges, and process weakly connected graphs. Moreover, the tree provides us a set of available edges for traversal. Therefore, the next question to be answered is: Which query edge should be preferred for traversal to overcome the multiple search from all query vertices and multiple traversals of the same data edges? We will answer this question in the following section.

4.3.2 Selection of Single Traversal Path

To decrease the number of intermediate results and to prevent multiple search along the same data, DISCOVERMCS and BOUNDEDMCS are extended with several heuristics to establish a single traversal path. These heuristics improve search performance, but they do not guarantee the discovery of an optimal solution.

Maximal In- And Out-Degree According to this heuristic, a query vertex and query edge are selected to be processed based on in- and out-degrees of query vertices.

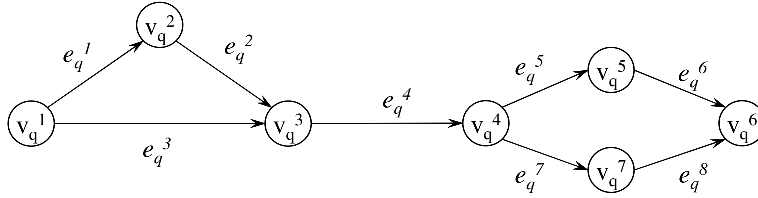


Figure 4.7: Example of missing largest subgraph caused by splitting query in unreachable components and traversal in its smaller part

A query vertex with maximal in-degree or out-degree is selected as a starting point. This method relies only on the query graph and does not consider the statistical information about the underlying data graph. This is supported by the fact that more edges need to be processed for the query vertex with a larger degree and therefore MCSs (BMCSs) can be discovered earlier.

Minimal Edge and Vertex Cardinality This heuristic selects a query vertex or query edge to be processed based on its cardinality, which shows the number of its data instances in a data graph. A query edge (vertex) with the minimal cardinality is selected as a starting point. This heuristic requires the calculation of cardinalities for all query edges and vertices in advance and proposes them for traversal according to their cardinalities in an ascending order, which allows to reduce the size of intermediate results in order to improve performance. It also supports selection of a search direction based on the cardinality of a source and a target if an all-covering spanning tree is constructed. 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, it is forwardly processed. In addition, for why-empty queries this method has the advantage that if an edge has zero cardinality then it is discarded from the traversal.

The proposed heuristics together with the construction of an all-covering spanning tree allow to process the query graph only once and therefore improve performance of DISCOVERMCS and BOUNDEDMCS algorithms. Thus, the first problem of duplicated answers and high amount of intermediate results is solved. The second question remains open: How to deal with unconnected components? The answer to this question is discussed in the following section.

4.3.3 Processing of Unconnected Components

Single traversal of a query graph works well if all query edges can be considered. However, in case of why-empty queries some edges can be missing from the data graph. In case of why-so-many queries, some edges are skipped, otherwise, the result cardinality would exceed a cardinality threshold. Such non-considered 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 one subgraph, we will miss an MCS (BMCS) from another one. Thereby, it leads to the problem of missing maximum subgraphs, which can be solved by a restart strategy.

Example The problem of missing MCSs (BMCSs) can be explained with a query graph containing a bridge. In our example query in Figure 4.7, edge e_q^4 does not have any matching data edges for why-empty queries or increases the result cardinality dramatically for why-so-many queries and therefore it serves as a bridge between two unreachable components. In this case, a MCS (BMCS) found by our algorithm DISCOVERMCS (BOUNDEDMCS) would be the left or right query parts of the graph. If the

search is conducted left, the MCS (BMCS), which is located right, will not be found. To solve this problem, we can resume the search for non-traversed edges and the final result will include MCSs (BMCSs) from both areas.

To restart the discovery process, a list of traversed query edges has to be maintained. After the first set of MCSs is returned, those edges are removed from the list that have already been traversed and the first edge to traverse is extracted from this list. This strategy facilitates the discovery of MCSs (BMCSs) for a given starting vertex if an all-covering spanning tree is constructed.

Example At the beginning, a query graph in Figure 4.7 has an empty list of traversed edges. Assuming the query is traversed from e_q^1 to e_q^2, then e_q^3 and the absence of edge e_q^4 cancels the discovery process. Therefore, processed edges $e_q^1 - e_q^3$ are moved from the list of non-traversed to traversed edges and a re-starting edge is chosen among non-traversed edges $e_q^4 - e_q^8$ for BOUNDEDMCS. For DISCOVERMCS, a re-starting vertex is chosen among source and target vertices of these non-visited edges. If the restarted search discovers larger MCSs (BMCSs) in the right query part they are returned to a user. This methodology can potentially return larger subgraphs than the original strategy with a single execution.

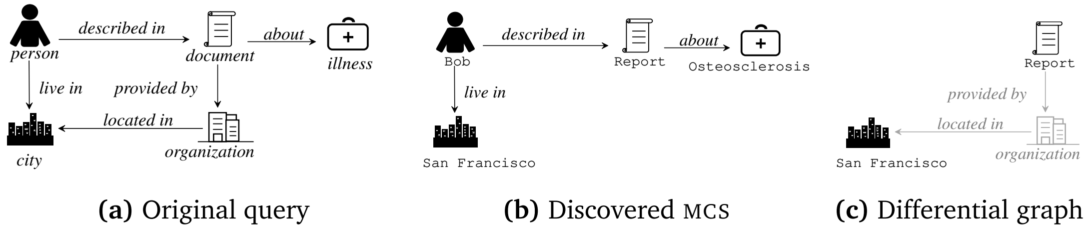
4.4 User Integration

In the previous sections, we consider the generation of subgraph-based explanations with several user-independent optimization techniques to increase performance and quality of discovered maximum subgraphs. Although the proposed methods can lead to the discovery of larger MCSs, they can miss such subgraphs, which are of interest for users. Therefore, to make the above presented solutions user-aware, they have to consider a user-preference model. For this purpose, we introduce a new heuristic for choosing a single traversal path along the query graph according to user preferences. Its task is to discover MCSs or BMCSs based on the user interest in specific query elements and to deliver the most interesting maximum discovered subgraphs.

In the following, we describe how to define user preferences in Section 4.4.1, how to adapt the establishment of a traversal path according to a user-preference model in Section 4.4.2, and how to calculate the rank of discovered MCSs (BMCSs) in Section 4.4.3.

4.4.1 Definition of User Preferences

To discover user-relevant query subgraphs, we require user preferences in query vertices and edges. Such preferences can be provided in the form of relevance weights that are float numbers $\in [0;1]$ assigned to vertices $\omega(v_q^i)$ and edges $\omega(e_q^j)$ in a graph query. They show the importance of annotated query elements for a user. Graph elements with higher relevance weights are more important than those with lower values. A weight $\omega = 0$ denotes low relevance and thus reflects the default of a vertex and an edge. The negative evidence is not defined because if a graph element is not interesting to a user then it would not be included in the query. The relevance weights form a user-preference model and are used for several purposes, e.g.: (1) for steering the discovery process in a more relevant direction, (2) for earlier processing of elements with higher relevance, and (3) in a scoring function for the ranking itself. The relevance weights facilitate the discovery of such subgraphs that are more interesting to a user, and the elimination of less relevant data subgraphs.



(a) Original query **(b)** Discovered MCS **(c)** Differential graph

Figure 4.8: Why-empty query example with user preferences

The relevance weights can be directly defined by a user. If no such assignment exists, they can be defined based on a use case or one of the following strategies.

Minimum-Representative-Element Strategy This strategy considers how well query elements are represented in a data graph. If a graph element has less representatives in a data graph, it can be of more interest to a user than generally described graph elements with more instances. Therefore, a graph element with the least number of data instances is annotated with the highest relevance weight and should be processed first during the traversal. This is valid for edges and vertices. In this case, the discovery process is managed by the minimum-cardinality heuristic and reduces the number of answers.

Maximum-Connected Vertex Strategy This strategy analyzes the degree of vertices that describes the number of connections (outgoing and incoming) in a graph query. If a graph element has the most number of connections to other vertices then it provides more information about a graph query than other vertices. Such vertices represent the topological centers of a query. As a consequence, they can be potentially of more interest to a user than less connected ones. Therefore, vertices with the maximum degree are annotated by the highest relevance weight and should be traversed first. In this case, the discovery process is managed by the maximum-incoming or maximum-outgoing heuristic, which facilitates the discovery of larger MCSs (BMCSs) as described in Section 4.3.2.

If relevance weights are defined by a particular use case, they have to consider its specific features, an objective function, which a use case tries to minimize or maximize. Some examples of objective functions are a data transfer rate in networks of hubs or traffic in road networks. If a user aims at maximizing an objective function, then graph elements with higher values of an objective function are annotated with higher relevance weights.

Example As an example, imagine a data graph derived from text documents that contains information about patients, their diagnoses, and medical institutions, which is stored together with a source description in a graph database to allow its collaborative use by several doctors. Assume a doctor is interested in names of all persons together with their illnesses, cities of residence, medical organizations, and information documents like in Figure 4.8a. If this query does not deliver any answer, it can be debugged for the reasons of the empty answer. If a doctor is more interested in the names of persons together with their diseases, then the highest relevance weights are assigned to corresponding vertices: $\omega(v_{person}) = 1$ and $\omega(v_{illness}) = 1$. The discovered results are illustrated in Figures 4.8b – 4.8c.

4.4.2 User-Centric Selection of Traversal Path

To integrate a user-preference model in the discovery of MCSs or BMCSs, the selection of a traversal path has to be adapted in such a way that the most relevant elements

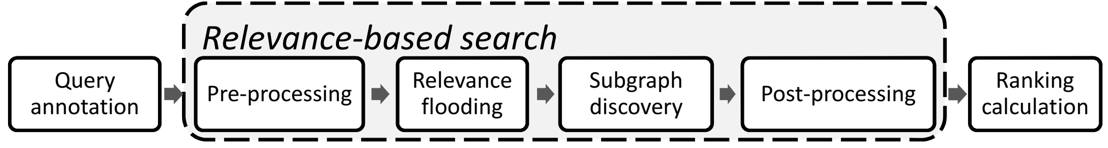


Figure 4.9: Relevance-based search

are processed first. If all query vertices and edges are annotated then the query graph can be traversed from most relevant to less relevant query elements. Otherwise, additional steps are required: pre- and post-processing of query elements and distribution of relevance weights across non-specified query elements to establish the most relevant traversal path along the query.

The relevance-based search is illustrated in Figure 4.9. After a user has annotated a query with the relevance weights, a relevance-based search is executed, which is outlined in the dashed box. At the stage of pre-processing, the relevance weights are transformed into the format required by the relevance flooding: edge relevance weights are converted into the relevance weights of incident vertices. Afterwards, the relevance flooding propagates the weights along the graph query, if at least one vertex does not have a user-defined relevance weight. Then, the subgraph discovery is executed along the most relevant traversal path, which detects the most relevant MCSs (BMCSs) by DiscoverMCS or BoundedMCS algorithms. Afterwards, the post-processing is executed over relevance weights to prepare them for final ranking.

4.4.2.1 Pre-Processing and Post-Processing of Relevance Weights

Relevance flooding considers relevance weights only on vertices. To account for the relevance on edges, edge weights are converted into the weights of incident vertices before flooding. The pre-processing consists of two steps: assignment of missing relevance weights and transformation of relevance weights. Graph elements without user-defined weights are annotated with zero values. Afterwards, the system distributes the relevance weights of edges to their incident vertices as follows. (1) The user-defined relevance weight of an edge is distributed equally across its source and target vertices. (2) Given a set of k incident edges, the relevance weight of the vertex v_q^i is the sum of the square root of k edge relevance weights $\omega(e_q^j)$, which are incident to the vertex v_q^i, and its initial relevance weight $\omega_{init}(v_q^i)$ (if any):

$$\omega(v_q^i) = \sum_{i=1}^{k} \sqrt{\omega(e_q^j)} + \omega^{init}(v_q^i) \tag{4.1}$$

The post-processing is conducted after the subgraph search; it prepares the weights for ranking. By default, the user-defined weights are used in ranking, therefore, the weights changed during the relevance flooding have to be reset to values derived at the pre-processing step. Non-annotated graph elements are specified by the minimal weights:

$$\omega^{min}(e_q^j) = \frac{1}{M}, \omega^{min}(v_q^i) = \frac{1}{N}, \tag{4.2}$$

where M is the number of query edges and N is the number of query vertices.

In order to use the relevance flooding weights for ranking, the weights for edges can be derived by multiplying the weights of their sources and targets:

$$\omega(e_q^j) = \omega(e_q^j(source)) * \omega(e_q^j(target)) \tag{4.3}$$

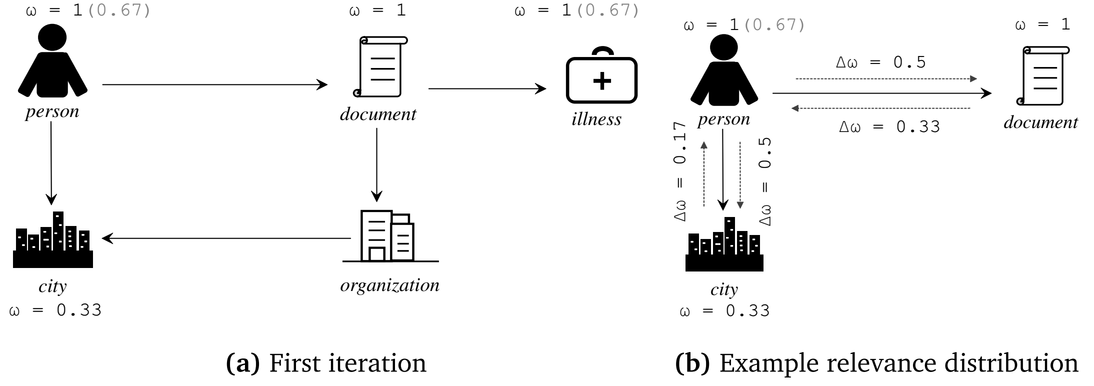
Algorithm 7 Relevance flooding

Input: query graph G_q
Output: annotated query graph G_q
1: **for all** vertex v_q^i in query graph G_q **do**
2:     **if** v_q^i has user-defined weight **then**
3:         $\omega \leftarrow$ get weight of v_q^i
4:         $neighbors \leftarrow$ get all direct neighbors of v_q^i
5:         $\Delta\omega \leftarrow$ calculate propagation weight as $\omega/$size$(neighbors)$
6:         **for all** $neighbors_j$ in $neighbors$ **do**
7:             store $\Delta\omega$ in $neighbors_j$
8: **for all** vertex $v_q^i \in G_q$ **do**
9:     increase weight for v_q^i
10: $max(\omega) = 0$
11: **for all** vertex v_q^i in query graph G_q **do**
12:     **if** $max(\omega) <$ get weight of v_q^i **then**
13:         $max(\omega) \leftarrow$ get weight of v_q^i
14: **for all** vertex $v_q^i \in G_q$ **do**
15:     **if** initial weight of $v_q^i > 0$ **then**
16:         weight of $v_q^i \leftarrow$ get initial weight of v_q^i
17:     **else**
18:         weight of $v_q^i \leftarrow$ weight of $v_q^i/max(\omega)$
19: $sum = 0$
20: **for all** vertices $v_q^i \in G_q$ **do**
21:     $sum \leftarrow sum +$ (get previous weight of $v_q^i -$ get weight of $v_q^i)^2$
22: **if** $sum <= \epsilon$ OR $\kappa >= diameter$ **then**
23:     terminate flooding

4.4.2.2 Relevance Flooding

After the relevance weights have been pre-processed, the relevance flooding can be conducted if not all query elements are annotated with relevance weights. This procedure distributes the weights along query edges and establishes the most relevant traversal path. The algorithm for relevance flooding is based on similarity flooding [97], where two schemes are matched by comparing similarities of their vertices. We extend this algorithm to propagate the relevance weights to all vertices in a graph query and to keep the initial user-defined relevance weights. The relevance flooding has to exhibit two properties in order to establish the most relevant path such as locality and stability of relevance. The locality assigns higher relevance weights to the direct neighbors and lower relevance weights to remote vertices. The stability keeps the relevance weights provided by a user and prevents the system from reducing them during flooding.

Relevance flooding works as described in Algorithm 7. At lines 1 – 9, each vertex broadcasts its value to direct neighbors according to the locality property. Afterwards, the values are normalized to the highest value at line 18 and user-defined relevance weights are set back to ensure the stability of given relevance weights at line 16. If a termination condition is satisfied, the propagation is interrupted at line 23. As a termination condition, we can use either threshold ϵ for the difference of relevance



(a) First iteration **(b)** Example relevance distribution

Figure 4.10: Relevance flooding: gray relevance weights correspond to non-resetted weights

		Iteration			
Vertex	**Initial**	**1st**	**2nd**	**3rd**	**4th**
Person	1	1	1	1	1
Document	0	1	1	1	1
Illness	1	1	1	1	1
City	0	0.33	0.33	0.34	0.34
Organization	0	0	0.06	0.09	0.11

Table 4.1: Relevance weights produced at different iterations of relevance flooding

weights of two subsequent iterations or number of iterations κ, which corresponds to the diameter of a graph query. The number of iterations κ is more appropriate, because we can be sure that relevance weights reached all elements in a query graph.

Following the example in Figure 4.8a and assigned relevance weights $\omega(v_{person}) = \omega(v_{illness}) = 1$, at each iteration equal relevance weights are distributed across all direct neighbors (an exemplary weight propagation for a query subgraph during the second iteration is shown in Figure 4.10b). Flooding considers a query graph as undirected and transmits the weights along all edges. After the first iteration, vertices $v_{document}, v_{city}$ get the propagated relevance weights from $v_{person}, v_{illness}$ according to the locality property (see Figure 4.10a). Vertex $v_{organization}$ still remains without a relevance weight, because it is two hops away from the vertices with non-zero initial weights and during one iteration the weights are distributed only among direct neighbors. After each iteration, the weights are normalized by the highest value and weights of vertices, which were originally annotated with user-defined values, are reset to their initial relevance values. The gray relevance weights in brackets for vertices $v_{person}, v_{illness}$ show the weights without reset. The process is repeated, until it converges according to specified threshold ϵ or when number of iterations κ has exceeded the diameter of the graph query. The final relevance weights are presented in Table 4.1.

4.4.2.3 Maximum Common Subgraph Discovery with Relevance Weights

User-defined relevance weights represent an interest of a user in dedicated graph elements: such elements have to be processed first. Therefore, a traversal path between all relevant elements in a query graph are treated as a cost-based optimization, where the relevance of a traversal path is maximized.

The DiscoverMCS and BoundedMCS algorithms are adapted to consider user-defined relevance weights as following. (1) The vertex with the highest relevance weight is chosen as a start vertex. (2) That edge is traversed first, which second vertex has the highest relevance weight among second vertices of all incident edges. This process continues until all vertices and edges in a query graph are considered. If a chosen edge does not satisfy the cardinality constraint then the search is adapted dynamically—the next best query edge and vertex are chosen for traversal. If several vertices have the same weight, then an edge is chosen dependent on a used heuristics from Section 4.3.2. The proposed strategy steers the search in the most relevant direction first, guaranteeing the early discovery of the most relevant parts.

4.4.3 Rank Calculation

After MCSs or BMCSs have been discovered they have to be ranked. The answers with higher relevance weights are ranked higher. A rating score is calculated based on the values of edges and vertices a result comprises. After ratings of all results are computed, they are normalized to the highest discovered rating score. Given N vertices and M edges in a query graph G_q, the rating of discovered subgraph G'_d is calculated as follows

$$rating(G'_d) = \sum_{i=1}^{i=N} \begin{cases} \omega(v_q^i) & \text{, if } v_d^i \in G'_d \\ 0 & \text{, otherwise} \end{cases} + \sum_{j=1}^{j=M} \begin{cases} \omega(e_q^j) & \text{, if } e_d^j \in G'_d \\ 0 & \text{, otherwise} \end{cases} \quad (4.4)$$

Following our example in Figure 4.8, the rating of the discovered subgraph in Figure 4.8b before normalization equals to $rating = 3$ by default or $rating = 5.68$ by using the relevance flooding weights from the fourth iteration (see Table 4.1).

4.5 Evaluation

In this section, we evaluate DiscoverMCS and BoundedMCS algorithms on DBpedia and LDBC data sets for the empty-answer and too-many-answers problems. The description of both data sets and evaluated queries are provided in Appendix A.

The DBpedia queries taken from the logs for the SPARQL endpoint mainly represent a star topology with a limited number of predicates on query vertices and edges. The LDBC queries taken from the interactive workload of the LDBC benchmark have diverse topologies and multiple predicates on vertices and edges.

The evaluation of the DBpedia data set includes five queries with empty results for DiscoverMCS and one query with a star topology delivering more than 200,000 subgraphs for the BoundedMCS algorithm. The LDBC evaluation considers eight instances of four queries LDBC Query 1 – Query 4 such that four of them deliver no results and the remaining four of them have result cardinalities varying between 1,626 and 5,020 data subgraphs.

In this evaluation, we mainly focus on the following questions:

- Do the proposed algorithms discover query subgraphs responsible for the delivery of unexpected results?
- Does the construction of an all-covering spanning tree increase the quality of the discovered data subgraphs?
- Does the performance of the algorithms degrade with constructing an all-covering spanning tree?

Abbreviation	Short Name	Description
MAX(CARD)	Maximal cardinality	Edge with maximal cardinality
MIN(CARD)	Minimal cardinality	Edge with minimal cardinality
MAX(IN)	Maximal in-degree	Edge which non-traversed incident vertex has maximal in-degree
MIN(IN)	Minimal in-degree	Edge which non-traversed incident vertex has minimal in-degree
MAX(OUT)	Maximal out-degree	Edge which non-traversed incident vertex has maximal out-degree
MIN(OUT)	Minimal out-degree	Edge which non-traversed incident vertex has minimal out-degree

Table 4.2: Evaluated heuristics for choosing traversal paths

- Which single-traversal strategy should be preferred during the discovery of MCSs (BMCSs) to solve empty-answer and too-many-answers problems?

In the following sections, we first evaluate the DISCOVERMCS algorithm for DBPEDIA and LDBC queries for the empty-answer problem. Afterwards, we test the BOUNDEDMCS algorithm for different cardinality thresholds for both data sets.

4.5.1 DISCOVERMCS Algorithm for Empty-Answer Problems

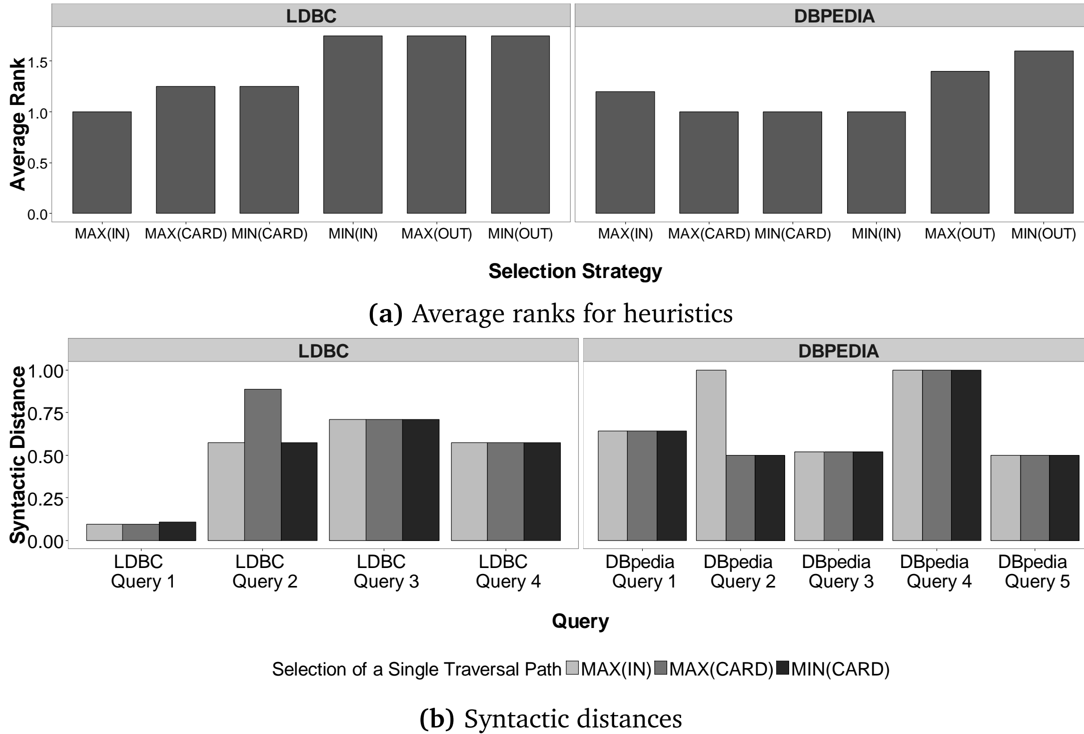
In the evaluation of the DISCOVERMCS algorithm, we compare seven strategies, which select a single traversal path, in different setups: with or without constructing an all-covering spanning tree, and with a single traversal or with restarting the search. In total, each query is evaluated in 28 experimental configurations.

As a performance measure, we consider response time, which shows how much time the MCS search requires in a specific experimental configuration. This measure depends on the number of processed data subgraphs. As a quality measure, the syntactic distance described in Chapter 3.2.2 is used, which shows how different two queries appear to a user. For evaluating DISCOVERMCS and BOUNDEDMCS, this similarity measure describes how the query corresponding to the best discovered MCS differs from a failed one. Higher values for the syntactic distances describe stronger differences from the original query and the lower values are preferable over higher ones. In addition, the number of discovered MCSs is also provided.

In this section, we start with the evaluation of selection strategies for a single traversal path along a failed query. The best-performed strategy is then used in all subsequent tests. Afterwards, we show how evaluated metrics differ by considering the construction of a spanning tree and the restart strategy.

Selection of Traversal Paths

In the first experiment, we evaluate different strategies for selection of a single traversal path. For this experiment, we construct a spanning tree and launch the discovery of MCSs without restarting for four LDBC and five DBPEDIA queries. For each considered heuristic presented in Table 4.2, we calculate a syntactic distance of its explanation. Afterwards, all runs for the same query are ranked based on the derived syntactic distances, the runs with smaller distances are ranked higher than ones with larger



(a) Average ranks for heuristics

(b) Syntactic distances

Figure 4.11: Comparison of heuristics for choosing single traversal path for DISCOVERMCS

distances. Configurations with the same syntactic distances have the same rank. If a tested configuration discovers no MCSs, its syntactic distance equals 1. We calculate the average rank for the same configuration among all tested queries and present it in Figure 4.11a.

The top-3 strategies for the LDBC graph include maximal in-degree, maximal cardinality, and minimal cardinality. However, their average ranks are very similar to each other. For the DBPEDIA queries, the following heuristics perform best: minimal in-degree, minimal cardinality, and maximal cardinality. Minimal in-degree performs very badly for the LDBC graph and therefore it is not considered in subsequent experiments. As we can see, the cardinality-aware strategies perform better, because they omit non-existing elements and can always deliver at least some data subgraphs. In contrast, most of the strategies, which do not consider cardinalities, can fail to produce an explanation, if they start the search from query elements, which are not represented in the data graph.

In Figure 4.11b, we illustrate syntactic distances for three best-performed heuristics including maximal in-degree, minimal cardinality, and maximal cardinality for all evaluated queries. All strategies show very similar results except two runs. In the first case, the maximal in-degree does not find a solution for DBPEDIA QUERY 2 (the corresponding query graph is illustrated in Figure A.3c), because it starts the discovery process on edge e_q^2 from vertex v_q^2, which has no matching instances in the data graph. Therefore, an empty result set is returned to a user and no explanation is generated. In order to discover at least some results, it is necessary to consider cardinalities of vertices and edges by constructing a traversal path. In the second case, the maximal-cardinality heuristic derives a worse explanation for LDBC QUERY 2. In this query illustrated in Figure A.2b, only the left or the right side is represented in the data graph. The maximal-cardinality heuristic launches the discovery process from vertex v_q^2. Although it has multiple data instances, none of their connections matches the adjacent

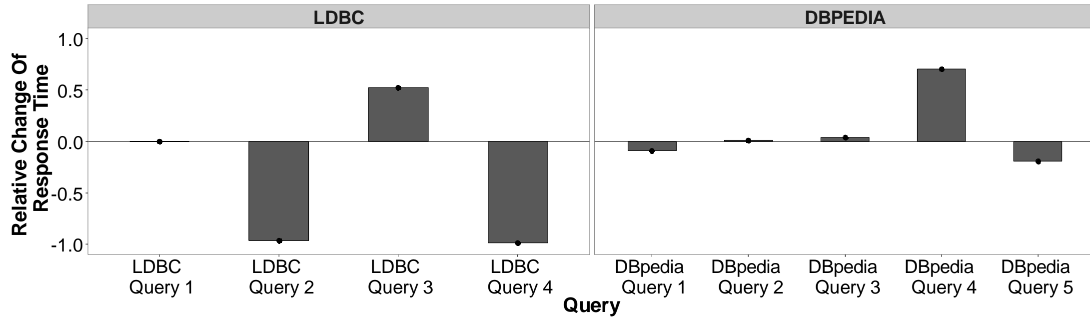


Figure 4.12: Relative change of response time for DISCOVERMCS algorithm with construction of spanning tree

query edge. As a result, this part of the query becomes unconnected and consists only of a single vertex. A larger subgraph can only be found if we restart the discovery process from non-processed query vertices and edges. From top-3 heuristics for choosing a single traversal path, the minimal-cardinality strategy shows stable behavior among all queries and therefore is used in further experiments.

To conclude, in order to discover large data subgraphs, it is necessary to consider cardinality of vertices and edges by choosing a single traversal path. To prevent the loss of a larger subgraph, we have to be able to restart the discovery process for non-processed query parts.

Construction of an All-Covering Spanning Tree

In the previous test, we concluded that the minimal-cardinality heuristic provides stable results among all queries and almost always derives the best result. Therefore, in this experiment we consider only this strategy and evaluate the construction of a spanning tree.

In this evaluation, we use experimental results without a spanning tree as a baseline and calculate relative changes for syntactic distances, numbers of discovered MCSs, and response times. A negative value of a relative change means that the use of a spanning tree improves the corresponding metric. In eight out of nine cases, the same data subgraphs with the same syntactic distances are delivered independently of constructing a spanning tree. However, in LDBC QUERY 4, its creation is necessary, because it reduces the syntactic distance up to 40% and therefore delivers a better explanation. For this query in both setups, the search starts with edge e_q^5 (see Figure A.2d), which has vertex v_q^3 as a source and vertex v_q^1 as a target. By considering a spanning tree, DISCOVERMCS can choose a traversal direction. Therefore, v_q^3 is used by default when a spanning tree is not created, which leads to a large number of small graphs consisting only of a single vertex. It cannot traverse further to less representative vertices and the search terminates. In contrast, with a spanning tree, the backward traversal is allowed and the discovery process begins at vertex v_q^1, which has only a few instances in the data graph, and detects a smaller amount of data subgraphs and of a larger size. The use of a spanning tree improves not only the quality of discovered solutions, but also can reduce the response time as presented in Figure 4.12. For this query, the number of discovered MCSs reduces twice.

The construction of a spanning tree can also cause changes in the performance as shown in Figure 4.12. In four out of nine cases, it allows to reduce the search time up to two times, because it facilitates early processing of less representative query elements and therefore reduces the size of intermediate results. For LDBC QUERY 1, a spanning

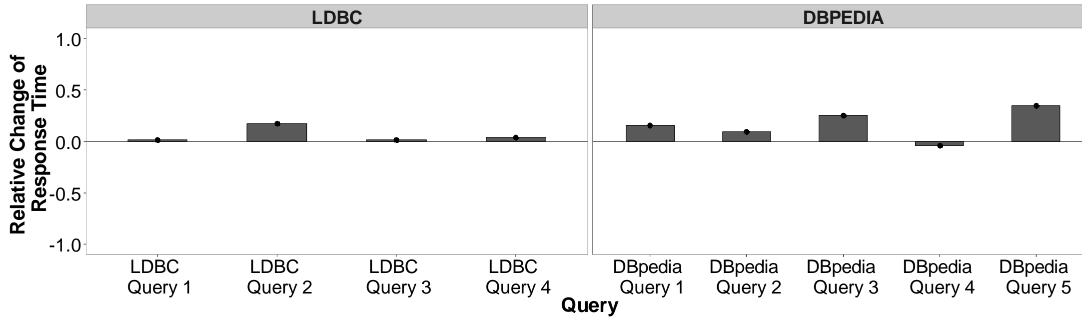


Figure 4.13: Relative response time of DISCOVERMCS algorithm for restart with minimal-cardinality heuristic

tree does not change a traversal order and therefore, there is no change in time or quality of generated explanations. In remaining four out of nine cases, the use of a spanning tree increases the response time. However, for DBPEDIA QUERY 2 and QUERY 3 this increase is minor. For LDBC QUERY 3 and DBPEDIA QUERY 4 it does not exceed 70%.

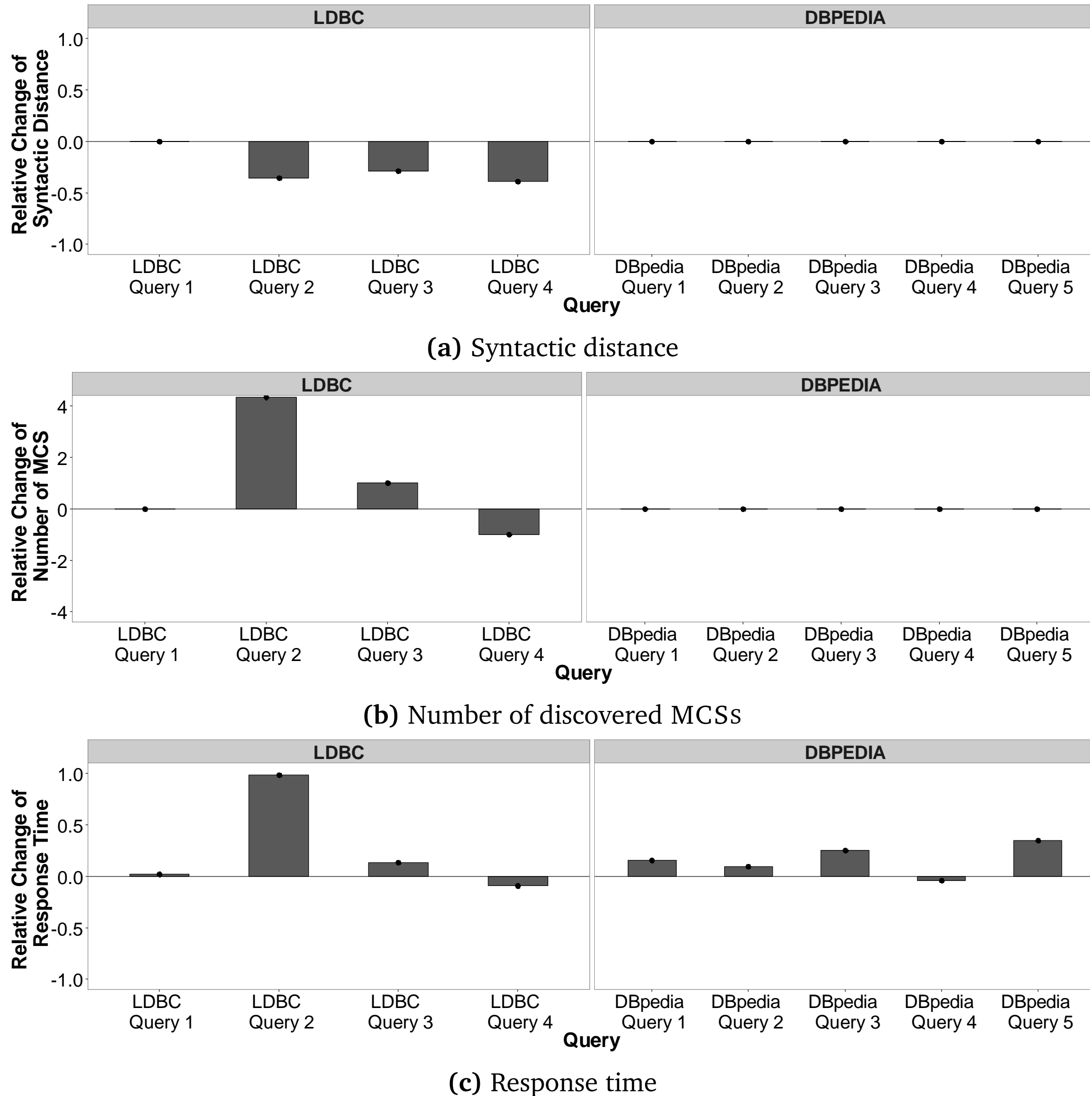
To summarize, the use of a spanning tree derives results of the same or even better quality and therefore, it should be considered during the search for MCSs. It allows to traverse edges in a backward direction and to cover vertices without incoming edges. Performance and quality gain of derived explanations depends on the cardinality of query elements and query topology. The use of a spanning tree can change the traversal order established by a traversal heuristic in such a way that backward edges can be processed earlier. This allows to discover larger subgraphs earlier. The next optimization feature, which we evaluate, is re-execution of the discovery process if some query edges and vertices were not visited.

Search Restart vs. Single Traversal

In the previous experiment for the DISCOVERMCS algorithm, we evaluate whether the use of a spanning tree is advantageous for the discovery of MCSs. In this section, we restart the search and consider relative metrics for the syntactic distance, number of discovered MCSs, and response time. These measures are calculated in respect to results for a single execution. Negative values describe the improvement achieved by the restarting the discovery process.

According to the evaluation results, independent on whether the discovery process has been restarted or not, for all queries we receive the same number of MCSs and of the same quality. It means that no query has been split in unreachable components during the search and therefore the restart strategy does not lead to any improvement. Only the performance changes, which is illustrated as a relative change of response time in Figure 4.13. It increases by up to 35%, which is required to restart the search and to check, which vertices and edges have to be processed. The largest increase is observed for DBPEDIA queries, which have a larger number of discovered data subgraphs, for which the restart is triggered.

To show the advantages of restarting the discovery process in terms of quality of discovered MCSs, in Figure 4.14, we present evaluation results for the minimal in-degree heuristic with and without re-executing the search, which according to its rank described in Figure 4.11a has the best rank for DBPEDIA queries and performs badly for the LDBC queries. The restart strategy improves the quality of discovered MCSs for three out of nine evaluated queries, which are represented by the LDBC queries.



(a) Syntactic distance

(b) Number of discovered MCSs

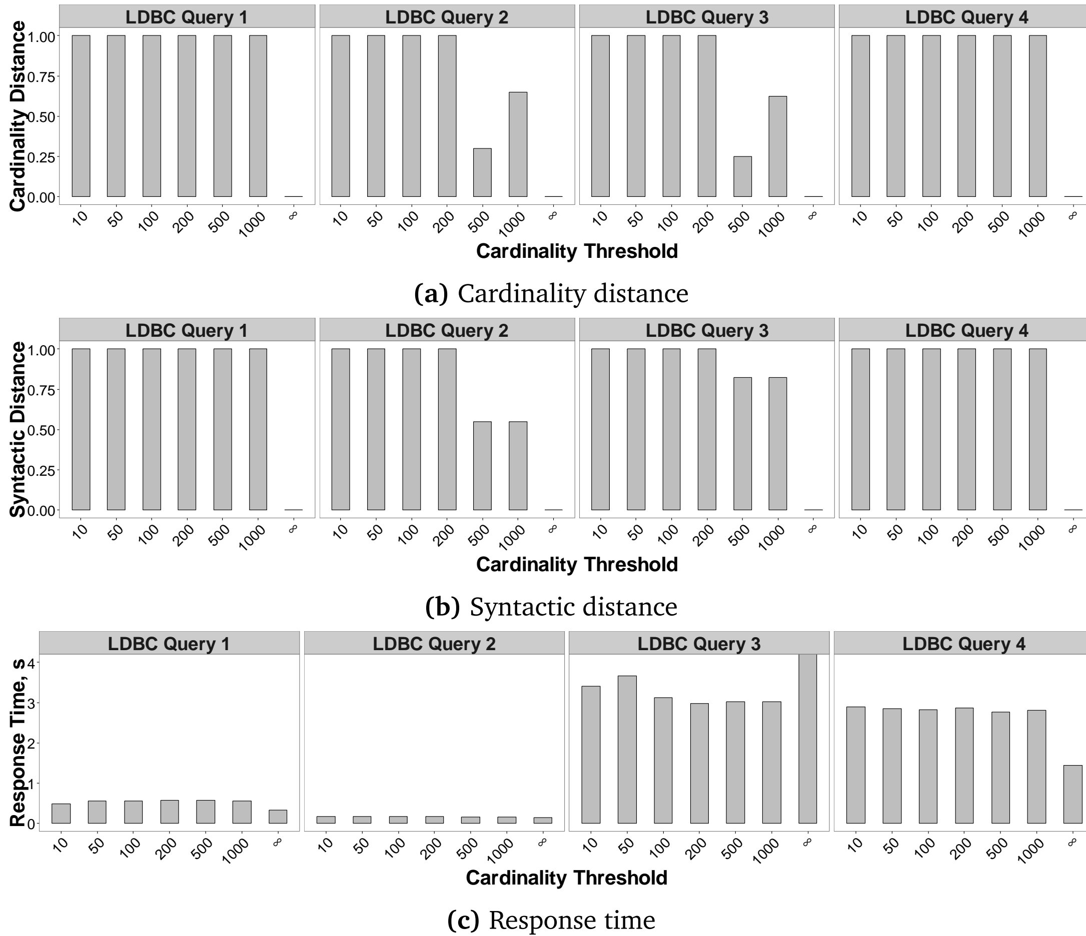
(c) Response time

Figure 4.14: Evaluation results of DiscoverMCS algorithm for restart with minimal in-degree heuristic

The syntactic distance reduces for them by from 29% to 39%. In these cases, the heuristic for choosing a traversal path splits a query in unconnected components such that the discovery process is conducted only in a smaller query part. By restarting the search, we force the algorithm to traverse a non-processed query part and to discover larger MCSs with smaller syntactic distances. This quality improvement affects also the number of discovered MCSs and response time. For LDBC Query 2, the amount of delivered data subgraphs is four times as big as without restarting the search, which increases the response time only twice. For the remaining six out of nine queries, the restart increases the response time maximum by 35%.

To summarize, to restart the discovery of MCSs is advantageous if a failed query is split by a traversal-path strategy in several unconnected components. In this case, restart can balance the drawbacks of the used traversal-path strategy and improve the quality of discovered subgraphs. However, it comes with additional costs for restarting, which do not exceed 35% for cases if no quality improvement is expected. Therefore, the restart strategy should be used in order to discover high-quality MCSs.

Summary To conclude, in this evaluation, we test the DiscoverMCS algorithm on two data sets. While the LDBC queries are characterized by different topologies and

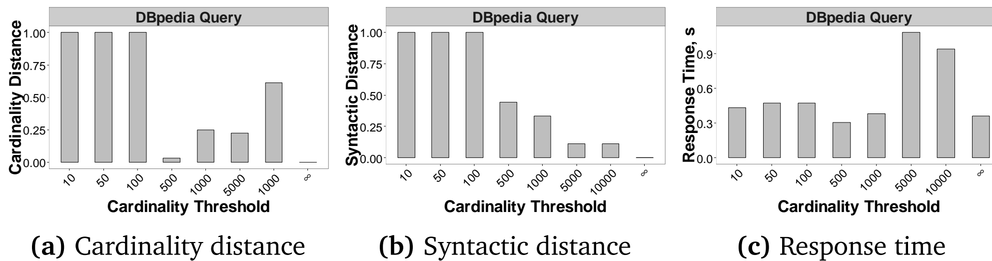


(a) Cardinality distance

(b) Syntactic distance

(c) Response time

Figure 4.15: Evaluation results of BOUNDEDMCS algorithm with single traversal for LDBC data graph



(a) Cardinality distance **(b)** Syntactic distance **(c)** Response time

Figure 4.16: Evaluation results of BOUNDEDMCS algorithm with single traversal for DBPEDIA data graph

multiple attributes, the DBPEDIA queries represent stars with only a few attributes. This characteristic explains the difference in evaluation results of heuristics for selecting a single traversal path: While the diversity of the LDBC queries results in different quality of produced results, similar explanations are generated by different heuristics for the same DBPEDIA queries. To increase the quality of discovered MCSs, the discovery process has to be conducted with constructing an all-covering spanning tree and its restarting for non-considered graph elements.

4.5.2 The BOUNDEDMCS Algorithm for the Too-Many-Answers Problem

The second set of experiments evaluates the BOUNDEDMCS algorithm, which was introduced in Section 4.2.2. For the LDBC data set, the configuration of the LDBC queries

from Appendix A with cardinalities of the class C_3 are chosen, which are provided in Table A.1. Considering a typical star topology for the DBPEDIA graph, we create only a single query for its evaluation presented in Table A.2 as DBPEDIA QUERY, which originally delivers more than 200,000 solutions.

We use eight cardinality thresholds $[10; 50; 100; 500; 1000; 5000; 10000; \infty]$ for DBPEDIA queries and seven thresholds $[10; 50; 100; 200; 500; 1000; \infty]$ for LDBC queries, where ∞ means that no cardinality threshold is considered. We evaluate them in four different configurations, which include single-traversal search or restart and construct or do not construct an all-covering spanning tree. Considering large intermediate results, which can be created by non-directing the search along less representative elements, we use a minimal-cardinality heuristic to select a single traversal path. We provide the response time in seconds as a performance measure. The quality measure is represented by the syntactic and cardinality distances, which are described in detail in Section 3.2. The cardinality distance is further normalized to a cardinality threshold, in order to make it comparable across different queries.

Consideration of a Cardinality Threshold

We start this set of experiments with evaluation of different cardinality thresholds for the BOUNDEDMCS algorithm, whose results are illustrated in Figure 4.15 for the LDBC data graph and in Figure 4.16 for the DBPEDIA data graph. The y axes describe evaluated quality or performance metrics and the x axes denote the used cardinality thresholds. In this evaluation, we execute the search only one time without restarting and construct a spanning tree.

First, we analyze how well the BOUNDEDMCS algorithm achieves its goal: The result size of a generated explanation should not exceed a cardinality threshold. With changing the threshold for the result size, the cardinality distance of discovered BMCSs changes as well, which is illustrated in Figure 4.15a. By increasing the cardinality threshold, cardinality and syntactic distances typically decrease. While the algorithm aims at discovering BMCSs by not violating the cardinality threshold, in some cases the cardinality distance for a higher threshold can be larger than with a smaller threshold. This situation happens for example for LDBC QUERY 2 for cardinalities 500 and 1,000. Explanations generated for these two cases are equal and have the same syntactic distance as presented in Figure 4.15b. By increasing the cardinality threshold and keeping the syntactic distance the same, the cardinality distance grows, which is illustrated by its spikes in Figure 4.15a.

Considering the original cardinality of a result set, which varies between 1,626 and 5,020, we could assume that starting from cardinality threshold $C_{thr} = 1,000$, the cardinality and semantic distances should approach 0. However, this assumption is wrong. During the search, the cardinality of BMCSs can violate the cardinality threshold multiple times, which depends on the topology of a query, the complex dependencies between elements and their predicates, etc. As a result, although the cardinality threshold is similar to the cardinality of the original result set, the size of the discovered BMCSs does not reach its original value.

In Figure 4.15b, the syntactic distances for discovered explanations are presented. With increasing a cardinality threshold, we receive better explanations with smaller syntactic distances. By small values of cardinality threshold $\in [10, 200]$, no explanation can be generated for LDBC queries and the corresponding syntactic distances equal 1. LDBC QUERY 2 and QUERY 4 are the most difficult queries for rewriting, because all their elements have cardinalities larger than any used cardinality threshold. Therefore, no edge and vertex can serve as starting elements for the BOUNDEDMCS algorithm.

In Figure 4.15c, response time of generating the explanations is presented as a performance metric. The response time reduces with the increasing cardinality threshold, because at each step less data subgraphs are considered. The corresponding syntactic distances are represented in Figure 4.15b.

The evaluation results for DBPEDIA QUERY are illustrated in Figure 4.16. With increasing the cardinality threshold, the syntactic distance of discovered BMCSs decreases, which corresponds to the goal of the BOUNDEDMCS algorithm: the size of a BMCS should be increased without violating a given cardinality threshold.

The cardinality distance becomes smaller, which can also be explained by the topology of the evaluated query: each time we add a new edge to a star, it increases the cardinality of a result set. We also see that for $C_{thr} = [1,000; 10,000]$ the cardinality distance increases, which can be explained by the fact, the result size grows more slowly with introducing new edges than the increase of a cardinality threshold. This very similar results can be explained by the topology of the evaluated query, which represents a star. Each time we require a backtrack, we return to its center, if we start from it. Otherwise, we arrive at the center after we processed the first edge.

To conclude, with increasing a cardinality threshold, better explanations can be discovered that have smaller syntactic distances. However, if a query initially has a high number of answers and each query edge has a cardinality, which exceeds a cardinality threshold, no explanation can be generated.

In the following experiments, we also analyze the influence of a spanning tree and restart strategy.

Construction of an All-Covering Spanning Tree

Consideration of a spanning tree for LDBC queries affects only experimental runs for high cardinality thresholds, which exceed the original cardinalities. For small cardinalities, the quality and performance are similar for both setups: with or without a spanning tree.

Therefore, for this evaluation, we provide additional experimental results for $C_{thr} = [5,000; 10,000]$. For the LDBC QUERY 1 and LDBC QUERY 2, the complete queries can be processed without violating a cardinality threshold, which change corresponding cardinality distances.

The construction of a spanning tree improves the quality of generated subgraph-based explanations by up 0.35 for high cardinality thresholds for LDBC QUERY 1 – QUERY 2 and by up to 0.75 for the LDBC QUERY 3. For LDBC QUERY 1 and QUERY 2, the spanning tree has the same response time or varies only slightly. For the LDBC QUERY 4, it is even less than for the configuration without a spanning tree, which can be explained by the smaller number of intermediate results. For the LDBC QUERY 3, the processing of a spanning tree increases the response time by up to 0.5 s for low cardinality thresholds. The strong increase in response time for the case without limiting the size of a result set is explained by 75% of quality improvement, which is achieved if a spanning tree is constructed.

For the DBPEDIA query, no difference in quality of generated explanations is observed. In this case, the discovery process begins from the center of the star, vertex v_q^0 (see the query graph in Figure A.3f), and visits query edges in the same traversal order with and without a spanning tree. We observe only a minor variation of response time (less than 5%) by considering a spanning tree, which can be tolerated.

To conclude, the construction of a spanning tree can improve the quality of generated explanations for high cardinality thresholds. For small cardinality thresholds, the

BOUNDEDMCS algorithm behaves similarly, because only a few query elements with cardinalities less than a cardinality threshold are considered.

Search Restart vs. Single Traversal

In this set of experiments, we consider multiple restart and single traversal for the BOUNDEDMCS algorithm. Like in the previous evaluation, we consider here some additional cardinality thresholds $\in [5,000; 10,000]$. In most cases, syntactic distances do not differ between the single-run strategy and multiple restarts. However, for LDBC QUERY 3, the syntactic distance improves up to 50% for high cardinality thresholds if we restart the discovery process. Such a limited difference of the syntactic distance can be explained as follows. As a heuristic for choosing a traversal path, we use a minimal-cardinality strategy, which visits elements with a smaller number of instances in the data graph first. Therefore, it is very unlikely that restarting the search from elements with a larger number of data representatives might reduce the syntactic distance. However, it can be possible for high cardinality thresholds, which for example happens to LDBC QUERY 3.

The restart strategy increases the response time very strongly for small cardinality thresholds for LDBC QUERY 3, which is up to three times so high as the time for the single-traversal search. However, the increase is rather small for LDBC QUERY 1 and QUERY 2. The increase is caused by the fact, that for small cardinality thresholds no explanation can be generated and the syntactic distance equals one. As a consequence, the restart strategy tries to re-execute the discovery process from all query elements. With increasing a cardinality threshold, some explanations can be found. Therefore, the restart re-executes the discovery process from a smaller number of query elements and therefore the response time decreases.

Similar to the evaluation of using a spanning-tree, in this experiment the restart strategy and single traversal behave very similarly for different cardinality thresholds. The restart strategy produces explanations with the same cardinality distance as a single traversal, because both of them deliver in most cases the same BMCSS. A small variation can only be seen for LDBC QUERY 3, where for high cardinality thresholds we receive different BMCSS. In this case, the restart strategy discovers a better explanation by restarting.

The evaluation results for the DBPEDIA graph are similar to the results derived for the LDBC graph. The restart strategy increases the response time in the same way: it is higher for smaller cardinality thresholds because only small explanations can be generated and multiple query elements remain for the restart. The restart strategy does not affect the quality, which remains the same for both configurations: a single traversal and with restart. As a consequence, the cardinality distance behaves in exactly the same way. In both setups, the discovery process begins at central vertex v_q^0 (see Figure A.3f), from which all vertices are reachable. Therefore, no unconnected components are created by using only a single traversal along the query graph and therefore, the restart does not give any advantages.

In this section, we evaluate the BOUNDEDMCS algorithm on two data sets for different cardinality thresholds and consider three metrics such as syntactic and cardinality distances and response time. We conclude that the use of a spanning tree can improve the quality of generated subgraph-based explanations and therefore it should be used in the debugging process, if a directed query graph becomes unconnected during its traversal. The use of a restart strategy does not give a clear advantage over a single-traversal for low cardinality thresholds if the minimal-cardinality heuristic for choosing a traversal path is used for the too-many-answers problem or maximal-cardinality

heuristic is applied for the too-few-answers problem. In such setups, the restart strategy can be omitted. The restart strategy is advantageous in such scenarios where a single traversal makes a query graph unconnected and a larger BMCS can be missed.

4.5.3 Evaluation Summary

In this section, we evaluate two algorithms for generating subgraph-based explanations such as DISCOVERMCS and BOUNDEDMCS. Both methods are able to deliver differential graphs corresponding to the query part, which are responsible for no or too many answers. We optimize both approaches by online adapting a single traversal path along the query graph and show that just a single traversal can deliver good results by the use of minimal-cardinality traversal strategy. Any poor performance of a traversal strategy can be compensated by restarting the search along non-traversed query branches. DISCOVERMCS and BOUNDEDMCS can discover larger MCSs and BMCSs with constructing the all-covering spanning tree without strong performance degradation.

4.6 Summary

In this chapter, we investigated the first kind of explanations—subgraph-based explanations, which have to be provided by the debugging tool for pattern matching queries. This kind of explanations describes what was the reason of unexpected results as differential subgraphs, which are missing from the data graph for the empty-answer problem or violate the cardinality threshold for too-few- and too-many-answers problems. In addition, a user receives maximum common connected subgraphs between a query graph and a data graph, which correspond to the cardinality constraint.

To generate subgraph-based explanations for the empty, too-few-, and too-many-answers problems, we proposed two algorithms: DISCOVERMCS and BOUNDEDMCS. Both methods 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 the 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, constructing a spanning tree 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-based explanations are query-based ones, which focus on the query topology and deliver MCSs. They can be used as an upper limit for rewriting of a failed query. Still, they do not give any detail about query predicates and defined types and therefore in the following chapters more fine-granular explanations will be presented,

which represent rewritten queries derived by changing the types and predicates of query elements.

5

Coarse-Grained Why-Empty Query Modification

In the previous chapter, 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 chapter, we will go one step further and consider predicates for attribute value on edges and vertices during explanation generation. We will 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 from 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.

In Section 5.1, the general architecture of the why-empty system, the runtime process for query rewriting, and its central component, query relaxation, are described. The relaxation process is supported by a cardinality estimation module, which is introduced in Section 5.2. In Section 5.3, a comparison of rewritten queries is described in detail. The model for integrating user preferences in the relaxation process is outlined in Section 5.4. Finally, the proposed approach is evaluated in Section 5.5. Parts of the modification-based approach presented in this chapter have been published in [138, 139, 143].

5.1 Predicate- and Topology-Aware Modification Process

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 engine described in this chapter.

5.1.1 System Architecture

The why-empty system illustrated in Figure 5.1 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

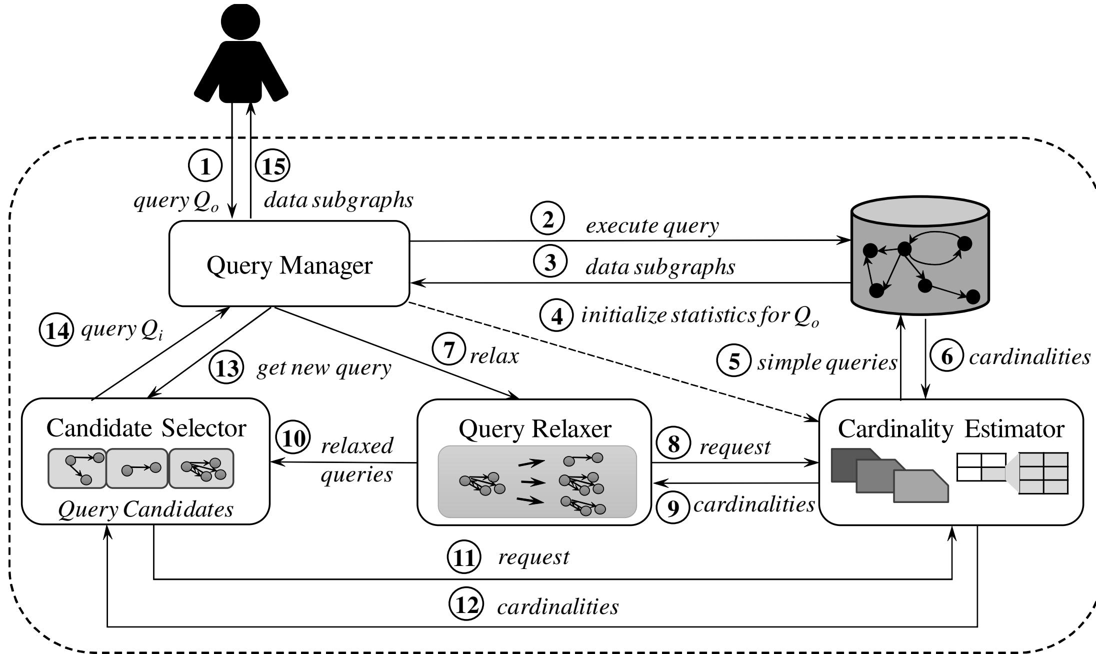


Figure 5.1: System architecture for why-empty query rewriting

a request the relaxation process is triggered, which starts with initializing the *cardinality estimator* that maintains, calculates, and estimates cardinalities for cardinality queries. At the initialization, the cardinality estimator collects some cardinalities from the graph database, which is described in detail in Section 5.2. Afterwards, 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 can be optimized with applying different relaxation strategies reducing the size of candidate space by choosing the most promising vertex and edge to be relaxed. These strategies are described in Section 5.1.2. 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, which is described in detail in Section 5.3. The query manager receives the best query candidate from the queue and executes it 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.

To summarize, the modification process is based on 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. In the following, we will discuss different aspects of this modification process starting with query relaxation.

5.1.2 Query Relaxation

The main component of the why-empty engine is the query relaxer that receives a query as an input and produces a set of query candidates (relaxed versions of the query) as an output.

To modify a query, the relaxation manager uses *relaxation operations* and *relaxation heuristics*. Relaxation operations have been already presented in Section 3.2.1 and include the deletion of a vertex, an edge, a type, a direction, and a predicate. A refined

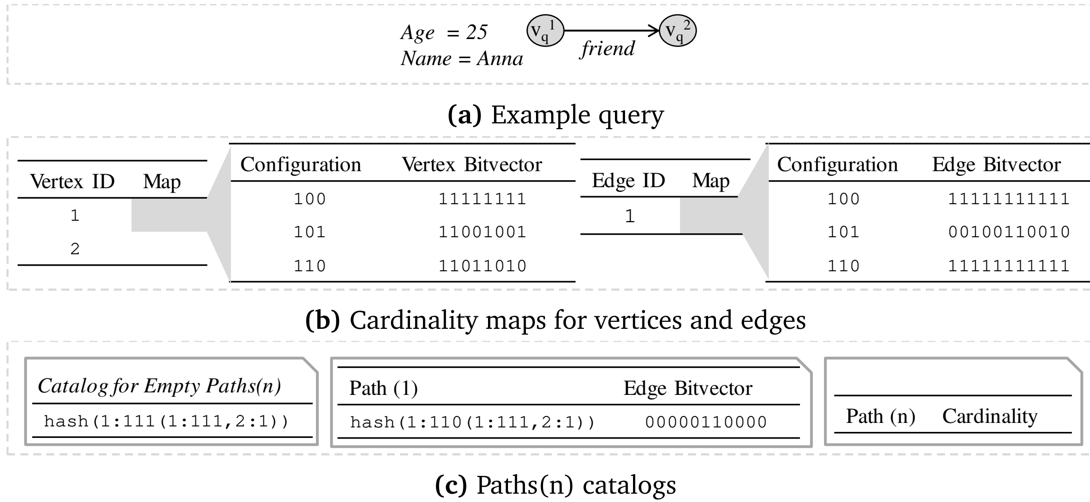
query is generated by applying one of these operations on a query. By default, only one change is induced by a single iteration. To model more complex changes like subgraph deletion, several relaxation iterations have to be executed. However, some operations can cause additional relaxations such as the edge deletion, which removes also the type, the direction, and predicates of an edge. The vertex removal deletes a vertex together with its adjacent edges.

In general, the total number of generated query candidates at a single iteration is $k \cdot (m_q + n_q)$, where m_q is a number of query edges, n_q is a number of query vertices, and k is a number of relaxation operations. This candidate space represents the full relaxation space for a query and a relaxation iteration. With an increasing number of iterations l, the total candidate space grows and can include up to $k \cdot l \cdot (m + n)$ candidates. To minimize the number of generated candidates and thus to reduce search space, relaxation heuristics are proposed which choose the most promising query elements to relax.

Minimum Cardinality of Edges and Vertices This heuristic assumes that less representative vertices and edges in the data graph are more likely to produce empty results than query elements with a large number of instances. Therefore, the first proposed strategy relaxes query elements with the lowest cardinalities in the data graph. To discover such vertices and edges, estimated cardinalities for each query vertex and edge are provided by the cardinality-estimation module. At least one edge and vertex are chosen for the relaxation, which have the lowest cardinality. If several elements have the same number of data instances, all of them are relaxed and multiple query candidates are generated.

Maximum Impact & Minimum Vertex Cardinality Relaxation of specific query edges and vertices can have different impact on the rest of the query graph. Some relaxations can cause more changes in the query than others. A higher impact caused by a single relaxation can reduce the number of iterations and facilitate early discovery of a non-failed query. Therefore, the second heuristic tries to discover and relax the most influencing query element. This heuristic is based on the calculation of vertex and path cardinalities. Vertices are selected according to the first relaxation heuristic. To choose an edge, it has to be observed how cardinalities of its direct neighbors are changed by its relaxation. The total impact of a relaxation operation for this particular edge is modeled as the sum of relative path cardinalities of neighboring edges (As a reminder, a single path cardinality shows the number of edge instances by considering specific source and target). Edges with the maximum impact are chosen for further relaxation. This heuristic accounts indirectly for the correlation between edges and their vertices in a query graph and relaxes less representative elements first. If several vertices have the same number of data instances or several edges have the same maximum impact, all of them are considered for relaxation.

In Section 5.5, both heuristics are compared against two baselines *full relaxation* and *random relaxation* where the first one produces the whole candidate space and the second one considers only some random elements. The heuristics *maximum impact* and *minimum cardinality* require cardinalities for edges, vertices, and paths, which are provided by the cardinality estimator described in detail in Section 5.2.



(a) Example query

(b) Cardinality maps for vertices and edges

(c) Paths(n) catalogs

Figure 5.2: Example query and its query-dependent statistics

5.2 Cardinality Estimation

The query relaxer and candidate selector require cardinalities of edges, vertices, and paths to decide which query vertices and edges have to be relaxed and to prioritize modified queries, correspondingly. In this sense, both modules rely on the cardinality estimator that calculates or estimates queried cardinalities. To prevent duplicated calculation of the same cardinalities, the cardinality estimator maintains queried statistics and re-uses them if necessary. It also 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.

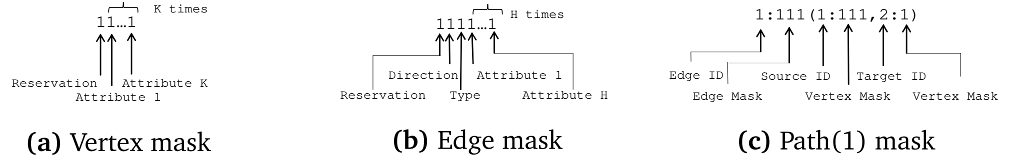
In Section 5.2.1, we start with describing query-dependent statistics that represent the core of the cardinality-estimation module, keep already requested statistics, and provide required data for calculating new statistics. Afterwards, we explain how to query statistics for vertices and edges in Section 5.2.2 and for paths(n) in Section 5.2.3.

5.2.1 Query-Dependent Statistics

The query-dependent statistics (see the example in Figure 5.2) are maintained in a core data structure that stores the original query as presented in Figure 5.2a and several kinds of cardinalities: cardinalities for edges and vertices in the cardinality maps as well as cardinalities for path expressions in the path catalogs as illustrated in Figures 5.2b – 5.2c.

Statistics for Vertices and Edges

The maps for cardinalities of vertices and edges have two levels like presented in Figure 5.2b. On the first level, the key is the identifier of a query vertex or edge, which can be acquired from the query. On the second level, the value of this outer map represents a map itself which entries are pairs of two bit vectors *(a configuration mask, a data bitvector)*. The configuration mask describes which properties of the vertex (edge) have been relaxed. The data bitvector shows which data vertices (edges) match the queried vertex (edge) with this configuration mask.



(a) Vertex mask **(b)** Edge mask **(c)** Path(1) mask

Figure 5.3: Configuration masks for vertices, edges, and paths(1)

The size of a configuration mask is retrieved from the description of the corresponding query vertex (edge) and each of its bits defines a particular property of a vertex (edge). Templates for the configuration masks are illustrated in Figure 5.3. The configuration mask for a vertex comprises a reservation bit and bits for all predicates specified for this vertex in a query as illustrated in Figure 5.3a. The configuration mask for an edge provided in Figure 5.3b consists of at least three elements: a reservation bit, a direction bit, and a type bit. In addition, it can include bits for predicates, if available. The size of a data bit vector equals to the number of vertices (edges) in the data graph. Those bits are set in the data bit vector, whose data vertices (edges) match the configuration mask.

Example Assume the original query and its statistics presented in Figure 5.2. The vertex v_q^1 has two predicates as illustrated in Figure 5.2a, therefore, its configuration mask has in total three bits: two bits for predicates and one reservation bit. The vertex v_q^2 has no predicates and its configuration mask consists of a single reservation bit. In Figure 5.2b, configuration masks and data bitvectors are illustrated for vertex v_q^1 and edge e_q^1 in the cardinality maps.

Statistics for Paths(n)

The query-dependent statistics also keep the cardinalities for paths(n), which can be of different sizes $n \in (1, \mathbb{N})$. Statistics about them are collected in the catalogs for paths up to size n or non-existing paths. The catalog for non-existing (empty) paths(n) in Figure 5.2c is a set of paths, which have no matching data in a data graph and therefore are empty. The catalog for paths with length 1 maps simple paths consisting of single edges with their adjacent vertices to data edge bitvectors. The path(n) catalog keeps mappings between paths(n) and numbers of their instances in a data graph.

The key in a path catalog represents the descriptor of a queried path. A simple path descriptor (for a path of size 1) has three tuples *(an identifier in a query, a configuration mask)*. The first tuple describes a query edge, second and third ones represent its source and target vertices as sketched in Figure 5.3c. In the example in Figure 5.2, the path(1) for edge e_q^1 can be encoded as `1:111(1:110,2:1)` and stored in the path(1) catalog. If a path consists of several hops, its descriptor includes descriptors of all single paths it comprises, which are ascendingly ordered according to their query-edge identifiers.

Statistics Initialization

The query-dependent statistics represent query-specific information, which is used if a query failed and should be refined in order to deliver at least some results. Therefore, they have to be initialized before the relaxation process begins. At the initialization, query-dependent statistics are filled with cardinalities for vertices and edges together with the description of the failed original query. For this purpose, the query manager poses simple cardinality queries to the graph database that describe only single properties for a vertex (edge) and correspond to configuration masks with at most two

Algorithm 8 CALCULATEDATAVECTOR: Calculation of data bit vector for given query vertex

Input: relaxed vertex $v^i_{q_j}$, for which data vector has to be calculated
Output: data bit vector $result$ with set bits for data vertices, which match v^i_q

1: `// 1. Constructing a configuration mask`
2: $v^i_{q_o} \leftarrow$ get vertex from original query(v^i_q)// $v^i_{q_o}.id = v^i_q.id$
3: $queriedMask \leftarrow$ calculate configuration($v^i_{q_o}, v^i_q$)
4: **if** $\exists$ $queriedMask$ **then**
5: $result \leftarrow$ get data bitvector from $vertexMap(v^i_q.id, queriedMask)$
6: **return** $result$
7:
8: `// 2. Extracting matching configuration masks`
9: $masks[\,] \leftarrow$ get all configurations from $vertexMap(v^i_q.id)$
10: **for all** $candidateMask \in masks[\,]$ **do**
11: **for all** $bit \in queriedMask$ **do**
12: **if** $(bit == 0)$ AND $(candidateMask[position(bit)] \neq 0)$ **then**
13: break
14:
15: `// 3. Extracting data bit vectors`
16: **if** bit is last bit **then**
17: $dataVectors[\,] \leftarrow$ insert data bitvector($v^i_q.id, candidateMask$)
18:
19: `// 4. Calculating an output data bit vector`
20: **for all** $dataVector \in dataVectors[\,]$ **do**
21: **if** $dataVector$ is first **then**
22: $result \leftarrow dataVector$
23: **else**
24: $result \mathrel{\&}= dataVector$
25: **return** $result$

set bits (the reservation bit is always set). Responses to such queries describe data bit vectors which set bits indicate matching data vertices (edges). If a vertex without any predicate is requested all bits in the returned data bit vector are set. In such a way, data bit vectors are collected for all query vertices and edges.

Example Assuming the previous example in Figure 5.2, during initialization four queries have to be requested for two vertices in total: three for vertex v^1_q and one for vertex v^2_q. We can omit execution of two queries for masks `100` of vertex v^1_q and `1` of vertex v^2_q, because they request the same data bit vector for all data vertices. This data bit vector can be constructed without querying a graph database by creating a set data bit vector corresponding to the number of vertices in the data graph. Therefore, the number of simple cardinality queries can be derived as $\sum_{i=1}^{|V_q|} |predicates(v^i_q)|$. Cardinalities for edges are collected in the same way like for vertices. The only difference is that two additional kinds of queries are executed to acquire data bit vectors for the type and direction of an edge.

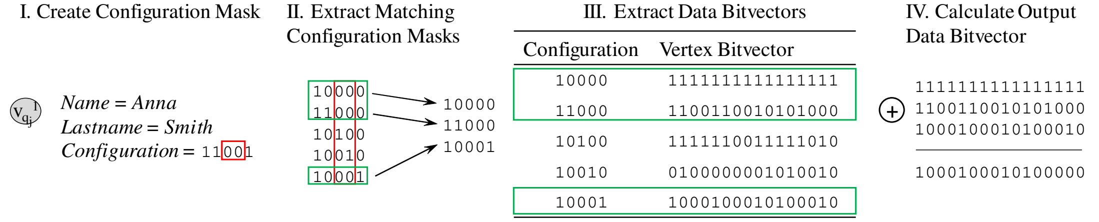


Figure 5.4: Example of vertex bitvector calculation

5.2.2 Querying Statistics for Edges and Vertices

Which statistics for vertices and edges are collected is determined by the used relaxation heuristic described in Section 5.1.2 and priority function of the candidate queue provided in Section 5.3.1.

In general, acquiring cardinalities for edges and vertices implies four steps:

1. *Constructing a configuration mask*. A configuration mask has to be constructed for the queried vertex (edge). The size of the mask is derived from the corresponding vertex (edge) of the original query according to templates in Figure 5.3. In the configuration mask, those bits are set, which correspond to the properties existing in the investigated query element.
2. *Extracting matching configuration masks*. At this step, the cardinality map is traversed for the queried vertex (edge) and the corresponding inner map is extracted. Then the inner map is traversed and those configuration masks are collected which contribute to the queried configuration mask.
3. *Extracting data bit vectors*. At this step, the data bit vectors are collected for the collected configuration masks.
4. *Calculating an output data bit vector*. Finally, the data bit vector is produced for a queried vertex (edge) by combining collected data bit vectors.

The process of discovering the required configuration masks and combining their data bit vectors for a vertex is described in Algorithm 8. First, the statistics controller selects only matching configuration masks based on the unset bits at lines 10 – 17. Second, a bitwise logical AND operation is performed on the data bitvectors for selected configuration masks at lines 20 – 24. Finally, the calculated data bitvector is returned at line 25. This calculation is possible without additional querying the graph database because data bitvectors were collected for each single bit from the configuration mask during the initialization. In a similar way, the data bitvector for an edge can be retrieved by acquiring matching data bitvectors from the edge cardinality map.

Example Assume the input query has a vertex $v_{q_j}^1$ with four properties like presented in Figure 5.4 and its cardinality map is populated with data bit vectors for all properties. After some relaxation, the cardinality estimator receives a cardinality request for the modified version of vertex v_q^1. First, the system identifies its corresponding vertex in the query description and constructs its configuration mask 11001, which has two unset bits corresponding to non-considered properties *type* and *age*. Second, matching configuration masks are extracted from the masks available in its cardinality map based on these two unset bits. Third, the corresponding data bit vectors are retrieved from the statistics. Finally, a bitwise logical AND is performed on the data bit vectors. The result data bitvector in Figure 5.4 shows which data vertices match the queried vertex $v_{q_j}^1$. There are four such data vertices and therefore the queried cardinality equals four.

Algorithm 9 GETPATHCARDINALITY: Algorithm for retrieving path cardinality

Input: path p, which can be of size n
Output: estimated number of data instances, which match p

```
1:  descriptor ← construct path descriptor(p)
2:  if descriptor ∈ catalog for empty paths then
3:      return 0
4:  if p has one step then
5:      if descriptor ∈ path(1)-catalog then
6:          dataMask ← get data bitvector from path(1)-catalog(descriptor)
7:      else
8:          dataMask ← get single path data vector(descriptor)
9:          path(1)-catalog ← insert(descriptor, dataMask)
10:     return count(dataMask)
11: if descriptor ∈ path(n)-catalog then
12:     cardinality ← get cardinality from path(n)-catalog(descriptor)
13: else
14:     cardinality ← estimate path(n) cardinality(descriptor)
15:     path(n)-catalog ← insert(descriptor, cardinality)
16: return cardinality
```

5.2.3 Querying Statistics for Paths(n)

The cardinality of a path is derived according to Algorithm 9. First, at line 1 the descriptor of a queried path is constructed from descriptors of vertices and edges it consists of according to the path template in Figure 5.3. Second, we check whether this path was queried before and delivered empty results at line 2. In this case, it can be found in the catalog for empty paths and the cardinality calculation process can be terminated with the return value of 0. This part of the algorithm is common for paths of any length. If the path has not been found in the catalog for empty paths, remaining catalogs have to be inspected on the containment of this path. For a path(1), at line 5 we check whether it was queried before and calculate its cardinality if necessary. A similar procedure is executed at line 11 for paths with multiple hops.

To resolve a path miss in the statistics, a corresponding data bit vector $result$ for the path p has to be estimated based on the already collected information. The estimation of the path(1) cardinality is exact and described in Algorithm 10. For this purpose, the data bit vectors for its edge, source, and target are extracted at lines 2 – 7 by Algorithm 8. Afterwards, for each set bit in the edge bit vector, the corresponding entry $(sourceId, targetId)$ from the vertex-edge mapping is checked on the existence in the extracted vertex bit vectors at lines 10 – 14 and in case of non-existence, the corresponding bits of the edge mask are reset. The final bit vector is returned at line 15.

Example Assume the system acquired data bit vectors for an edge and its vertices like presented in Figure 5.5a. To derive the path(1) for the given edge and its vertices, the vertex-edge mapping is filtered according to Algorithm 10. The entries in the vertex-edge mapping are pairs of source and target positions in the vertex bitvector. The algorithm checks the set bits 1, 3, and 5 in the edge bitvector. The bits 3 and 5 are reset, because the source of the third edge and the target of the fifth edge do not exist in source and target data bit vectors, correspondingly. The final path(1) has only one set bit and therefore its cardinality equals one.

Algorithm 10 GETSINGLEPATHDATAVECTOR: Algorithm for calculating data bit vector for path(1)

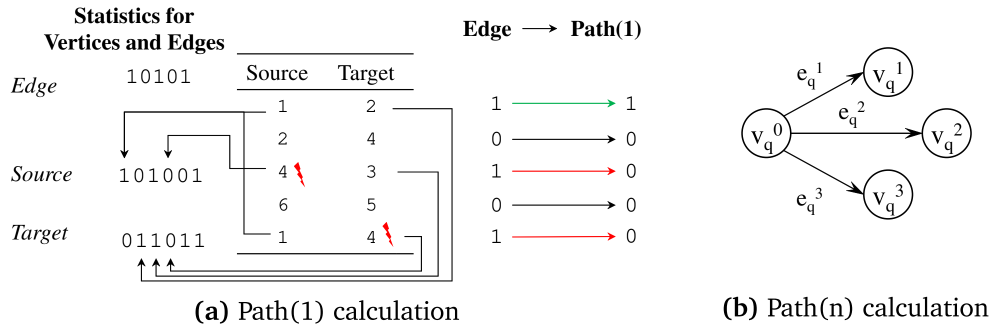
Input: path p of length 1
Output: data bitvector $result$ for edges, which match p with their vertices

```
1: // 1. Extracting independent statistics
2: edge ← extract edge from path(p)
3: result ← calculate data bit vector for edge
4: source ← extract source from path(p)
5: sourceDataMask ← calculate data bit vector for source
6: target ← extract target from path(target)
7: targetDataMask ← calculate data bit vector for target
8:
9: // 2. Filtering out edges with non-matching vertices
10: for all set bits ∈ dataMask do
11:     sourceId ← get source identifier from mapping for bit
12:     targetId ← get target identifier from mapping for bit
13:     if sourceId ∉ sourceDataMask OR targetId ∉ targetDataMask then
14:         unset bit in result
15: return result
```

The cardinality of a path(n) is estimated from the n paths(1), which it consists of, in four steps:

1. *Encoding the data.* At this step, each path(1) from path(n) is encoded in a sparse matrix in the coordinate format, which consists of three fields: identifiers of source and target vertices as well as the number of data edges between these two data vertices. To construct these matrices, first, data bit vectors for each path(1) are retrieved by Algorithm 10. Second, their corresponding pairs of sources and targets are extracted from the vertex-edge mapping. If a single path is specified as undirected then edges have to be retrieved in the backward direction as well, which is done by transposing the entries and adding to the directed edges. As a result, each path(1) is described as a set of source-target pairs. Finally, the number of edges is summed up between the same source and target vertices.
2. *Establishing a traversal order.* At this step, we have to choose a traversal order for the path. As the starting point, a vertex without incoming or outgoing connections is used. Then, we traverse the path as an undirected graph in a depth-first manner and memorize the traversal order.
3. *Multiplying the matrices.* After the traversal order is prepared, the corresponding matrices can be multiplied along it. If an edge is traversed in the backward direction, its matrix has to be transposed before multiplication.
4. *Estimating a cardinality.* The cardinality of a path is estimated as a sum of all non-zero elements of the derived matrix.

Example Assume the example in Figure 5.5b, where the query describes a vertex connected with three other vertices. To calculate the cardinality of the path(2) including edges e_q^1, e_q^2, first, sparse matrices for both edges are prepared in the coordinate format. Next, a vertex with the minimal degree is chosen as the start of the path. Assume vertex v_q^1 with an incoming edge e_q^1 to be a starting edge. To traverse backward from vertex v_q^1 to vertex v_q^0, the sparse matrix for edge e_q^1 has to be transposed. Vertex v_q^0 in path e_q^1, e_q^2 has two outgoing edges: already processed edge e_q^1 and non-processed edge e_q^2. Next, edge e_q^2 is traversed to its target vertex v_q^2. The direction of this edge



(a) Path(1) calculation

(b) Path(n) calculation

Figure 5.5: Examples for path calculations

corresponds to the direction of the traversal. Therefore, its sparse matrix is used as it is and, finally, both matrices are multiplied. The sum of all non-zero elements of the produced matrix is the estimated cardinality of path(2) e_q^1, e_q^2.

To summarize, in this section we present how to derive cardinalities for vertices, edges, and paths, which is based on constructing a configuration mask and calculating a data bit vector for it. The derived statistics are extensively used among different components of the why-empty system such as the query relaxer and candidate selector. In the following section, we will discuss how the candidate selector makes use of cardinalities from the query-dependent statistics.

5.3 Query-Candidate Selection

The above presented query-dependent statistics are used to prioritize query candidates and to select the most promising ones to be checked on the delivery of non-empty results. Received query candidates are ordered dependent on a ranking function and stored in the priority queue that represents an ordered list of rewritten queries, where more promising candidates are located at the beginning of the queue.

5.3.1 Placement of New Query Candidates

To insert a new query candidate in the queue, the exact position has to be derived for it. For this purpose, the new candidate is compared with query candidates from the queue from head to tail according to two criteria like illustrated in Figure 5.6. The first criterion compares average path cardinalities of both queries and sorts them in a descending order. It requires parameter n that describes the maximal path length, which can be used for comparison. To calculate the compared values, both queries are split in sets of paths of length n if possible. If there is no path with length n at least in one query, paths with a smaller length are acquired. Afterwards, for each path of each query, its cardinality is requested and the average path cardinalities are calculated for them. If both queries have equal average-path cardinalities, the path length is decremented and they are compared again. The last available path length is 1, which corresponds to a path(1). In case the average path(1) cardinalities are equal the queries are ordered according to the second criterion.

The second criterion considers the relative cardinality change induced by applied relaxations that 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

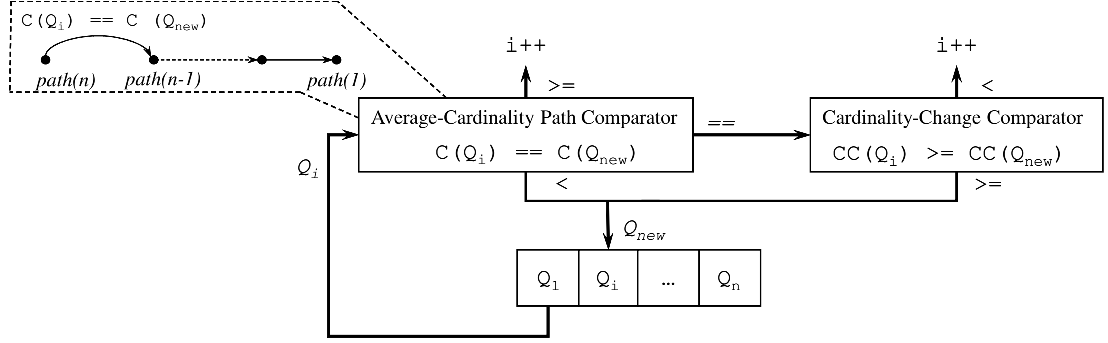


Figure 5.6: Placement of new queries in priority queue

lower induced cardinality change is placed ahead in the queue. In Section 5.3.2, the calculation of the induced cardinality change is discussed in detail.

To summarize, the first criterion chooses those changes, which promise to increase the result cardinality, the second one keeps the changes minimal by preventing strong cardinality modifications. From them, the induced cardinality change is a stronger comparator, because it leads to a more optimal solution by passing all possible relaxations. Although it may provide better results with a limited cardinality change, it can increase time for discovery of a non-failed query strongly. Based on these observations, induced cardinality changes are compared at the second step. If it would be executed on the first place, the changes to the original query would be too small and this would increase the number of relaxation iterations and, as a consequence, the response time.

5.3.2 Calculation of Induced Cardinality Changes

To place a new candidate in the queue, relative cardinality changes have to be calculated, which are induced in the query candidate by the relaxation. The term *induced cardinality change* describes a maximum possible relative cardinality change caused by a relaxation of a specific query element. Induced cardinality changes can be calculated for each query constraint and relaxation operation, which can change it. For example, for edge e_q^i we can calculate cardinality changes for deletion of its type, direction, each predicate, and edge deletion itself.

The hierarchy of induced cardinality changes is represented in Figure 5.7, which consists of four levels: (I) full relaxation, (II) attribute and topological relaxations, (III) an operational level, and (IV) an instance level. If all elements have been relaxed and a new query candidate is empty, this corresponds to a full relaxation, which is characterized by a total induced cardinality change. This measure considers all possible relaxations which can be applied to a query and aggregates their induced cardinality changes. The total cardinality change corresponding to a full relaxation equals 1. We also can distinguish lower levels for aggregating cardinality changes such as attribute and topological relaxations (level II) or operational level (level III). The instance level (level IV) represents induced cardinality changes of relaxation operators which are instantiated for specific query elements and their properties.

In general, all cardinality changes on the instance level are dependent on each other. Any relaxation can modify the cardinality changes of other elements as well, which leads to a complexity of $\mathcal{O}(m_q \cdot n_q \cdot k)$ for calculating the cardinality change, where n_q, m_q are numbers of vertices and edges in a query graph, respectively, and k is a number of candidates. We consider the changes to be independent and thus reduce the complexity to $\mathcal{O}(n_q + m_q)$ and consider the maximum possible cardinality changes.

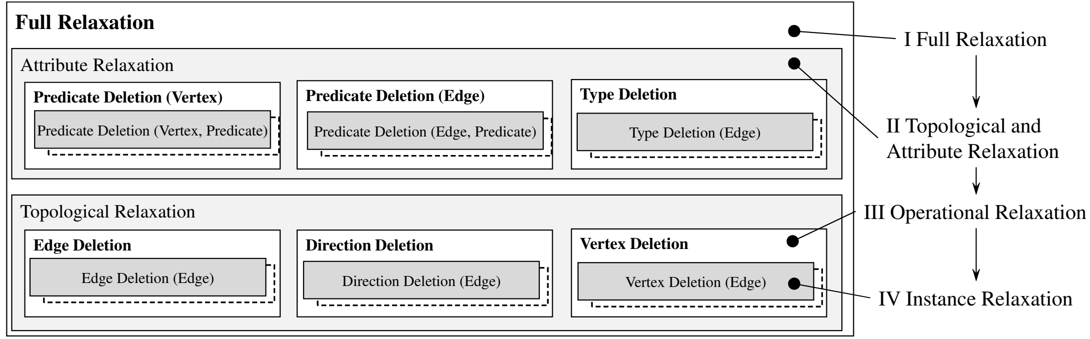


Figure 5.7: Hierarchy of induced cardinality changes

Cardinality changes are calculated in two steps before the relaxation takes place. First, the cardinality changes are calculated on the instance level and summed up. The derived cardinality value describes then the full relaxation, which equals 1. Therefore, as second, the instance-level cardinality changes are scaled by the value for a full relaxation.

The cardinality changes on the instance levels are derived as the upper bounds for the relative cardinality changes of their elements or elements' neighbors caused by relaxation operations without considering any correlation between graph elements and their predicates. Cardinality changes for operations on attributes consider only a modified element, while cardinality changes for topological operations can be derived from the neighboring elements.

We start the description of calculating cardinality changes with relaxation operations on predicates. The predicate deletion $PD(v_q^i, a_q^k)$ of predicate a_q^k for vertex v_q^i is a cardinality change of vertex v_q^i caused by this relaxation:

$$PD(v_q^i, a_q^k) = \frac{C(V_d) - C(V_d|a_q^k)}{C(V_d)} = \frac{\Delta C(V_d, a_q^k)}{N_d} \tag{5.1}$$

Similarly, the cardinality change for predicate deletion $PD(e_q^j, a_q^h)$ of predicate a_q^h for edge e_q^j is derived as

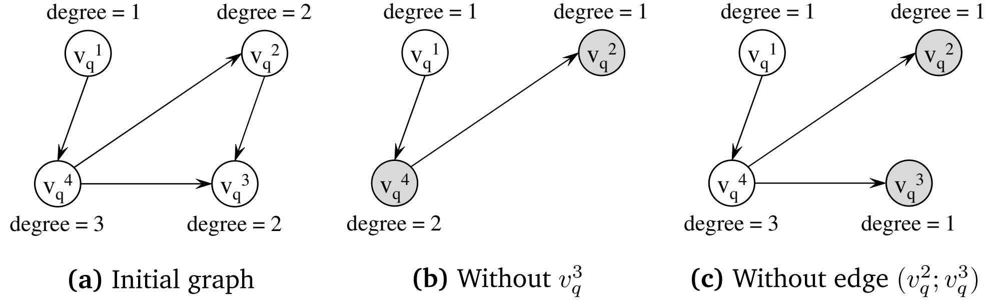
$$PD(e_q^j, a_q^h) = \frac{C(E_d) - C(E_d|a_q^h)}{C(E_d)} = \frac{\Delta C(E_d, a_q^h)}{M_d} \tag{5.2}$$

The similar equation is used for type deletion $TD(e_q^j)$ of edge e_q^j:

$$TD(e_q^j) = \frac{C(E_d) - C(E_d|T)}{C(E_d)} = \frac{\Delta C(E_d, T)}{M_d} \tag{5.3}$$

In contrast to relaxation operations on attributes, topological relaxations modify the query structure. Topological operations are represented by a vertex deletion, an edge deletion, and a direction deletion. While the influence on predicates can be expressed by the relative cardinality of the relaxed elements themselves, topological changes for a vertex and an edge deletion are difficult to express, because if the elements are removed, their cardinalities cannot be calculated. Therefore, topological modifications are interpreted as cardinality changes of direct neighbors of removed elements. The change for a direction deletion is expressed by the cardinality change of a relaxed edge. A direction deletion $DD(e_q^j)$ of an edge e_q^j increases its cardinality twice, because a relaxed edge can be considered during graph traversal in both directions.

$$DD(e_q^j) = \frac{C(E_d) - C(E_d|D)}{C(E_d)} = \frac{\Delta C(e_q^j, D)}{M_d} = 0.5 \tag{5.4}$$



(a) Initial graph **(b)** Without v_q^3 **(c)** Without edge $(v_q^2; v_q^3)$

Figure 5.8: Influence of vertex deletion on degrees of neighboring vertices

By deleting a vertex or an edge, the cardinalities of neighboring vertices can change, which can be expressed through the number of data vertices having a specific degree.

Example Assume a graph query as presented in Figure 5.8a. There are four vertices with a degree $\deg \in [1; 3]$. The deletion of vertex v_q^3 also removes its adjacent edges. Therefore, the degrees of direct neighbors v_q^2, v_q^4 are reduced by the number of removed adjacent edges like presented in Figure 5.8b. If the edge between vertices v_q^2, v_q^3 is deleted their degrees reduce as well (see Figure 5.8c).

If an edge e_q^j is deleted by the edge deletion operation $ED(e_q^j)$ the degrees of its source and target decrease. Therefore, its change is described as a cardinality change of an edge's ends derived from the change of the number of instances with a specific degree.

$$ED(e_q^j) = \frac{C(V_d|deg(src(e_q^j)) - 1) - C(V_d|deg(src(e_q^j)))}{C(V_d|deg(src(e_q^j)) - 1)} + \frac{C(V_d|deg(tgt(e_q^j)) - 1) - C(V_d|deg(tgt(e_q^j)))}{C(V_d|deg(tgt(e_q^j)) - 1)} \tag{5.5}$$

$$VD(v_q^i) = \sum_{p=0}^{N_{adj}} \frac{C(V_d|deg(v_p) - 1) - C(V_d|deg(v_p))}{C(V_d|deg(v_p) - 1)} \tag{5.6}$$

where $C(V_d|deg(v_p)-1), C(V_d|deg(v_p))$ is a number of data vertices with $deg \geq deg(v_p)-1$ or $deg \geq deg(v_p)$, respectively.

After cardinality changes on the instance level are calculated they have to be scaled by the full relaxation. The total cardinality change that corresponds to full relaxation equals 100%. Therefore, the calculated cardinality changes have to be normalized to this value with $\alpha = \frac{change(total)}{100}$.

Application to Different Use Cases The proposed induced cardinality change is general and not limited to any specific use case. To adapt it to a particular use case and to express the influence of specific relaxations on the final cardinality change, the normalization weight can be calculated at different levels: at the level of all attribute or topological operations or even further at the level of specific operations. This thesis focuses on property graphs, for which the attribute and topological descriptions are equally important and, thereby, both total predicate and topological changes have the same weight $\alpha = 50\%$.

Assume a corporate database including information about company partners, business processes, etc., which is filled with additional information about data flows in the company extracted from internal text documents. The quality of a created graph depends

strongly on the extractors and may be not fully correct. Based on the quality of extractors, a user can define different weights for attribute and topological parts. In this example, the topology is created from unreliable data and includes mistakes. Therefore, topological aggregation can be annotated by a smaller weight $\alpha \ll 100\%$, which would describe how much this information is trusted.

To summarize, in this section we discussed how to prioritize refined queries in such a way that the most promising candidates to deliver non-empty results are processed first. For this purpose, we introduced two criteria based on calculations for average path cardinalities and induced cardinality changes. The first one characterizes a query itself, while the second one describes applied relaxations. Until now, we presented all components of the why-empty system, which allow to conduct automatic query relaxation without considering a user feedback. In the following section, we will discuss how to improve the relaxation process with non-intrusive user integration.

5.4 User Integration

The why-empty system takes its decisions based on several heuristics, which allow to reduce the search space by considering only the most promising changes. This is an automatic relaxation process without considering user interest in rewriting decisions. Although this approach can lead to the early discovery of a successful query candidate, it may ignore sensitive information from the query and deliver non-interesting relaxed queries. To prevent deletion of sensitive information from the query, user interest in specific query elements should be incorporated in the relaxation process.

A straightforward approach may integrate a user in each relaxation iteration. In this case, the system generates several query candidates and proposes all of them to the user who chooses one candidate for further investigation. If a selected candidate fails to deliver a non-empty result, it is relaxed again in multiple ways and a new set of candidates is proposed to the user. The user is completely integrated into the relaxation process, all critical decisions are taken by him. This approach is called navigational [108]. It can lead to a high number of iterations and therefore frustrates the user. The navigational approach can be improved by predicting user interest in some query parts based on domain knowledge or querying history. This means that such query candidates can be first proposed for ranking, which are assumed to be more relevant. If user interest cannot be predicted, such an improvement is impossible.

Considering drawbacks of the navigational approach and necessity of user integration, we aim at combining the advantages of automatic and navigational methods and propose a semi-automatic approach for integrating a user in the relaxation process. The main idea is to integrate the user non-intrusively by allowing to rank only already discovered non-failed queries. While the navigational approach provides query candidates to the user, which are not tested yet on the delivery of non-empty results, our semi-automatic method suggests already evaluated query candidates called query solutions that deliver non-empty results. The why-empty system automatically discovers query solutions and proposes them to the user, who can either reject or accept them. Based on the provided rating, the system constructs a user model and adapts the rewriting process accordingly.

Rejecting Changes The rejection of a solution can mean that this solution is probably over-relaxed and includes some unfavorable changes. From this information, the why-empty system can conclude which changes are not likely to be preferred by the user and try to postpone their relaxation to a later point in time. Therefore,

rejection of a solution reduces the search space by discarding relaxation branches below the rejected query candidates and prioritizes the changes.

Accepting Changes Similar to the rejection, a user can choose queries which are interesting to him. If a query is accepted by the user, it includes information which is critical for him. In addition, relaxations performed in accepted queries are tolerated by the user and can be prioritized during the relaxation. If a user accepts a solution then the search can be either terminated or continued dependent on whether a user is interested in discovery of a better solution with less changes.

5.4.1 User-Preference Model

From the above presented acceptance and rejection of query solutions, a user-preference model can be derived to prioritize specific query elements. This model consists of identifier-preference pairs, where *identifier* is a vertex or edge query identifier in the original query graph and *preference score* is a real number which describes the importance of this element to the user.

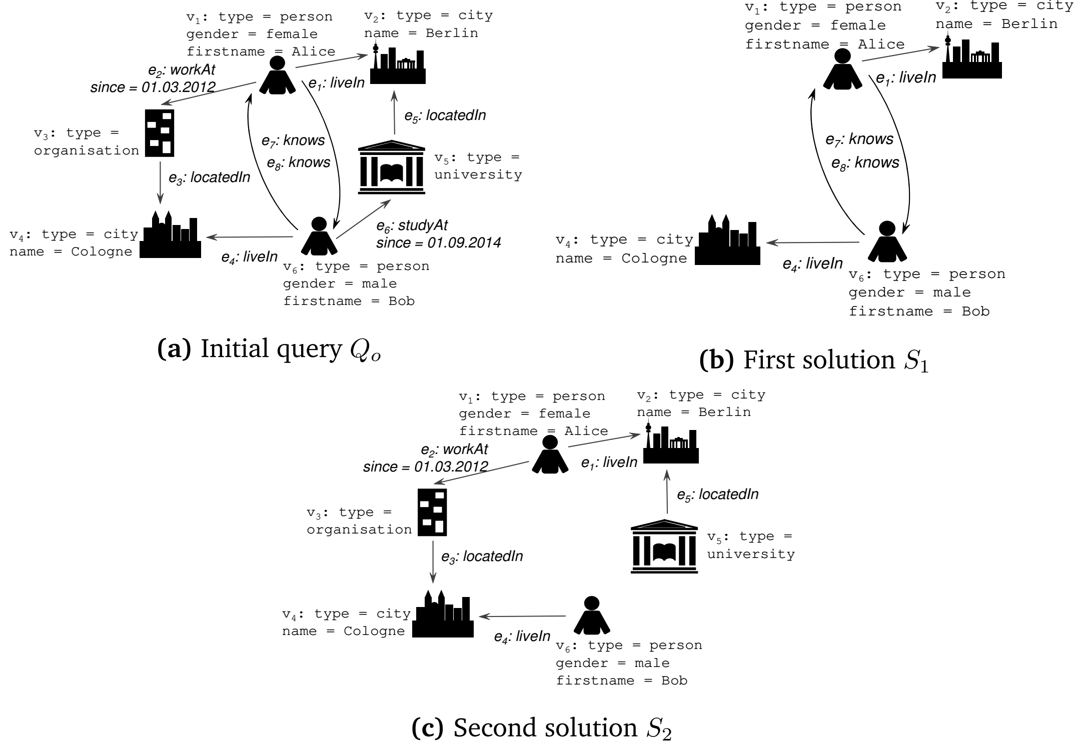
The preference scores can be positive, negative, or neutral. The relaxation of elements with lowers scores is more easily tolerated by the user than the modification of elements with higher scores. The neutral value ($score = 0$) means that the user interest in this element is unknown.

At initialization, all pairs have neutral scores. With the appearance of new user ratings, the model is adapted. For each rated solution, its user-preference model is created, which is further integrated in the overall user-preference model.

Model Construction For Accepted Solutions Assume the query Q_o consists of N vertices and M edges and delivers an empty answer and assume it has been relaxed in such a way that its solution S_1 with $N - K$ vertices and $M - L$ edges delivers a non-empty result. A user accepts it and continues the search for a better solution. Based on this user rating, the K removed vertices and the L removed edges are considered to be tolerated changes and therefore each of them is rated by the negative score -1. The rest of the graph is considered to be unchanged and should be kept during the search. As a result, $N - K$ vertices and $M - L$ edges are rated by the positive score $+1$.

Model Construction For Rejected Solutions Assume the same query Q_o and its second solution S_2 with $N - P$ vertices and $M - T$ edges delivers a non-empty result. A user rejects it and continues the search for a better solution. Any rejection shows non-preferred changes. Therefore, the P vertices and T edges are rated by the system with the positive score $+1$ showing that they have to be kept without changes. The rest of the elements existing in S_2 is difficult to rate: they can be important or irrelevant to a user. Therefore, they are set to be unknown and annotated by the neutral score 0. After a user-preference model is constructed for a solution, it can be integrated in the overall user-preference model. For this, the preference scores for the same vertices and edges are aggregated. As a consequence, new levels apart -1, 0, +1 can arise, which describe stronger acceptance or rejection of relaxations.

Example Assume a user searches for data subgraphs with Alice and Bob, who know each other, live in Berlin and Cologne, respectively (see Figure 5.9a). Alice works in Cologne, while Bob studies at a university in Berlin. The why-empty system automatically detected two solutions S_1, S_2 (see Figure 5.9b – 5.9c), which return non-empty results. A user accepts the first proposal S_1 and rejects the second one S_2. Therefore,



(a) Initial query Q_o

(b) First solution S_1

(c) Second solution S_2

Figure 5.9: Example query delivering empty result and its two explanations

Action	e_1	e_2	e_3	e_4	e_5	e_6	e_7	e_8	v_1	v_2	v_3	v_4	v_5	v_6
Initialize	0	0	0	0	0	0	0	0	0	0	0	0	0	0
Accept S_1	+1	-1	-1	+1	-1	-1	+1	+1	+1	+1	-1	+1	-1	+1
Reject S_2	0	0	0	0	0	+1	+1	+1	0	0	0	0	0	0
Model	+1	-1	-1	+1	-1	0	+2	+2	+1	+1	-1	+1	-1	+1

Table 5.1: User-preference model derived from ratings of two solutions S_1 and S_2

the S_1 mode includes positive and negative constraints and the S_2 model consists of positive and neutral elements (see Table 5.1). The overall user model derived from the combination of the initial ratings and S_1, S_2 models introduces the new level $+2$, which highlights the importance of the friend relationship to the user.

5.4.2 Adaptation of Query Rewriting

Considering the why-empty system architecture introduced in Figure 5.1, the user-preference model can be integrated in the candidate selector or query relaxer.

Adapting the Candidate Selector To integrate the user-preference model in the candidate selector, its priority function has to be adapted. Originally, the candidate selector of the why-empty system aims at prioritization of query candidates which are likely to produce non-empty answers and considers average path cardinality and induced cardinality changes as presented in Section 5.3. The first ranking criterion focuses strongly on cardinalities. Therefore, the only way to

influence the prioritization of query candidates is to modify the calculation of cardinality changes. By integrating user preferences, the cardinality changes for graph elements with positive weights from the user model are annotated with higher weights. Processing of query candidates with higher cardinality changes are postponed according to the priority function. After the annotation, all cardinality changes are recalculated and all query candidates in the queue have to be resorted according to recalculated values. This means that each time a user provides feedback the cardinality changes have to be updated and query candidates have to be re-ordered in the priority queue. However, the cardinality change corresponds to the second ranking criterion, which is rarely considered. Most of the query candidates are filtered out already by the first criterion and therefore only a limited number of query candidates are checked according to the second one. As a consequence, multiple low-relevant query candidates will not be filtered out. Therefore, this integration options is less preferable.

Adapting the Query Relaxation To generate query candidates, the query relaxer requires a heuristic to select query vertices and edges for modification. Therefore, we propose a feedback-based heuristic, which considers the user-preference model. According to it, elements with lower preference scores are selected first for the relaxation. Assume a query candidate Q_i and user-preference model P. First, the system extracts the lowest preference score p_{min} and all query edges and vertices $\in Q_i$ for this score p_{min}. Second, the query relaxation component generates new query candidates according to them. If there is no $e(p_{min}), v(p_{min}) \in Q_i$ or no new query candidates were generated, the next lowest preference score is extracted from the model and the generation is repeated. This recursion terminates if some new candidates were generated or no higher preference scores exist. Referring to the example in Figure 5.9 and the user integration model in Table 5.1, first the elements with $p_{min} = -1$, then with $p_{min} = 0$ are relaxed.

Having integrated user interest, the why-empty engine works 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 candidate queue. The best candidate is extracted from the queue and tested on the delivery of any answer. After a query explanation with a non-empty result is found the user can rate 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 relaxation 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.

5.5 Evaluation

In this section, the proposed why-empty engine is evaluated that provides refined query candidates delivering at least some results as modification-based explanations. We test the four LDBC and five DBPEDIA failed queries in all experiments, which are described in Appendix A and whose original cardinalities equal 0. The same queries have already been used for evaluating the generation of subgraph-based explanations for why-empty queries discussed in Section 4.5.

In this evaluation, we mainly focus on the following questions:

- Does the why-empty system discover refined queries delivering non-empty results?
- Which priority function and relaxation heuristic allow to discover solutions faster and of better quality?
- How much resources do query-dependent statistics consume?

To answer these questions, we consider the syntactic distance as a quality measure, which is described in Section 3.2.2 and show how a refined query differs in description from an original query. As a performance measure, the number of relaxation iterations is used, which corresponds to the number of executed refined queries. We normalize it by the maximum number of iterations for a specific experiment per query.

In the following, we first evaluate different functions to order query candidates in the priority queue of the query selector and choose the top-3 functions from them, which are used for comparing relaxation heuristics in Section 5.5.1. In Section 5.5.2, we consider the convergence of the best-performed priority function for first five discovered explanations. In Section 5.5.3, we discuss resource consumption for maintaining statistics and specifics of the best performing priority function. Finally, we evaluate integration of a user-preference model in the relaxation process in Section 5.5.4.

5.5.1 Priority Functions of Query-Candidate Selector

In the first set of experiments, we consider different priority functions, which can be used by the query-candidate selector to prioritize query candidates in the candidate queue. We compare them against the priority function proposed in this thesis in Section 5.3 that considers two criteria, i.e., an average path(1) cardinality and induced cardinality change.

We conduct this experiment for all nine queries and four relaxation heuristics such as full relaxation, random, maximum impact, and minimum cardinality, where the two last heuristics are described in Section 5.1.2. Each experiment is terminated when the first explanation is found or the time limit of two minutes is exceeded. If no solution has been discovered, the priority function is annotated with a syntactic distance of 1, and the maximum number of iterations per query. The goal of this experiment is to discover the best three priority strategies for the further evaluation in this section. As follow, we describe the evaluated priority functions.

Evaluated Priority Functions

For this evaluation, we generated 287 priority functions by combining some of seven criteria including an induced cardinality change, a number of query vertices or edges, average vertex or edge cardinality, average, or minimal path(1) cardinality. These criteria characterize the size of queries and cardinalities of specific query elements. We generate priority functions in such a way that a function may include either average or minimal path(1) cardinality, but not both of them.

In this evaluation, we consider two general ways for comparing two queries: rule- and sum-based ones. In the first case, constraints of a priority function are evaluated sequentially until one of the queries dominates another one. In the second case, rewritten queries are compared according to all criteria and a total comparison score is used for sorting the queries. The comparison of two queries according to a single criterion derives a score, which equals 1 if the first query dominates, 0 if both queries are equal,

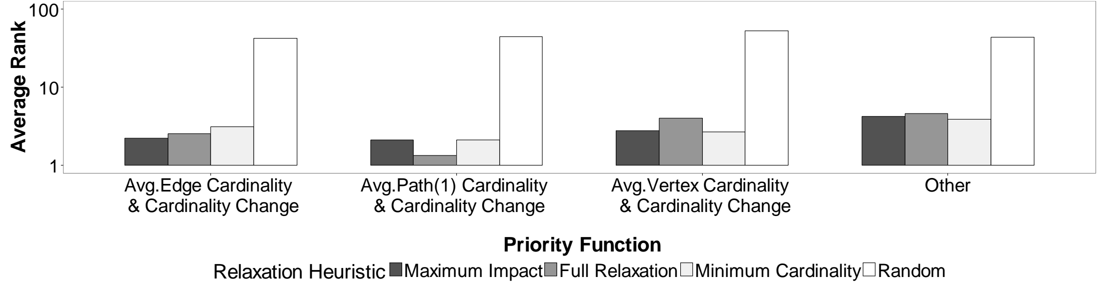


Figure 5.10: Average rank for priority functions

and -1 if the second query dominates. The scores for all evaluated criteria are aggregated in a total score used for the final prioritization. A positive score means that the first query dominates over the second one and should be placed ahead in the queue.

In total, we generated 287 priority functions, which are defined as follows:

- 7 single-criterion functions,
- 12 rule-based functions, where each of them considers an induced cardinality change and one of the six remaining criteria from the list,
- 120 rule-based functions with three criteria, where $k = 3, n = 6$:

$$\frac{n!}{(n-k)!} = 120,$$

- 60 rule-based functions with three criteria, where $k = 3, n = 6$ and one of the criteria is the minimal path(1) cardinality:

$$k \cdot (\frac{(n-1)!}{((n-1)-(k-1))!}) = 60,$$

- 57 sum-based functions with the number of criteria varying from two up to six from the list, where $n = 6$:

$$\sum_{k=2}^{n} \frac{n!}{(n-k)! \cdot k!} = 15 + 20 + 15 + 6 + 1 = 57,$$

- 31 sum-based functions with the number of criteria varying from two up to six from the list, where $n = 6$ and one of the criteria is the minimal path(1) cardinality:

$$\sum_{k=2}^{n} \frac{(n-1)!}{((n-1)-(k-1))! \cdot (k-1)!} = 5 + 10 + 10 + 5 + 1 = 31.$$

Average Ranks of Top-3 Priority Functions

For comparing priority strategies according to the performance and quality measures, we calculate a ranking score as an average of the syntactic distance and the number of relaxation iterations. In order to compare both measures between different queries, they are normalized per query Q for each priority function $priority$ according to the feature scaling as follows:

$$distance'(priority, Q) = \frac{distance(priority, Q) - distance_{min}(Q)}{distance_{max}(Q) - distance_{min}(Q)}$$

$$iterations'(priority, Q) = \frac{iterations(priority, Q) - iterations_{min}(Q)}{iterations_{max}(Q) - iterations_{min}(Q)}$$

We calculate ranking scores for all priority functions and order them ascendingly such that priority functions with the same ranking score occupy the same position in the ranking list. Afterwards, we extract the rank for each function as its position in this list and select three best-performed priority functions that consider two criteria in a rule-based manner. Each selected function inspects the induced cardinality change as the second criterion and average vertex, edge, or path(1) cardinality as the first criterion.

The experimental results for selecting top-3 evaluated priority functions are illustrated in Figure 5.10, where the y axis defines the average rank among the queries and the x axis describes priority functions. We aggregate ranks dependent on used relaxation heuristics, which are illustrated by different colors. The best rank is provided by the priority function that considers average path(1) cardinality as the first criterion. This function outperforms other best functions by two to three times. Although we evaluate 287 functions, the average rank is rather small. Referring to the evaluated functions, most of them consider three comparison conditions. From this observation, we can assume that multiple priority functions have the same rank and two ranking criteria are already enough to prioritize refined queries.

Considering relaxation heuristics which are illustrated by different colors in Figure 5.10, we can conclude that random relaxation should not be used because none of the evaluated priority functions clearly dominates over other functions. Therefore, we do not evaluate it in the following experiments. It also can be seen that the influence of maximum-impact, minimum-cardinality, and full relaxations on the relaxation process is weaker than the influence of priority functions. Although the full relaxation shows the best results for average path(1) cardinality it is unstable among other best-performed functions. This means that the priority function with the average path(1) cardinality compensates the disadvantages of the large search space induced by the full relaxation, which cannot be easily done by another priority functions. Maximum-impact and minimum-cardinality heuristics show the most stable behavior among different priority functions.

To summarize, the priority function that considers average path(1) cardinality with the induced cardinality change derives the best rank among all evaluated queries, which is two to three times better than the second and third best priority functions. A priority function has a higher influence on the relaxation process than a relaxation heuristic.

Quality and Performance of Top-3 Priority Functions Among Queries

Calculating the rank for priority functions helps us to judge which functions perform best. However, it hides the quality of produced explanations and the performance of the relaxation process from us. Therefore, in the following we present results for these metrics.

In Figure 5.11, the performance and quality measures are provided for all evaluated queries, where the y axis represents the normalized number of iterations as a performance measure and the x axis describes the normalized syntactic distance of discovered explanations as a quality measure. Relaxation heuristics are distinguished by different colors. The shape of a point defines a priority function. In addition to using the shapes, we increase the size of the points for top-3 priority functions.

According to the used relaxation strategies, full relaxation typically requires a high number of iterations, which can be seen from the red points in the upper right corner. In the limited amount of time, only a small fraction of experiments delivers queries with

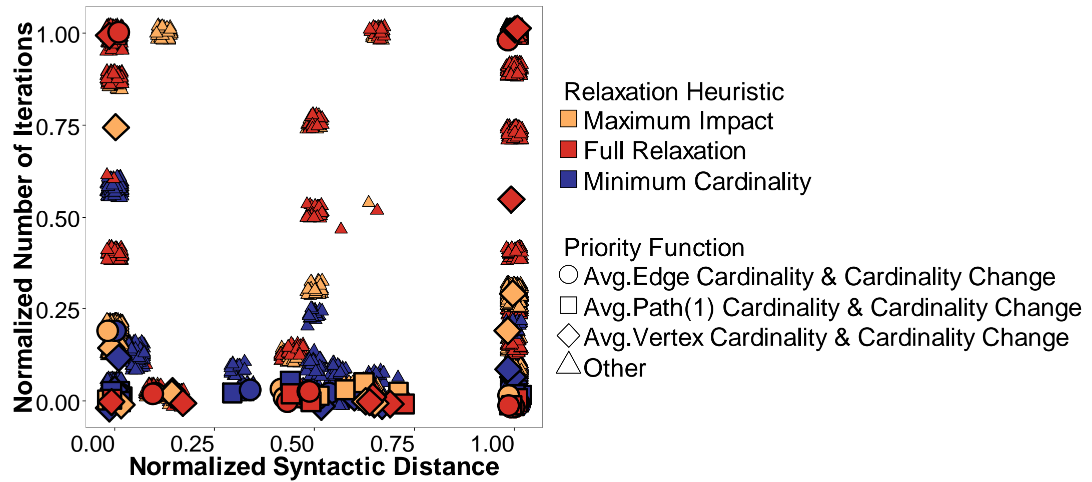


Figure 5.11: Evaluation results for different priority strategies across evaluated queries

non-empty results. This explains the poor behavior of full relaxation for remaining priority functions.

In this chart, we can distinguish about 30 groups of points, which correspond in most cases to the same relaxation heuristic and to those priority functions which behave very similarly. Referring to our previous discussion about the ranking in Figure 5.10, the existence of these groups supports our conclusion for a low value of the average rank. The establishment of these groups can be explained by characteristics of used priority functions, namely: 180 out of 287 functions compare queries in a rule-based manner according to three criteria. In many cases, the third criterion is not evaluated and therefore two criteria are already enough for comparison.

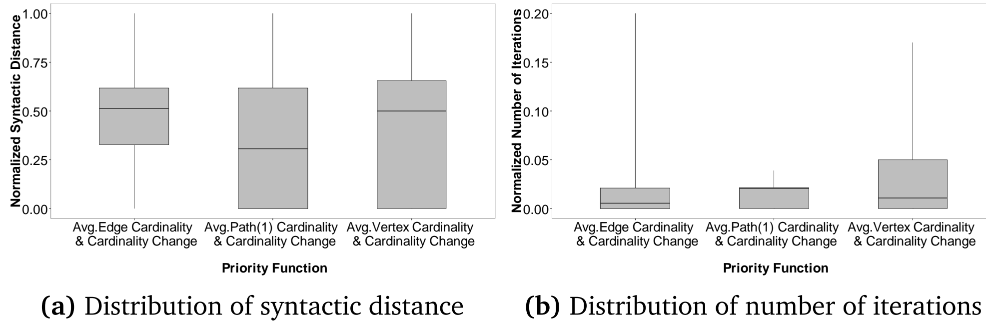
In the chart, we also can see that in most cases the discovered solutions have a normalized syntactic distance at around 0.6. Only a small part of experiments can deliver queries of better quality, which is less than 0.25, which can be explained by existence only of a few queries, which do not require strong modifications in order to deliver some results.

For most queries, points for the top-3 functions lie in the low part of the figure. The normalized number of iterations is more than 0.5 only for six points for the top-3 priority functions. Five out of six points correspond to the full relaxation and four out of six are produced by the third best priority function that considers average vertex cardinalities.

To summarize, the full relaxation requires a high number of iterations in order to deliver explanations with non-empty results. Therefore, in the limited amount of time it may not always produce non-failed queries. In most cases, top-3 priority strategies deliver non-failed queries quickly at the cost of their reduced quality expressed by high syntactic distances. In the following, we will analyze the distributions for the quality and performance metrics of the top-3 priority functions.

Distributions of Quality and Performance Measures for Top-3 Priority Functions

Distributions of the normalized syntactic distances and normalized numbers of iterations are illustrated in Figure 5.12 for the top-3 priority functions and the minimum-cardinality relaxation strategy. In Figure 5.12a, the best distribution for the syntactic distance is derived by priority function with the average path(1) cardinality: it has the lowest median.



(a) Distribution of syntactic distance **(b)** Distribution of number of iterations

Figure 5.12: Evaluation results for top-3 priority functions

Considering the normalized number of iterations in Figure 5.12b, we can conclude that the best performance is also provided by the best-ranked priority function that considers the average path(1) cardinality. This priority function prefers stronger changes and therefore provides the fastest answer.

To conclude, the best combination of quality and performance can be achieved by the top-ranked priority function that considers the average path(1) cardinality and induced cardinality change in combination with minimum-cardinality relaxation heuristic. This strategy also provides the most stable behavior in terms of quality and performance in comparison with other top-ranked functions.

To summarize, in this evaluation, we test 287 different priority functions, which can be used to order queries in the query-candidate queue. In addition, four relaxation heuristics have been evaluated such as full relaxation, random, maximum impact, and minimum cardinality. According to the results, the most promising priority functions consider consequently cardinality-based properties like average cardinalities for vertices, edges, or paths(1), and induced cardinality changes of rewritten queries. The best ranking is exhibited by the strategy with average path(1) cardinality, which outperforms other top-ranked functions by the factor of two to three. From the priority functions and relaxation heuristics, the first ones affect the rewriting process stronger and can compensate a poor behavior of relaxation strategies, if required. The best combination of quality and performance is achieved by the priority function with the average path(1) cardinality and minimum-cardinality relaxation strategy.

5.5.2 Runtime Convergence

In this set of experiments, we check whether better explanations with lower syntactic distances can be delivered right after the discovery of the first one and how the relaxation process converges. For this purpose, we terminate the relaxation process after the first five modification-based explanations are found. In this experiment, we evaluate only the best-ranked priority function that considers the average path(1) cardinality and three relaxation heuristics including full relaxation, maximum impact, and minimum cardinality. The experimental results are provided in Figure 5.13. As quality measures, we use two metrics such as a syntactic distance described in Section 3.2.2 and its improvement, which is calculated as the difference in syntactic distances between the first and the best discovered explanations among the five ones.

In Figure 5.13, the evaluation for the syntactic distance of the best explanation is represented dependent on its order. The x axis denotes the order of explanations from 1 to 5 and the y axis represents the smallest syntactic distance among the explanations found until a specific order. A new explanation might have a larger syntactic distance than previously discovered ones, therefore, Figure 5.13 shows the syntactic distance of

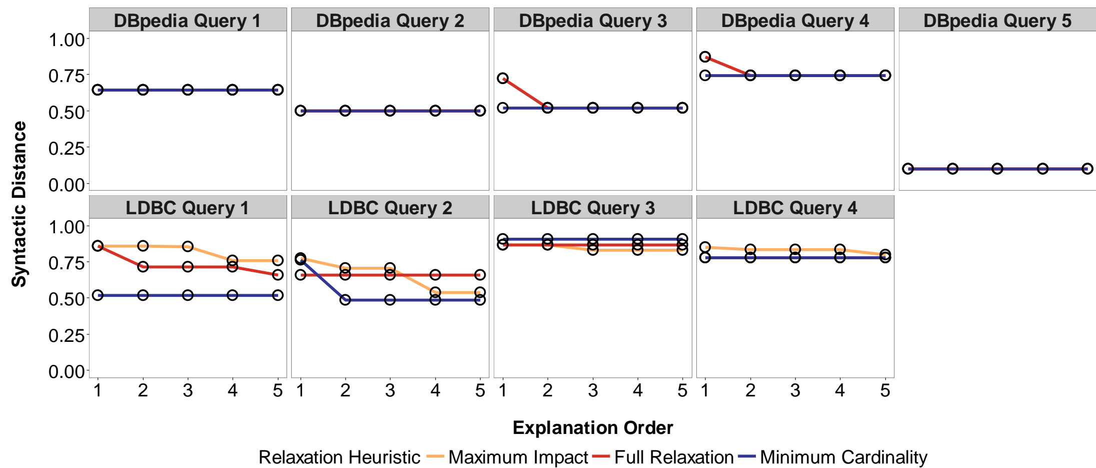


Figure 5.13: Syntactic distance of first five discovered explanations

the best explanation detected to a particular point in time. In six out of nine cases, a better candidate can be found with an increasing number of considered explanations. According to the used data sets, the convergence of DBPEDIA queries happens in three out of five cases directly with the first discovered query candidate, while for LDBC queries better solutions can be found later. This different behavior can be explained by the characteristics of the evaluated queries. Thus, the DBPEDIA queries have fewer constraints than the LDBC queries and therefore the relaxation space is smaller.

We also analyze how syntactic distances of discovered explanations improve among different relaxation heuristics in respect to evaluated data graphs. Relaxation heuristics allow to discover better results with time. The maximum improvement achieves between 24% and 35% for the LDBC queries among different relaxation heuristics. This improvement is rather limited for the DBPEDIA queries and varies between 0% and 28%.

According to the LDBC queries, the maximum-impact relaxation provides always a better solution over time with improvement from 5% to 30%. These results show that this strategy prefers stronger relaxed queries first and postpones less modified ones to a later point in time, which leads to this improvement.

In contrast, the minimum-cardinality relaxation has 0 as the minimum value. Indeed, this heuristic produces immediately the best rewriting in seven out of nine cases and it also provides the largest improvement for the LDBC QUERY 2 as presented in Figure 5.13, which achieves up to 36%. However, for LDBC QUERY 3 the minimum-cardinality relaxation does not improve over time.

The third evaluated heuristic, the full relaxation, considers all possible relaxations at all iterations and has the largest relaxation space among heuristics. Potentially, this heuristic can provide the best improvement, if all rewritten queries are tested. However, in this experiment we limit the relaxation process for the discovery of only the first five solutions. As a result, only a few solutions are processed before the relaxation is terminated and full relaxation does not provide the best improvement in the limited time.

To summarize, in this experiment, we investigate the possibility of receiving a better explanation with a lower syntactic distance if the search is resumed after the discovery of the first one. We consider the first five solutions for LDBC and DBPEDIA queries and compare the achieved improvement of the syntactic distance among three relaxation heuristics. The analysis shows that improvement of a syntactic distance can be achieved

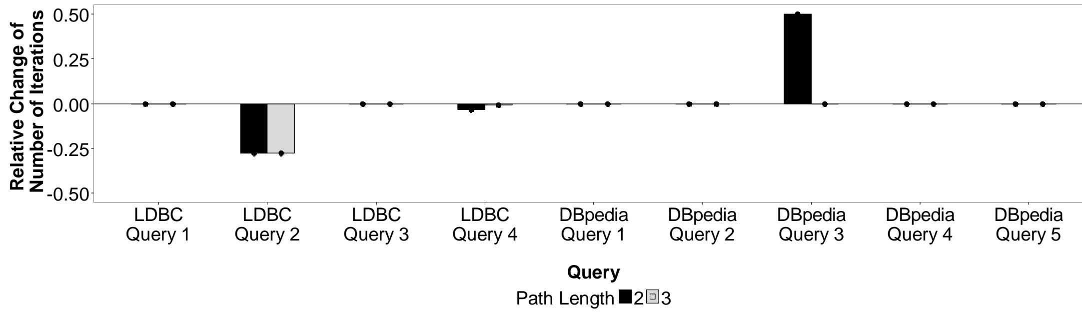


Figure 5.14: Relative change of number of iterations for path(2) and path(3) with respect to path(1)

for queries with multiple attribute and topological predicates such as LDBC queries. For topological queries such as the DBPEDIA queries, in the most cases the best explanation is discovered directly at the beginning and therefore the syntactic distance is not improved. This conclusion can also be explained by the small number of constraints that DBPEDIA queries have and their small sizes. We also test the improvement distribution for different relaxation heuristics. Although the full relaxation might potentially deliver the best explanations, the achieved improvement is rather small for the limited number of considered refinements. In contrast, the maximum-impact relaxation generates more relaxed queries first and therefore the syntactic distance decreases in most cases with new generated modification-based explanations. The minimum-cardinality relaxation delivers often the best solution as the first one.

5.5.3 Priority Function with Average Path(1) Cardinality and Induced Cardinality Changes

Based on the previous set of experiments, the proposed priority function that compares the average path(1) cardinality and induced cardinality change out-performs other priority functions. Therefore, in this experiment, we consider memory consumption for storing query-dependent statistics, processing time, and specific parameter n of this function that shows the length of considered paths. As a relaxation strategy, the maximum-impact relaxation is used. The priority function is initialized with path lengths $n = 1, 2$, or 3. We run this test ten times for each query and path configuration and represent here the average values. The absolute values are provided in Section B.2.

Path Length

For evaluating the influence of a path length on quality of generated explanations and performance of the rewriting process, we use the relative change of a syntactic distance as a quality measure, which is calculated as follows:

$$change(path(n)) = \frac{distance(path(n)) - distance(path(1))}{distance(path(1))} \tag{5.7}$$

The relative change equals 0 means that it is exactly the same like the syntactic distance derived by considering path(1) cardinalities. In the same way, we calculate the relative change of a normalized number of iterations, a performance measure, that describes how performance of the generation process changes by considering longer paths.

According to the evaluation results, the same solution is discovered for all path configurations for seven out of nine queries and therefore the relative change of a syntactic distance equals 0 for them. The only exception is the DBPEDIA QUERY 3,

which solution is better and has a syntactic distance, which is smaller by about 25% for the path(2) configuration than for the path(1) setup. However, this improvement comes at additional costs—the number of iterations for this case increases by 50%, which is illustrated in Figure 5.14.

According to the performance evaluation in Figure 5.14, there are some improvements for two out of nine queries. For the LDBC QUERY 2, the why-empty system requires 25% less iterations for path(2) and path(3) configurations than for the path(1) setup. For the LDBC QUERY 4, the number of iterations is reduced by about 5% with respect to the path(1) configuration.

For two cases, LDBC QUERY 2 and DBPEDIA QUERY 3, we see the strongest changes, because the used path lengths correspond to the query diameter and therefore use the most available information which can be extracted for the query.

To summarize, for a third of the evaluated queries, the use of increased path lengths improves either the quality or performance of the query prioritization. This improvement has been seen for those queries, which diameters equal to the used path length, because the maximum available information about cardinality, which can be extracted, is used.

Memory Consumption of Query-Dependent Statistics

In addition to the quality and performance of the best priority function, we consider in this section the resource consumption for storing the query-dependent statistics. We first describe the memory consumption for the path(1) setup and then its increase by the use of larger paths.

Query-dependent statistics are populated in two ways: Before the relaxation of a failed query begins, we initialize the query-dependent statistics by collecting the cardinalities for query vertices and edges. During the relaxation process, cardinalities for paths are stored, which are required by the candidate selector and query relaxer. Therefore, in this experiment we measure which part of the statistics is occupied during initialization and how much memory is consumed for storing it by the end of experiments. The evaluation results show that the largest part of statistics is occupied at runtime for storing path cardinalities. During initialization at most 40% of the statistics is filled.

We also analyze the size of query-dependent statistics with respect to the size of data graph for three path lengths. The LDBC data graph occupies 1503.49 MB of memory, while the DBPEDIA graph consumes only 91.31 MB. If we consider the path length 3, the query-dependent statistics occupy less than 0.005% of memory required for storing a corresponding data graph. By considering smaller paths, this number is even smaller and do not exceed 0.004%.

To summarize, about 60% of the statistics is populated online during the query rewriting. The consumed memory for storing query-dependent statistics is minor and does not exceed 0.005% of consumed memory for keeping a data graph.

Response Time Distribution

In this experiment, we consider distribution of the response time if the why-empty rewriting system uses the best evaluated priority function that considers the average path(1) cardinality. The absolute numbers for this evaluation are represented in Section B.2.

In Figure 5.15, the distribution of response time is shown for nine evaluated queries and three path lengths. The x axis provides the length of a path, which is used for prioritizing query candidates. The y axis describes the time distribution in percent. The

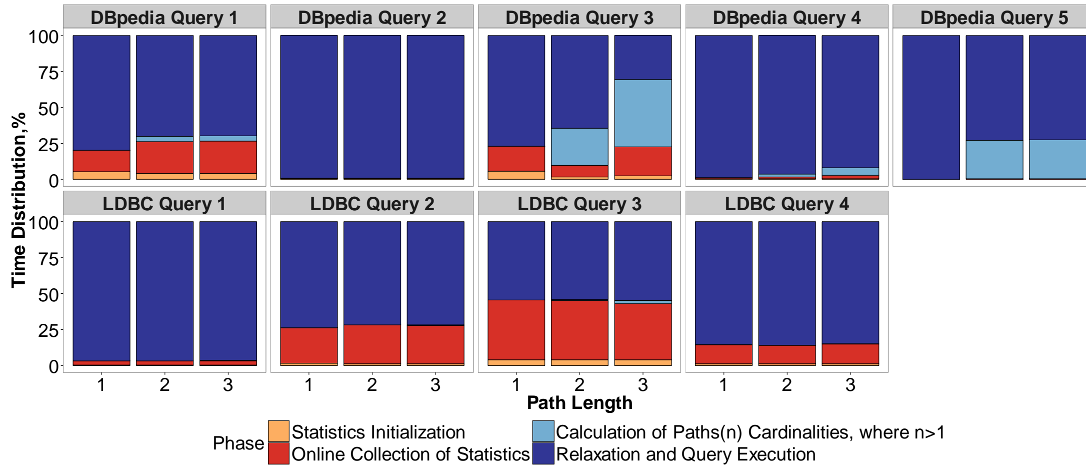


Figure 5.15: Time distribution for evaluated queries

generation of modification-based explanations includes the following phases: statistics initialization, online collection of statistics, calculation of path(n) cardinalities (with $n > 1$), and relaxation itself, which also includes the time for checking query candidates on the delivery of non-empty results. The calculation of path(1) cardinalities is included in the online collection of statistics.

First, statistics initialization requires the least time among the all stages and consumes at most 7% of the time, which can be seen for DBPEDIA QUERY 3. Second, the longest phase is the relaxation process itself, which consumes from 30% up to 95% of the total response time. Third, online collection of statistics requires from 1% up to 45% of the response time, where the maximum value is needed for the processing of the LDBC QUERY 3 that exhibits the maximum number of constraints among all queries. The path(n) calculation is minor in seven out of nine cases and takes no more than 5% of the processing time. However, its maximum values (about 40%) are shown for the DBPEDIA QUERY 3 and QUERY 5. This can be explained by the fact, that these two queries have a limited number of attribute predicates and therefore the cardinalities of paths, which have to be calculated, are higher than for LDBC queries, which have multiple attribute predicates.

To summarize, in most cases, the longest time is required for the relaxation process itself, which includes query relaxation and execution of the most promising query candidates. With increasing a path length, the why-empty system consumes additional time for calculating path(n) cardinalities, which is minor for queries with multiple attribute predicates. However, for topological queries with only a few attribute predicates, this consumption can grow very strongly with an increasing path length.

To summarize, in this experiment, we evaluate specifics of the best performing priority function, which considers average path(1) cardinality and induced cardinality change: the length of a path, the size of query-dependent statistics, and distribution of response time. The evaluation results show that with increasing the path length, the number of required relaxation iterations and syntactic distance can be reduced by up to 25% and 30%, correspondingly. However, the improvement of the syntactic distance comes at additional costs: the performance reduces by up to 50%. This performance reduction is observed for the DBPEDIA queries, which include only a few attribute predicates, and therefore, the calculating of path(n) cardinalities is complicated by the high number of data instances for paths. However, this behavior is not observed for the LDBC queries with multiple attribute predicates. These conclusions are also supported by the

time distribution, which show the higher time consumption for DBPEDIA queries and path(n) calculations. We also consider the size of query-dependent statistics, which highest amount is collected at runtime and occupies at maximum several kB of memory, which is a little amount for large data graphs. To conclude, the increase of a path length can derive better results and faster, if queries include attribute predicates. For topological queries, the calculation of long paths should be avoided.

5.5.4 User Integration

To evaluate the feedback-based relaxation process proposed in Section 5.4, we use an automatic approach based on a pseudo-relevance from the information retrieval research [47]. We run the why-empty system in a general mode (no heuristics are considered during the query relaxation) and collect the discovered explanations. We consider the first and second solutions for simulating a user feedback and create in total six user-preference models for each query and feedback configuration (*Reject*; *Accept*; *Reject, Reject*; *Reject, Accept*; *Accept, Reject*; *Accept, Accept*).

In this evaluation, we compare two setups for the why-empty system: without and with user feedback. In the first setup, without user feedback, the why-empty system uses the maximum-impact relaxation strategy and the best performed priority function, which considers the average path(1) cardinality and induced cardinality change. In the second setup, with user feedback, the why-empty system selects elements for relaxation based on the feedback-based strategy. For this purpose, it first relaxes the query according to the setup without user feedback and collects one or two explanations with non-empty answers. A user-preference model is derived from them and the relaxation process is re-launched such that the why-empty system modifies an original query according to the created model.

We use the appearance order of the explanations as a performance measure and an aggregated score (relevance) of the solutions as a quality measure. The relevance of a modified query is an aggregated weight of the edges and vertices it consists of, which are extracted from the corresponding user-preference model. The relevance $\in [0, 1]$ is normalized to the maximum relevance score an explanation can have.

Two sets of experiments are conducted for three LDBC queries, which include LDBC QUERY 1, LDBC QUERY 2, and LDBC QUERY 4 described in detail in Section A.2. In the first set, only one successfully modified query is rated as accepted or rejected and therefore two user-preference models, *Accept* or *Reject*, are calculated. In the second set, both explanations are rated according to the four user-preference models, i.e., *Reject, Reject*; *Reject, Accept*; *Accept, Reject*; *Accept, Accept*. In total, we execute 18 tests: 6 of them in the first setup and 12 runs in the second setup. All experiments are interrupted, after the first 100 query candidates are tested on the delivery of non-empty results.

Figure 5.16 shows the average relevance among evaluated queries on the y axis in respect to six user-preference models. The light gray bars correspond to the experiments running without user feedback according to the first setup described above. The dark-gray bars illustrate relaxations in the second setup, when a user-preference model is considered. According to this evaluation, by using a user feedback we can generate more relevant modification-based explanations for users. The relevance improvement achieves up to 25% for *Accept, Accept* and *Reject, Accept* user-preference models. No improvement is obtained in average for the *Reject* user-preference model.

Some additional experimental results are provided in Section B.1 including runtime evaluation of relevance and relevance distribution for discovered explanations and query candidates. All results are presented in respect to evaluated queries and user-preference models.

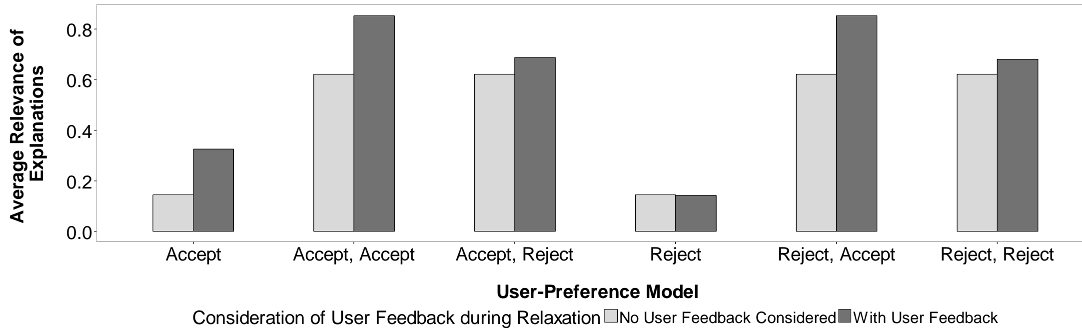


Figure 5.16: Relevance distribution of discovered explanations

In this evaluation, we consider user ratings for discovered query solutions in order to adapt the relaxation process dependent on the user preferences. According to the evaluation results, we can conclude that user feedback facilitates early discovery of user-relevant modification-based explanations for the empty-answer problem. The best improvement of the relevance is derived if a user rates several successfully modified queries and achieves up to 25% for two user-preference models such as *Accept, Accept* and *Reject, Accept*. Combining ratings of several solutions creates a fine-granular user model, which facilitates a more gradual selection of query elements for relaxation.

5.5.5 Evaluation Summary

In this section, we evaluated the why-empty system for generating modification-based explanations according to several features including a priority function, a relaxation heuristic, and user integration. As the best performing priority function, we considered the function that inspects two criteria for prioritizing query candidates in the candidate selector: an average path(1) cardinality and induced cardinality change. In general, this function leads to a fast discovery of solutions.

According to a relaxation heuristic, the best distributions of syntactic distance and number of relaxation iterations were achieved by the minimum-cardinality heuristic in conjunction with the best performed priority function. This heuristic also provides the most stable behavior among top-3 priority functions. To summarize, for evaluated queries, the why-empty engine generated the best modification-based explanations in a time frame of two minutes if it used the best performed priority function with average path(1) cardinality and induced cardinality change and minimum-cardinality relaxation strategy.

Considering the best priority function, we also tested its parameter n that describes the path length and resource consumption such as response time and memory for storing query-dependent statistics. The use of large paths can lead to a faster generation of modification-based explanations. However, the calculation of large paths resulted in longer response times and therefore should be omitted.

Although all proposed optimizations improved the performance of the relaxation process, they can potentially deliver non-interesting explanations to a user. Therefore, we also evaluated the use of user feedback in the relaxation process based on six user-preference models derived automatically from ratings of one or two successfully modified queries. The evaluation results showed that the relevance of solutions could be improved in average by up to 25% if two solutions were rated.

5.6 Summary

In this chapter, 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 a new query failed it is relaxed and the process is repeated until a non-failed rewritten query is discovered.

The proposed method is optimized in several ways. First, only limited search space is considered by relaxing the failed query. Second, query candidates are checked on the delivery of some answers according to their promise to deliver non-empty results. Third, the system maintains query-dependent statistics that allow to store and reuse already queried cardinalities.

In this chapter, a short overview how to integrate a user into the refinement process was also given, which allows to steer the search according to an estimated user-preference model. This model can be easily integrated with above presented optimizations for the user-aware relaxation.

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 propose fine-grained changes in attribute descriptions of a failed query for why-so-few and why-so-many queries.

6

Fine-Grained Cardinality-Driven Query Modification

In Chapter 4, we discussed how to generate subgraph-based explanations for why-queries and proposed two algorithms: DISCOVERMCS and BOUNDEDMCS. While DISCOVERMCS focuses mainly on solving the empty-answer problem, BOUNDEDMCS considers the cardinality threshold and therefore is more appropriate for too-few- and too-many-answers problems. Both approaches are coarse-grained and take only the topology of a query graph into account. As subgraph-based explanations, both of them deliver differential graphs, which represent that query parts which can be discarded from the original query in order to receive a better number of results and therefore can be used as an upper bound of a syntactic distance for generating modification-based explanations.

In Chapter 5, we proposed the next debugging step: a modification-based approach for why-empty queries that in addition to topological modifications considers coarse-grained predicate removals. According to this proposal, the originally failed query is modified by removing its subgraphs or predicates without considering fine-granular modifications of predicate values. Although such changes can be applied to all why-query types, this approach does not consider the cardinality threshold and is more appropriate for why-empty queries.

Therefore, in this chapter we will go one step further and propose a method to generate topology- and predicate-aware modification-based explanations, which considers the cardinality threshold and supports fine-grained topological and predicate changes. First, we give a general overview about the modification process in Section 6.1 and introduce an operational representation for graph queries, which serves as a base for query refinement, and a modification tree for tracking the rewriting process. We describe in detail the modification in Section 6.2 and several pruning techniques, which are used during query modification in Section 6.3. Finally, the proposed techniques are evaluated in Section 6.4.

6.1 Predicate- and Topology-Aware Modification Process

If a query delivers unexpectedly too few or too many answers, it can be rewritten by changing its structure or predicates in such a way that the size of a result set increases or decreases. In this section, we give a general overview about such a modification approach that supports both topological and predicate changes in order to deliver results

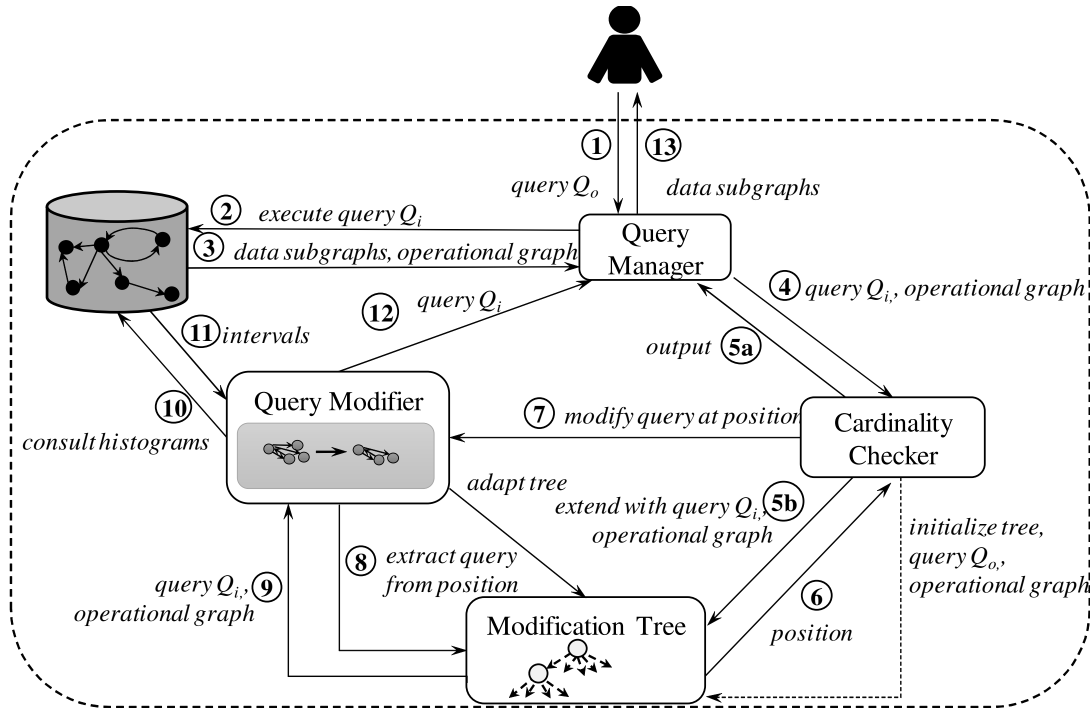


Figure 6.1: Modification process for why-so-few and why-so-many queries

of a better size with respect to a given cardinality threshold. In addition, 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.

6.1.1 General Modification Process

The rewriting procedure is presented in Figure 6.1. 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 describe how to construct it in detail in Section 6.1.2.

As a next step, 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 outputted 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. How to construct and maintain a modification tree is described in detail in Section 6.1.3. 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. This process of generating a new query candidate is described in detail in Section 6.2.2. Finally, a produced 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.

Operator	Abbreviation	Relational Algebra	CYPHER
`getVertices(vertex)`	$GV(v_i)$	σ, π	$(label : \{properties\})$
`traverse(path(h))`	$TR(e_j, h)$	$\sigma, \bowtie$	$source - () \rightarrow target$
`filterPath(path(h))`	$FP(e_j, h)$	σ, π	$[: type\{properties\}]$
`filterSource(path)`	$FS(e_j)$	$\sigma, \bowtie$	$(label : \{properties\}) \rightarrow [\,]$
`filterTarget(path)`	$FT(e_j)$	$\sigma, \bowtie$	$[\,] \rightarrow (label : \{properties\})$
`join(path,path)`	$join(e_j, e_k)$	$\bowtie$	$multiple\ traversals$

Table 6.1: Graph processing operators used in operational graphs

To summarize, the modification process is implemented as an iterative procedure, which intensively uses a modification tree and operational graphs.

6.1.2 Operational Graph-Query Representation

Before discussing how to generate modification-based explanations for why-so-few and why-so-many queries, in this section we introduce one additional representation of graph queries, an operational graph, which serves as a base for fine-grained query modifications.

In this thesis, we have already discussed two representations of graph queries for the property-graph model. The first one expresses a query as a property graph itself with vertices and edges, which are annotated with predicates for attribute values. This is the native representation for queries over property graphs. This model is described in detail in Section 3.1.1 and extensively used for traversing query graphs and generating subgraph-based and modification-based explanations in Chapters 4 – 5.

The second representation, the set-based model provided in Section 3.2.2, models a query as a set of vertices and edges, which are sets of their predicates. This concept is extensively used for calculating a syntactic distance between two graph queries and for judging the quality of generated explanations.

While the property-graph and set-based representations provide general and detailed query descriptions, respectively, they lack information about how a query is processed in a database, which database operators are used, and how they depend on each other. In other words, what is still missing is a query processing model that shows which database operators in which order extract matching data subgraphs from a data graph. We will use this model as a base for query modification in order to discover a query, which delivers results of a better cardinality.

The query processing model is an operational graph with nodes describing database operators and edges between them illustrating the propagation of operator results bottom-up along it. During the execution of this graph, each node is annotated with its selectivity and output cardinality. The operational graph processes the data graph starting from the bottom nodes. The input for each operator on the bottom of the operational graph is a data graph and the output is a set of data subgraphs.

For the property-graph model, we consider the following operators: acquiring data vertices (paths) according to the queried vertex (path), path traversal, filtering of a path, a source, and a target, and joining several paths. Table 6.1 describes a complete list of these operators, which provide general functionality required for pattern match-

(a) Example query

(b) General operational representation

(c) Relational-algebra representation

(d) NEO4J representation

Figure 6.2: Example query and its operational representations

ing. All operators have parameters `vertex` and `path(h)` that denote a query vertex or a path of length h with predicates for attribute values and types.

Example In this example, we describe the mapping between general operators of an operational graph and operators of two mostly-used graph-processing engines: RDBMS and the NEO4J database which are also given in Table 6.1. While RDBMS providing a graph abstraction allow to process graph and relational data together, the NEO4J database is one of the most popular graph databases implementing the property-graph model. For RDBMS, we represent corresponding operators of relational algebra. For the NEO4J database, we provide operators of the CYPHER language [146] that is one of the first query languages delivered by native graph databases, which is currently used as a base for a graph query language standardization. Assume the pattern query in Figure 6.2a consisting of four vertices and three edges. Figure 6.2b shows the canonical operational graph consisting of operators from Table 6.1. We substitute the general operators with the operators from the linear algebra in Figure 6.2c and from CYPHER in Figure 6.2d. The query language CYPHER implemented in the NEO4J database belongs to the class of traversal-based languages, where the discovery of subgraphs is tightly coupled with the order of path and vertex descriptions in the query. The result of the query is denoted as *output* in Figure 6.2d.

As we can see, the proposed operational graph is general enough to be used across GDBMS and RDBMS, which provide a graph abstraction. In the example above, we demonstrate its appliance to RDBMS and NEO4J database. In order to use an operational graph for query rewriting, we need to annotate it with output cardinalities of its nodes. We characterize each node also with its selectivity $sel(node)$ that is calculated from its input and output cardinalities:

$$sel(node) = \frac{C(node)}{\prod_{i=1}^{|children|} C_i}, \text{ where} \tag{6.1}$$

C_i denotes the input cardinality of its i-children and C_{node} describes the output cardinality of the node.

To summarize, in this section we propose the operational graph for pattern matching queries, which consists of graph processing operations required to produce a set of data subgraphs from a data graph, which match a given pattern. The operational query model considers dependencies between different operators and serves as a base for generating fine-granular modification-based explanations. The second important aspect of the rewriting process is a modification tree, which is illustrated on the bottom in Figure 6.1 and described in detail as follows.

6.1.3 Modification Tree

In order to maintain the modification process, we require the possibility to keep executed changes along with cardinality alterations they produce. For this purpose, the modification process stores all information in a data structure called *modification tree*. The root of the 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 branch, every operator of an operational graph appears only once. This property guarantees the termination of a modification process. Nodes of a modification tree keep the following information: a modified operator from an operational graph and a corresponding refined query with its operational graph. 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.

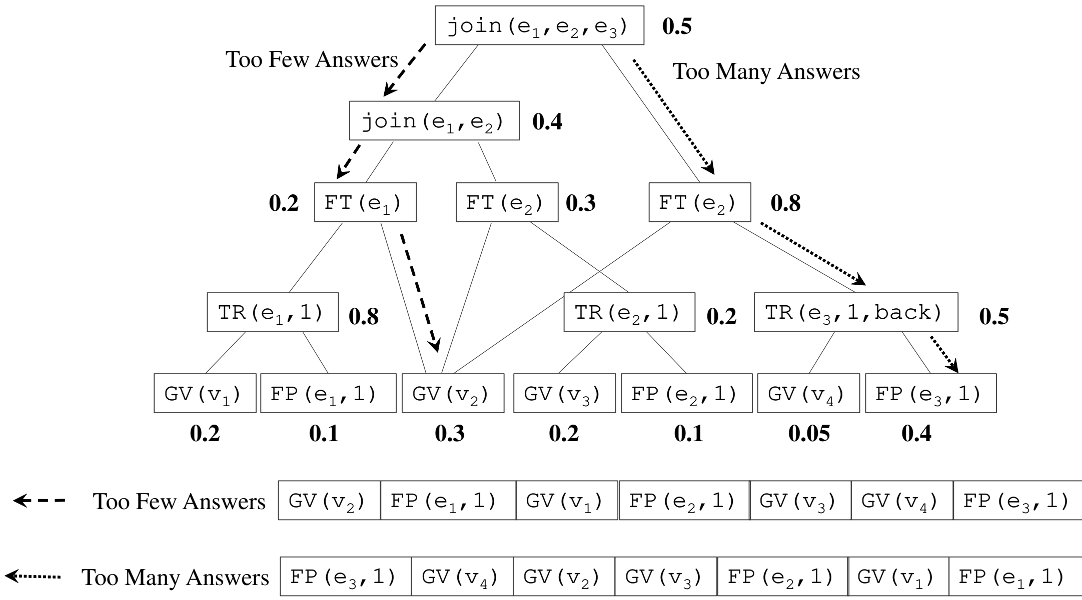
A modification tree supports several operations: extension, backtracking, pruning, and finalization. During *extension*, a new node is appended to a modification tree, which includes the last modified operator from an operational graph, a modified query, and its operational graph. *Finalizing* a tree branch means that a modification tree cannot be further extended at a current branch. *Backtracking* triggers traversing a modification tree bottom-up from a current position by a given number of changes. It also implies finalization of backtracked branches. *Pruning* rejects given modifications and forbids the refinement of corresponding operators from an operational graph on later iterations. It can also imply backtracking and finalizing tree branches.

Extension of Modification Tree

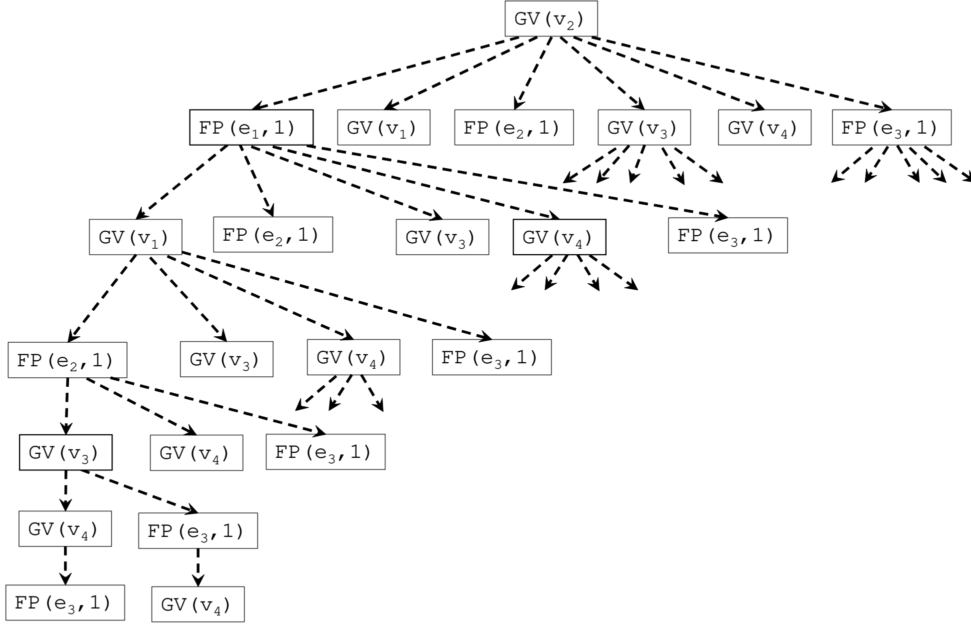
At initialization, the modification tree consists only of a root node that represents an operational graph of an original query annotated with selectivities and cardinalities. In each rewriting iteration, a modified operator is appended to a modification tree.

To extend the modification tree with a new node, the following steps are executed. (1) It should be decided which problem has to be solved: too few or too many answers. For this purpose, the cardinality difference between the last modified query and the cardinality threshold is calculated. (2) The non-processed leaves of the operational graph are extracted from the operational graph of the last modified query according to the problem to solve, which was determined at the previous step. Extracted operations are stored in a FIFO queue. (3) The first operator is extracted from the queue to extend the modification tree.

The main extension step implies extraction of operations from an operational graph according to a problem definition. For the too-many-answers problem, the operational graph is traversed top-down along the branches with higher selectivity values first and the leaves are collected according to the traversal order. For the too-few-answers problem, the branches with lower selectivity values are processed first. Some nodes in the



(a) Extraction of operator queues



(b) Modification tree for too-few-answers problem

Figure 6.3: Construction of modification tree for running example

operational graph can have several parents. To ensure the node uniqueness along each branch of the modification tree and termination of the search, each node is represented only once in the extracted queue.

For query modification, a single node is extracted from the queue and modified in the query. A produced refined query is executed and stored together with its operational graph in the modification tree.

To extend the modification tree vertically, the first operation from the queue is extracted. To extend the modification tree horizontally, operators are used from the second and larger positions of the queue.

Example The modification tree is constructed online and therefore its structure and size depend on which concrete problem has to be solved at a specific point in time: the too-few- or too-many-answers problem. Assume the operational graph with the assigned selectivities in Figure 6.3a for our example query in Figure 6.2a. On the bottom in Figure 6.3a, the created queues are presented for the too-few- and too-many-answers problems. A part of a potential modification tree for the too-few-answers problem is depicted in Figure 6.3b.

To summarize, the modification tree allows us to track changes applied to the query and cardinality fluctuations of corresponding operational graphs and to reject unnecessary changes by discarding some search branches and backtracking. In the following, we will describe the overall modification process, which maintains a modification tree at runtime.

6.2 Generation of Modification-Based Explanations

The proposed operational graph describes how different database operators are connected with each other and how they influence the output cardinality. For example in Figure 6.2b, the top operator corresponds to $join(e_1, e_2, e_3)$, whose output cardinality represents the result cardinality. An operational graph is constructed and annotated with cardinalities and selectivities during the query execution. On the bottom of the operational graph, all extraction operators are situated for filtering paths and acquiring vertices. In order to improve the output cardinality of a query, we have to change the output cardinality of the root of an operational graph that depends on all operational nodes below it. This can be done in two ways: we can consider topological or predicate changes. All applied changes have to be propagated along the operational graph to the root and modify selectivities and cardinalities of its nodes on the way. The modification of predicates implies changing the leaves of the operational graph, which describe predicates for attribute values and types. The question here is how and in which order to modify the leaves. For the topological changes, some nodes of the operational graph and connections between them can be removed, which can be modeled by changing their selectivities and cardinalities.

To summarize, by designing an algorithm for generating fine-grained modification-based explanations, several observations have to be taken into account:

1. Any change applied to the operational graph should propagate to its output node and all non-contributing modifications have to be rejected in order to keep the syntactic distance minimal.
2. The structure of the operational graph has to be considered in order to take advantages of known dependencies.
3. The query modification process has to be adapted according to rejected changes and current output cardinality.

To guarantee these properties, the algorithm has to be able to track the applied changes and to reject them if necessary. For this purpose, we construct an adaptive modification tree at runtime described in Section 6.1.3, which consists of modified leaf operators from the operational graph, and permits backtracking along it with prohibition of non-propagating changes. Each node of the modification tree is unique in the scope of a single branch and is annotated with its corresponding query candidate and operational graph. We have already presented the properties of a modification tree and supported operations in Section 6.1.3. Therefore, as follows in Section 6.2.1, we discuss in detail the backtracking procedure for query rewriting and change of query predicates in Section 6.2.2.

Algorithm 11 TRAVERSESEARCHTREE Algorithm

Input: modified query G'_q, number of tracked changes N
Output: number of pruned changes $counter$
Global Variables: original query G_q, best refined query G_q^{best}, cardinality threshold C_{thr}, modification tree $tree$, prohibited changes $prohibited$, $threshold$ for syntactic distance, processed prefixes $prefixes$

```
1: // 1. Check rejection and acceptance conditions, track best candidate
```
2: **if** syntactic distance between G'_q and $G''_q > threshold$ **then**
3: finalize branch
4: **return** N
5: $prefix \leftarrow$ calculate prefix for G'_q
6: **if** $prefix \in prefixes$ **then**
7: finalize branch
8: **return** N
9: $dataSubgraphs \leftarrow$ execute G'_q
10: **if** $|dataSubgraphs|$ satisfy C_{thr} **then**
11: $G_q^{best} = G'_q$
12: finalize branch
13: **return** 0
14: **if** $cardinalityDistance$ between G_q and G'_q < $cardinalityDistance$ between G_q and G_q^{best} **then**
15: $G_q^{best} = G'_q$
16:
```
17: // 2. Ensure propagation of changes or backtrack tracked changes
```
18: $position \leftarrow$ extend $tree$ with G'_q
19: $trackedChanges \leftarrow$ extract N last changes from $tree$
20: $queue \leftarrow$ extract non-changed elements for G'_q from G_q and $prohibited$
21: **if** $trackedChanges$ have non-processed neighbors $\in queue$ **then**
22: reorder $queue$ according to $trackedChanges$
23: increment N
24: **else if** $trackedChanges$ are non-propagated **then**
25: **return** N
26: **else**
27: $N = 0$
28:
```
29: // 3. Modify query graph by recursive call - traverse modification tree
```
30: **while** $queue \neq \varnothing$ **do**
31: $node \leftarrow$ extract from $queue$
32: **if** G'_q can be modified with $node$ **then**
33: $G''_q \leftarrow$modify G'_q with $node$
34: increment N
35: $counter \leftarrow$ TRAVERSESEARCHTREE (G''_q, N)
36: **if** solution found **then return** 0
```
37: // 3a. Prohibit branch
```
38: **if** $counter > 1$ **then**
39: prune $node$ in $tree$
40: $prohibited \leftarrow$ insert pruned $node$
41: **return** $counter - 1$
42: **else**
43: $prohibited \leftarrow$ insert non-modifiable $node$
44: **return** 0

6.2.1 TRAVERSESEARCHTREE Algorithm

In Algorithm 11, the TRAVERSESEARCHTREE algorithm is described, which explores the modification tree and operational graphs of already processed modified queries and implements the search properties for generating modification-based explanations described above. This is an iterative backtracking procedure that ensures change propagation and prohibition of non-contributing changes. It requires a modified query and the number of tracked changes, which are not propagated. The following global data structures are used: an original query, a cardinality threshold, a modification tree, a set of prohibited changes, a best refined query, a threshold for the syntactic distance, and already processed prefixes. The procedure includes three important steps: rejection and acceptance of a query candidate, change propagation, and query modification.

(1) Rejection and Acceptance of Query Candidates

After an improved query version is generated, we check whether it and its subsequent tree branch can be rejected from the search. Rejection prevents (1) the evaluation of too strongly modified queries and (2) repetitive consideration of similar candidates. For this purpose, at line 4 in Algorithm 11, both original and rewritten queries are considered as sets and a syntactic distance between them is calculated as an average set-set distance of their vertex and edge subsets. A set-set distance of each subset is modeled as a modified Hausdorff distance [45]. If the calculated distance exceeds the predefined threshold for the syntactic distance, the modified query is rejected and the corresponding branch of the modification tree is finalized. The algorithm traverses one step back and the query modifier extracts a query candidate from a new position in the tree and generates a new candidate along the neighboring branch. In our setup, we prevent changes, which result in 50% difference from the original query. This condition effectively reduces the depth of the modification tree.

If the syntactic distance of a refined query satisfies a predefined threshold, the query modifier checks whether a similar candidate has already been processed at line 8 by comparing the query with prefixes of already evaluated queries. A prefix represents the unique name of a changed node from the operational graph. Thus, each query candidate can be identified by the sum of its prefix hashes. During modification, we collect all these values and check prefix hashes for all new candidates against cached ones. If the same hash sum exists, a similar candidate has been already processed and this candidate can be rejected. This process called a prefix optimization allows us to effectively limit the size of the modification tree. Originally, the backtracking has a worst-case complexity of $\mathcal{O}(N!)$, where $N = |V| + |E|$. The proposed prefix optimization allows to reduce it up to $\mathcal{O}(2^N)$. Although this complexity is high, it is effectively reduced by other optimizations, which we present in the following.

After the rejection test, a refined query is sent to the query manager, which triggers its execution and redirects the query with its operational graph to the cardinality checker to test the delivery of the required amount of data subgraphs at line 10 in Algorithm 11. If the root cardinality of the operational graph matches C_{thr}, the query is accepted, the search is terminated, and query results are delivered to a user. Refined queries with empty results are assumed to be modified too strongly and are rejected because instead of refining original query results they discard all of them.

After the acceptance test, we analyze whether the query result has a smaller cardinality delta to a cardinality threshold than all previously evaluated query candidates and can be stored as the best candidate at line 14.

(2) Change Propagation

If after the rejection and acceptance tests, the rewritten process is still running, the cardinality checker extends the modification tree with the evaluated query and its operational graph and redirects a corresponding position in the tree to the query modifier, which has to control whether the changes applied by the tracked operations caused a cardinality change. For this purpose, those operators are extracted from the queue of a corresponding position in the tree, which were not modified in the query and not pruned by previous rewritings. In addition, the tracked operations are derived from the modification tree whose changes did not affect a query output cardinality and whose propagation has to be guaranteed at lines 21 – 27.

To ensure the change propagation and to optimize the search, the next branch of the modification tree is selected based on two conditions: (1) whether the search direction has changed (from too-many to too-few and vice versa) and (2) whether some tree branches have been prohibited. For this purpose, the query modifier tracks the highest element in the operational graph, which exhibits a cardinality change. If there are some operators, whose modifications can force this highest operator to propagate its cardinality change up the graph, they are prioritized and the queue of non-changed elements is reordered. Otherwise, the tracked changes are pruned and backtracking is triggered according to the number of tracked changes. The check is conducted online to react to the modification tree adaptation, which is described in detail in Section 6.3.

(3) Query Modification with Branch Prohibition

After the queue of non-modified operators has been adapted at line 22 in Algorithm 11, the backtracking algorithm resumes by traversing along each modification-tree branch to discover the best query candidate at lines 30 – 43. The first node is extracted from the queue and modified if possible. If its output cardinality has been successfully changed, the query is rewritten by adapting $node$ in G'_q and traversing the modification tree down the branch at lines 33 – 35. Non-modifiable operators are pruned for any following modification at line 43 and the next branch from the queue is traversed (see line 30). If some changes have to be pruned, the corresponding operators are removed from the modification tree and stored as prohibited ones at lines 38 – 41.

To summarize, the cardinality-driven modification process is modeled as a backtracking procedure along the modification tree, which reduces the search space by pruning non-progressing branches. The search begins at the root (no changes to the original query are done) and constructs the modification tree online by inserting modified nodes from the operational graph and prohibiting the non-propagating branches. During the modification process the best query candidate is stored and compared against each new one.

In this section, we have already presented three ways to reduce the search space such as the rejection of query candidates with large syntactic distances or delivering empty results and prefix optimization. In the following, additional strategies for pruning the search space will be introduced, namely: adaptive construction of a modification tree and prohibition of a tree branch. Before discussing them, the core of the rewriting process, the generation of a new query candidate, will be described in detail.

6.2.2 Generation of New Query Candidates

In the previous sections, we discussed the overall modification process, which is a backtracking procedure allowing to prohibit non-propagating changes and to revert

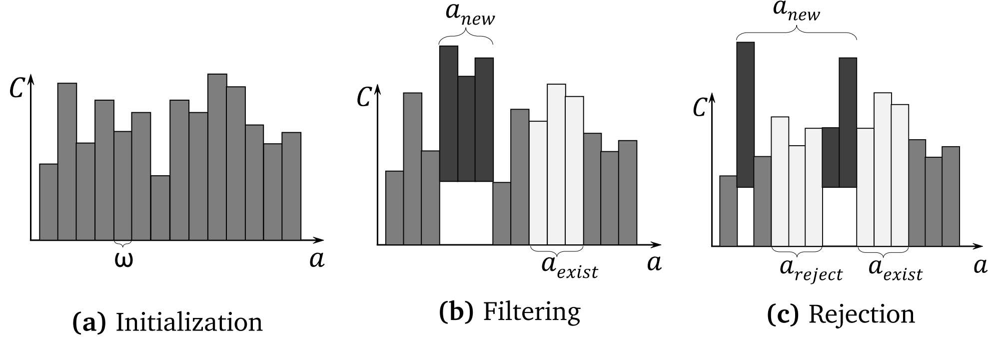


Figure 6.4: Interval modification

changes. The key component of this procedure is the modification of a specific operator from the operational graph as presented in Algorithm 11 at line 33.

To produce a new query candidate, the following input is required: (1) a previous query candidate with its result cardinality, which has to be improved and (2) an operational node, which has to be modified. In total, there are two kinds of changes, which can be applied to a query: predicate and topological changes. Any removal of operational leaves deletes the corresponding graph elements and all dependent nodes in the operational graph. Therefore, the system first tries to change only predicates of the given operational node. If it was unsuccessful, topological changes are applied.

Predicate Modification

To change predicates of an operational leaf, first we have to calculate a cardinality distance to the cardinality threshold. This measure describes a delta cardinality that has to be removed or introduced in a result cardinality. For a positive cardinality distance, a query extension is triggered; otherwise, its restriction is performed.

Second, we refine a corresponding vertex or path according to the cardinality distance. For this purpose, equi-width attribute histograms are constructed offline before the modification process starts. The histograms are built for all edge and vertex attributes in the data graph and describe cardinalities of attribute values [115]. An extension is achieved by adding new values for predicate attributes or types via their disjunctions according to the set-based model described in Section 3.2.2. The restriction removes some predicate values from the query. Here, the query generator has to take the main modification decision: which values have to be removed or attached to a predicate. For this purpose, the attribute histograms are consulted as follows.

Interval Extension Let C_i be the result cardinality of query Q_i and C_{thr} be the cardinality threshold. Cardinality distance $\Delta C = C_{thr} - C(Q_i)$ describes which cardinality has to be acquired from equi-width histogram $H(a, w)$ for attribute a and step size w. The solution for this task is a list of the most similar intervals $a_{new} \in H(a, w)$, whose aggregated cardinality $max(\sum C(a_{new})) \leq \Delta(C_i, C_{thr})$. To acquire a list of new values based on histogram $H(a, w)$ shown in Figure 6.4a, intervals $a_{exist} \in H(a, w)$, already used in query Q_i, are skipped and new intervals a_{new} are chosen from $H(a, w)$ by applying the maximum subset algorithm like shown in Figure 6.4b. A modification is rated as successful if the output cardinality of a node changes. To rate the rewriting, the query generator constructs a simple `getVertex(vertex), filterPath(path(n))` query. If the new cardinality differs from its old value, query Q_i is extended by selected predicate intervals a_{new}. Otherwise, the intervals are rejected and new values are chosen from the histogram (see Figure 6.4c).

Interval Restriction If $C_i > C_{thr}$ then those values are filtered out from a_{exist}, whose aggregated cardinality $max(\sum C(a_{exist})) \leq \Delta(C_i, C_{thr})$. In contrast to the extension process, only those intervals are considered in $H(a, w)$, which exist in query Q_i. We also support rejection and iterative acquisition of values for the restriction, if the modification causes no cardinality change.

In general, the number of iterations for acquiring or removing attribute values can be limited by a user-defined number of iterations, a used point-point distance d (see Section 3.2.2), or it lasts until all the values from the histogram are studied. The use of a point-point distance has the advantage that only the values with the shortest distances to already used attribute values are selected. Thus, it also facilitates the discovery of the most similar query candidates. In this thesis, we interrupt this process, when at least one of the following conditions evaluates truly: (1) The number of iterations exceeds the half of the query size expressed by its total number of edges and vertices, (2) The derived interval is shorter than the half of the original one for an interval restriction, or (3) The derived interval is twice as large as the original one for an interval extension. The last two conditions prevent predicate changes, which are stronger that 50%.

Each vertex or edge can have predicates for multiple attributes. If a predicate change did not influence the output cardinality of the operational node, a predicate for another attribute is chosen for the modification. The predicate order for the evaluation is determined by output cardinalities of predicates. For the too-many-answers problem, predicates with higher output cardinalities are first evaluated. For the too-few-answers problem, predicates with low cardinalities are adjusted ahead.

If individual modifications of all single predicates were not successful, predicates for several attributes are evaluated all at once. For this purpose, the predicates are changed only once and new one are involved stepwise until the rewriting is successful or there are no further predicates for refinement. If the conducted modification did not change the output cardinality of an operational node, it is changed topologically.

Topological Modification

Topological modification involves the deletion of an operational node together with its dependent nodes. To remove a `getVertices(vertex)` operational node, we need to discard all `filterPath(path(h))` nodes for adjacent edges. For a `filterPath(path(h))` operational node, those `getVertices(vertex)` nodes for incident vertices have to be removed which have only this path.

The topological change is modeled as an assignment of any vertex or any path to an operational node. In this case, the affected leaves are annotated with the selectivity 1 and their output cardinalities correspond to the number of data vertices for `getVertices(vertex)` and the number of data edges for `filterPath(path(h))` operators. The structure of an operational graph remains unchanged, which allows us to model the search if there would be no topological changes. Removing any leaf affects also the operational nodes on the upper levels. Any upper node has at most two children. If both children have selectivity 1 then the upper-level node is annotated with the same selectivity and its cardinality corresponds to the multiplied cardinalities of its children. If only one child has selectivity 1 then the upper node inherits cardinality and selectivity of its second child.

As we can see, any topological change might also affect the neighboring leaves in addition to a given operational node. They are collected and inserted in the modification tree, whose depth is effectively increased by their total number.

Example Assuming our example query in Figure 6.2a, leaf node $FP(e_2, 1)$ has to be removed from the operational graph in Figure 6.5. This node filters path e_2 of length

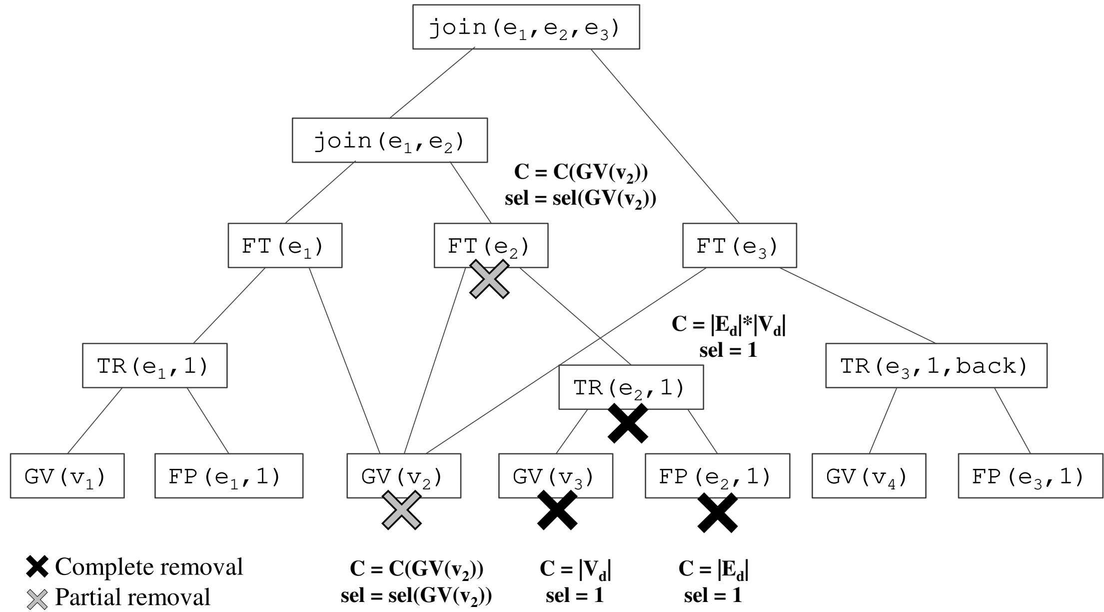


Figure 6.5: Topological removal for operator $FP(e_2, 1)$

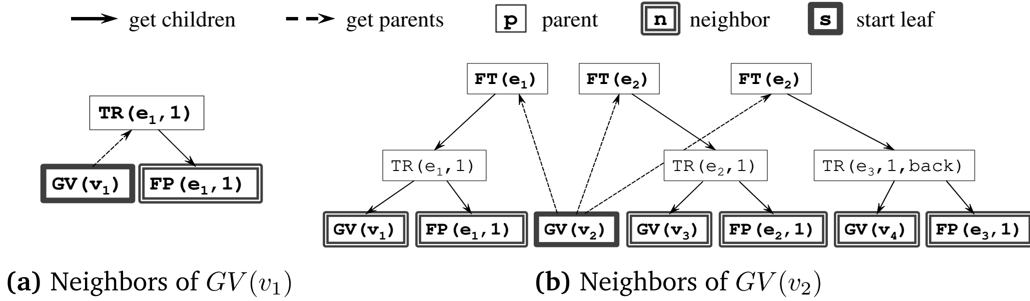
1, is marked as removed, and annotated with selectivity 1. This path has two vertices, one of them v_3 has only one connection. Therefore, its corresponding node $GV(v_3)$ is marked as removed and annotated with selectivity 1. As next, we have to propagate the changes bottom up to parental nodes. Traversal operator $TR(e_2, 1)$ is marked as removed and operator $FT(e_2)$ is annotated with the cardinality and selectivity of its second child $GV(v_2)$. To conclude, two operators have to be inserted into the modification tree: $GV(v_3)$ and $FP(e_2, 1)$.

If a topological change does not result in any cardinality change or creates an unconnected query graph then the complete modification of the node is discarded and it is marked as prohibited at line 43 in Algorithm 11. The node prohibition is spread along the whole modification tree and the prohibited nodes cannot be further used in the modification process. As a result, the modification tree is pruned and modification space is effectively reduced horizontally and vertically.

In this section, we had a look at the modification process with constructing and adapting the modification tree and discussed generation of new query candidates, its core component. The modification tree serves for storing changed operational nodes, modified queries, and their operational graphs and for navigating the search. The backtracking traversal along the modification tree allows to discard non-propagating changes and thus to reduce the search space. In the following section, we will describe optimizations we applied to this algorithm in detail.

6.3 Adaptation of Modification Tree

In this section, we will have a deep look at already sketched techniques, which prevent from unnecessary rewritings and reduce the search space: guaranteeing of change propagation, discarding of non-contributing changes, and pruning of tree branches.



(a) Neighbors of $GV(v_1)$ **(b)** Neighbors of $GV(v_2)$

Figure 6.6: Acquisition of neighbors for leaves $GV(v_1)$ and $GV(v_2)$

6.3.1 Guaranteeing of Change Propagation

This optimization describes the first property that the modification process has to exhibit, namely: any change applied to the operational graph should propagate to the top of the graph and result in a cardinality change. This is achieved by the online adaptation of the modification tree, which is executed at lines 21 – 27 in Algorithm 11.

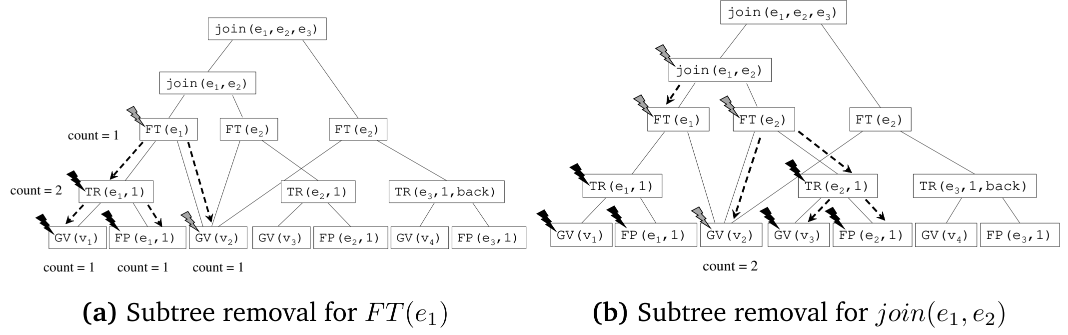
This property is realized by tracking the highest operator in the operational graph, whose cardinality has been changed, and propagating this modification to its parents. If the applied rewriting resulted in a cardinality change of the output node then the search continues without any adaptation. Otherwise, we have to compare the operational graphs of current and previously generated query candidates, i.e. the current position of the modification tree and its parent. We extract the highest node from the operational graphs, which cardinality differs between both query candidates. This is the highest query node along the operational graph, which has to be tracked and whose cardinality change has to be propagated to its parent.

The cardinality of each node strongly depends on its children. If a modification of one of them did not result in a cardinality improvement of its parent then we can assume strong correlation between children. To account for this correlation, the remaining non-modified children have to be adjusted until their delta cardinalities are propagated to the parent. For this purpose, we prioritize them by re-arranging the nodes in the queue on the current level of the modification tree such that they are processed first. The rewriting process continues as usually and goes to the next level in the modification tree. If a change of neighboring nodes affects the cardinality of the (grand-) parent then it is marked as a tracked one and its non-modified children are prioritized in the queue. Otherwise, the remaining neighbors are modified.

Example Following the operational graph for our example in Figure 6.2b, we assume that the change of node $GV(v_1)$ did not improve the delta cardinality of its parents; only its own cardinality changed. Then $GV(v_1)$ is marked as the tracked node, the search moves up and acquires its parent $TR(e_1, 1)$ as presented in Figure 6.6a. To propagate the cardinality change of $GV(v_1)$, its neighbor $FP(e_1, 1)$ has to be modified. If the tracking node has multiple parents like for example $GV(v_2)$ in Figure 6.6b then the children of all node's parents are returned for the modification.

6.3.2 Discarding of Non-Contributing Changes and Search Branches

The previously described optimization focuses on the propagation of changes to the output node of an operational graph. However, it can happen that although all children have already been modified they have no impact on the cardinality of the parent. Such changes increase only the syntactic distance, but are useless in order to achieve the cardinality threshold. Therefore, in this section we discuss the second property of the



(a) Subtree removal for $FT(e_1)$ **(b)** Subtree removal for $join(e_1, e_2)$

Figure 6.7: Subtree removal from modification tree

modification process: non-contributing changes have to be avoided in order to keep the syntactic distance minimal.

If a parent cardinality did not change, the modifications have to be prohibited and the search has to backtrack by the number of prohibited changes, which is tracked during the search. This number is resetted if the changes propagated to the output node. During backtracking, the algorithm traverses the modification tree up, finalizes the corresponding tree branches and marks finalized nodes as prohibited. The filtered-out nodes are not considered further in the search and in such a way the search space is reduced. The backtracking for this case is described at lines 25 and 38 – 41 in Algorithm 11. Considering the example in Figure 6.6a: if after modifying both children, the output cardinality of $TR(e_i, 1)$ did not improve the complete subtree has to be prohibited.

In the operational graph, there can be leaves with multiple parents. Their removal can affect the modification of the parents remaining in the modification tree. To overcome this problem, we consider partial removal for such nodes, which is achieved by tracking the number of removals for the leaves and discarding only those of them which counts equal to the number of their direct parents.

To discard the last changes applied to the query, the tracked parental node broadcasts removal messages to its leaves. Each node increments its internal parental counter by the number of received messages. If the parental counter on a leaf equals to the number of its direct parents then this leaf sends an acknowledgment up the operational graph along the transmission edges and the leaf is finalized. Those non-leaves are marked for complete removal, whose numbers of messages received from children and called children counters equal to the number of children. The leaves, whose parental counters differ from the number of parents, are marked for partial removal. Although the intermediate nodes are not considered during the search, we still require them to prevent the double counting of partially removed nodes.

Example Given the example query in Figure 6.7a and subtree for removal $FT(e_1)$, its root broadcasts messages down the tree. Vertices $GV(v_1)$ and $FP(e_1, 1)$ are marked for complete removal visualized using a black flash. $GV(v_2)$ still has two non-prohibited parents and is marked for partial removal depicted with a gray flash. Assume the modification process continues with the subtree for $join(e_1, e_2)$ and chooses it for deletion (see Figure 6.7b). In this scenario, $join(e_1, e_2)$ broadcasts removal messages along its children. Node $FT(e_1)$ was already finalized and therefore does not redirect the message to its children. Leaf $GV(v_2)$ receives the message from node $FT(e_2)$ and increments its parental counter. However, the parental counter does not equal to the number of parents, therefore, it does not send an acknowledge message to its parents.

To conclude, with the proposed pruning procedure we reduce the modification space, which is achieved by finalizing those branches of the modification tree from which the modification process does not benefit. We prevent removal of those operational nodes, which still can contribute to the modification process.

In this chapter, we discussed how to generate modification-based explanations for why-so-few and why-so-many queries. The underlying backtracking procedure explores an operational graph for the query and adapts the modification tree in order to guarantee change propagation, to discard non-contributing changes, and to reduce the search space. In the following, we will evaluate this approach and compare it with the modified rewriting method for why-empty queries introduced in Chapter 5.

6.4 Evaluation

In this chapter, we evaluate the proposed TRAVERSESEARCHTREE algorithm to produce fine-grained modification-based explanations for why-so-few and why-so-many queries, which guarantees the change propagation and prohibition of non-contributing modifications, and try to answer the following questions:

- Can TRAVERSESEARCHTREE generate explanations with a better size of a result set than the original query has?
- Does the considered method out-performs the baseline approaches according to the quality and performance metrics?
- How do topological changes affect the quality and performance metrics?

Similar to the experiments in previous sections, we consider the following metrics: As quality metrics, we use syntactic, cardinality and result distances, which are described in detail in Section 3.2. The first one, the syntactic distance between an original query and a generated explanation, qualifies how different the evaluated explanation appears to a user. The second one, the cardinality distance, judges how well the applied changes adjust the result size to the cardinality threshold. The last one, the result distance, compares the content of both result sets and shows how much information remains in the answer of the evaluated explanation after query rewriting. As a performance measure, we consider the number of rewritten queries, which have been executed until the best rewritten query has been found.

For evaluating the TRAVERSESEARCHTREE method for generating fine-grained modification-based explanations, we use the LDBC data set described in Appendix A.2. We test four LDBC query templates from cardinality classes $C_1 - C_4$ in Table A.1. For all configurations of each query, we created multiple cardinality thresholds, which correspond to 1%, 10%, 20%, 50%, 200%, 500%, 1,000%, and 10,000% of the result cardinality to an original query such that at least 1 data subgraph has to be delivered and no more than 10,000 subgraphs are requested. The cardinality factors and corresponding cardinality thresholds are represented in Table 6.2.

In Section 6.4.1, we describe the used baseline strategies. Afterwards, in Section 6.4.2, we evaluate them together with TRAVERSESEARCHTREE. We answer the question whether topological changes are beneficial for query rewriting in addition to predicate changes in Section 6.4.3.

6.4.1 Baseline Approaches

The TRAVERSESEARCHTREE algorithm for generating fine-grained modification-based explanations is compared against several methods including two naïve approaches

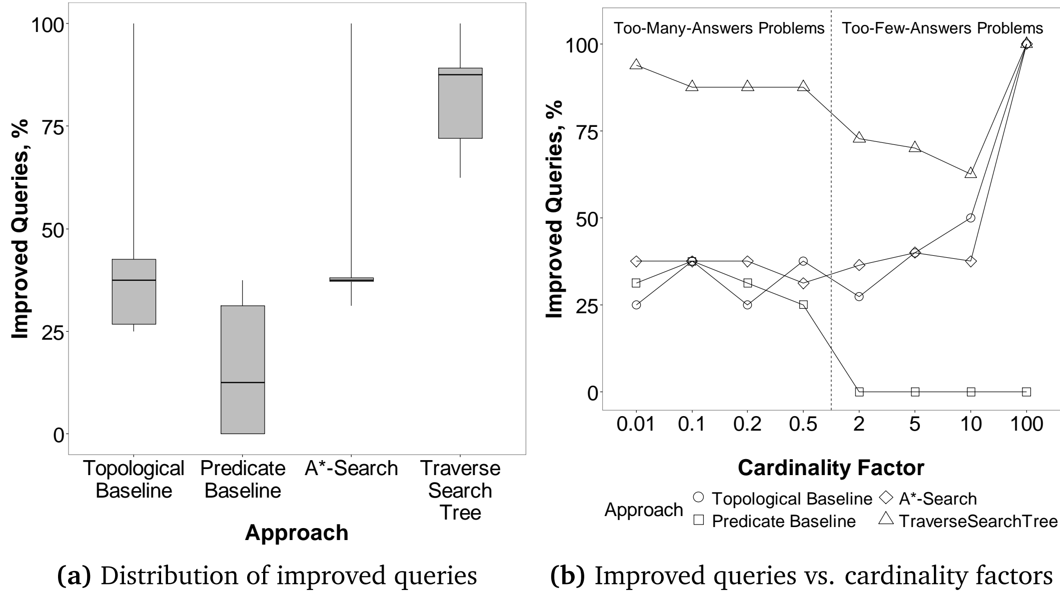
Query	Class	C_o	0.01	0.1	0.2	0.5	2	5	10	100
Q_1	C_1	21	1	2	4	10	42	105	210	2100
	C_2	262	2	26	52	131	524	1310	2620	–
	C_3	1687	16	168	337	843	3374	8435	–	–
	C_4	6991	69	699	1398	3495	–	–	–	–
Q_2	C_1	39	1	3	7	19	78	195	390	3900
	C_2	627	6	62	125	313	1254	3135	6270	–
	C_3	1626	16	162	325	813	3252	8130	–	–
	C_4	5933	59	593	1186	2966	-	-	–	–
Q_3	C_1	188	1	18	37	94	376	940	1880	–
	C_2	654	6	65	130	327	1308	3270	6540	–
	C_3	2139	21	213	427	1069	4278	–	–	–
	C_4	8871	88	887	1774	4435	–	–	–	–
Q_4	C_1	195	1	19	39	97	390	975	1950	–
	C_2	775	7	77	155	387	1550	–	–	–
	C_3	5020	50	502	1004	2510	–	–	–	–
	C_4	5044	50	504	1008	2522	–	–	–	–
In Total	-	16	16	16	16	16	11	9	7	2

Table 6.2: Original cardinalities and cardinality thresholds of evaluated LDBC queries

based on topological and predicate changes and an adapted A*-search described in Chapter 5.

Predicate and Topological Baselines The naïve approach for rewriting why-queries implies changes on vertices and edges in a random order. The maximum number of rewriting steps corresponds to the total number of query vertices and edges. At each step, the system chooses randomly a vertex or an edge for rewriting from the set of non-modified elements. The newly generated query is then checked whether it delivers a required cardinality. If necessary, the rewriting procedure continues by selecting a new query element for modification. This process terminates if a graph query delivering a required cardinality is generated or no further query elements can be modified. We distinguish two baselines based on this naïve solution. While the first one (*Predicate Baseline*) considers only predicate changes, the second one (*Topological Baseline*) implies only structural modifications. Predicate changes are done similarly to the TRAVERSESEARCHTREE algorithm without any propagation guarantee, i.e. one of the predicates on a vertex or an edge is chosen randomly for modification and its interval is extended or restricted as described in Section 6.2.2 by acquiring a delta cardinality from attribute histograms. In both topological and predicate rewriting methods, we track the best rewritten query with the lowest cardinality distance.

A*-Search The third baseline approach corresponds to the coarse-grained modification for why-empty queries described in detail in Chapter 5, which we adjusted in order to be used for why-so-few and why-so-many queries. The adapted version considers a cardinality threshold in order to change predicate intervals by acquir-



(a) Distribution of improved queries **(b)** Improved queries vs. cardinality factors

Figure 6.8: Baseline comparison: distribution and percentage of refined queries

ing a required cardinality delta from histograms like described in Section 6.2.2. Query proposals are ordered based on their syntactic distances. For this approach, we allow both predicate and topological rewriting. Query elements for rewriting are chosen based on the problem definition: For why-so-few queries, elements with lowest cardinalities are preferred. For why-so-many queries, query elements with the highest cardinalities are refined first.

To summarize, in this evaluation we compare TRAVERSESEARCHTREE against three baselines including *Predicate Baseline*, *Topological Baseline*, and *A*-search*.

6.4.2 Baseline Comparison

In the first set of experiments, we compare the TRAVERSESEARCHTREE algorithm and three aforementioned baselines for generating fine-grained modification-based explanations. We analyze quality as well as performance metrics between them.

As a performance measure, we consider the number of iterations that corresponds to the number of refined queries, which are generated before the best refinement is found. For the TRAVERSESEARCHTREE, this number can achieve maximally the amount of nodes in the modification tree. For the topological baseline, this number cannot exceed the total number of vertices and edges in a query graph. For the predicate baselines, it is limited by the total number of predicates in a query. For the A*-search, the maximal number cannot exceed the number of iterations consumed by the TRAVERSESEARCHTREE algorithm rounded to the next large value.

As a quality metric, we consider three measures: syntactic, cardinality, and result distances introduced in Section 3.2. To be able to aggregate cardinality distances between different queries, we normalize them to original cardinality distancess. Not all queries can be refined by all evaluated methods. Therefore, we also consider the percentage of improved queries, which shows how many queries can be refined from the total set of evaluated queries and deliver better cardinalities.

Distribution of Improved Queries

In Figure 6.8, we illustrate which amount of queries is modified and returns the cardinality distance with a value less than 1. The chart in Figure 6.8a describes the distributions of refined queries with minimum, 25% percentile, median, 75% percentile, and maximum. The x axis denotes evaluated algorithms. The most of the queries are refined by the TRAVERSESEARCHTREE algorithm, which also has the smallest variation. The predicate baseline improves the least number of queries, which varies between no and 45% of the queries. This fact is explained by the small number of changes, which are allowed for this baseline. The largest variation is derived by the topological baseline which refines from 25% up to 100% of queries. This baseline considers only topology, which belong to the class of strong modifications that can change the result size dramatically. The A*-search has a similar distribution to the topological baseline with the same median value.

Figure 6.8b describes the percentage of improved queries according to the evaluated cardinality factors. All factors below 1 correspond to the too-many-answers problems, while remaining ones describe the too-few-answers problems. The predicate baseline is able to restrict some queries in order to deliver less results, but it completely fails to produce refinements with more answers for the cardinality factors above 1. The topological baseline and A*-search refine similar amounts of queries. The TRAVERSESEARCHTREE algorithm out-performs compared methods for all cardinality factors, except 100. For this factor, three out of four algorithms refine all queries, which is explained by the small number of evaluated queries. Please note that we evaluate only two queries for this cardinality factor as highlighted in Table 6.2.

To summarize, according to the evaluation results for the number of refines queries, the TRAVERSESEARCHTREE algorithm is able to refine the same up to twice so many queries as compared methods. The predicate baseline fails to solve too-few-answers problems and improves no queries in these scenarios. The A*-search and the topological baselines behave very similarly and modify almost the same amount of queries. The percentage of refined queries quantifies the stability of the refinement process among different queries and cardinality factors. In the following, we will consider specific quality and performance metrics and will start with the cardinality distance, which describes how well different algorithms achieve desired cardinality thresholds.

Cardinality Distance vs. Cardinality Factor

The distributions for cardinality distances are represented in Figure 6.9 with minimum, 0.25 quantile, median, 0.75 quantile, and maximum values for cardinality distances. The x axis illustrates evaluated cardinality factors. The corresponding cardinality thresholds for specific queries are provided in Table 6.2. The medians of three compared baselines approach to the maximum cardinality distance that is equal one, because most of the queries are not improved. In comparison, the median of the TRAVERSESEARCHTREE algorithm varies from almost 0 to 0.8. Also the distribution of our method occupies lower values of the cardinality distance than the compared baselines.

To summarize, the TRAVERSESEARCHTREE algorithm out-performs the compared methods in terms of derived cardinality distances, which are closer to cardinality thresholds. Our algorithm generates explanations almost with exact result sizes for the too-many-answers problems. For the too-few-answers problems, it delivers refined queries with cardinalities which are from 10% to 50% better than ones produced by the evaluated baselines.

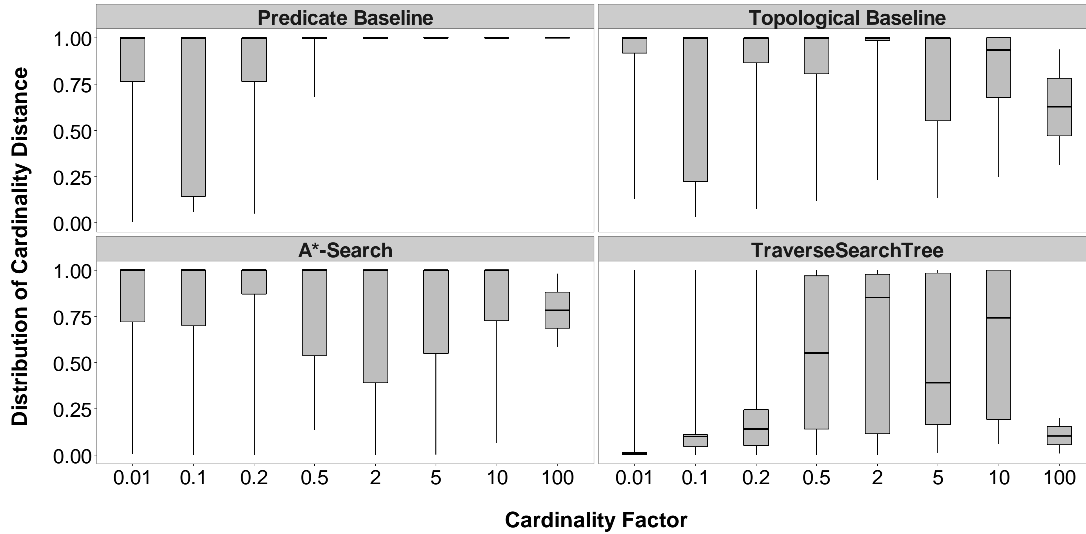
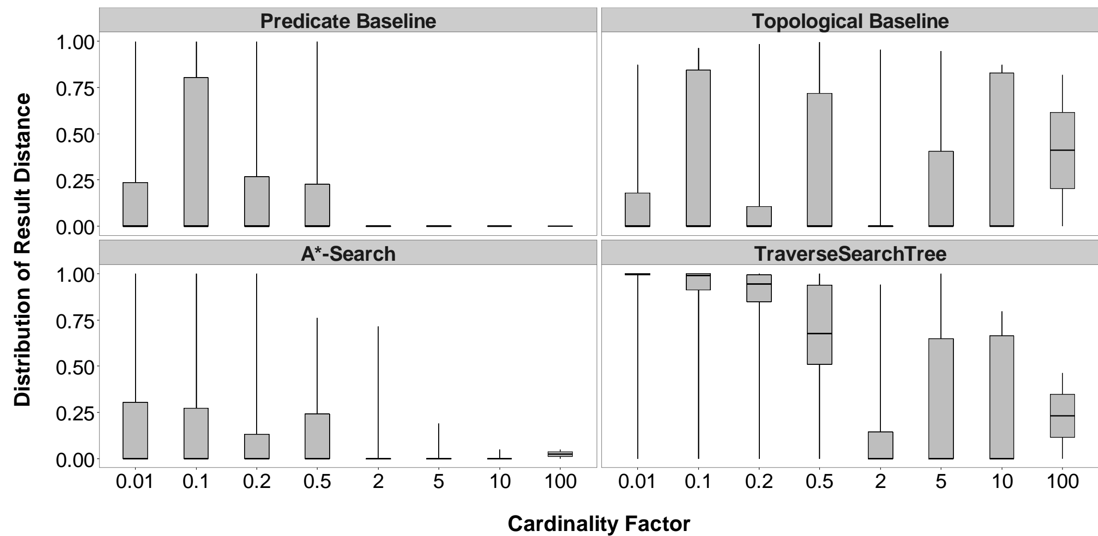


Figure 6.9: Baseline comparison: distribution of cardinality distance

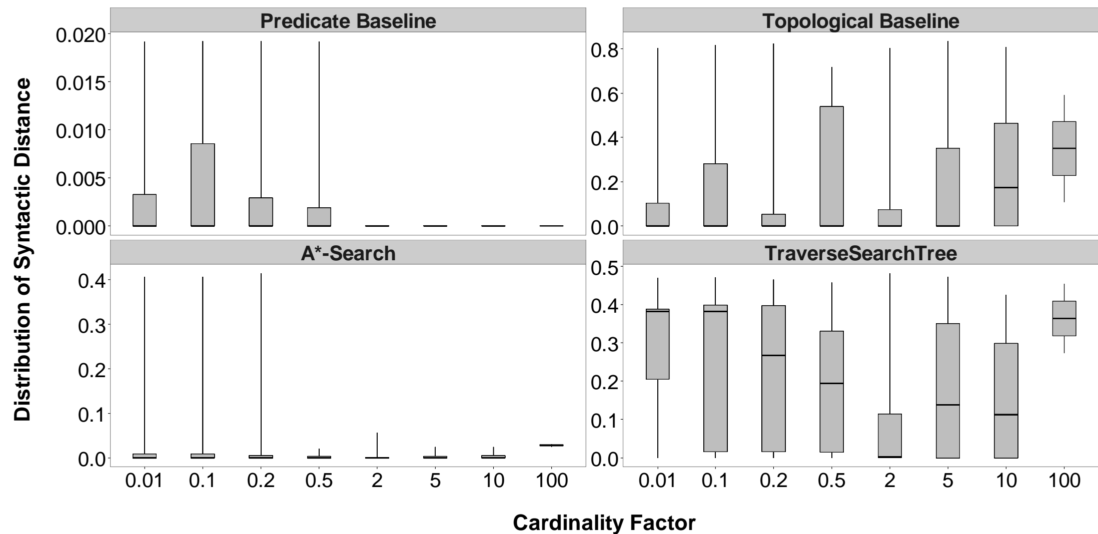
Result and Syntactic Distances vs. Cardinality Factor

As next quality measures, we consider result and syntactic distances with respect to used cardinality factors, which are presented in Figure 6.10. As we can see, syntactic and result distances have similar distributions, which can be explained by the dependency of a result difference on a syntactic measure: It becomes higher if a query is changed more strongly and its syntactic distance is high. As expected, the smallest syntactic distance is derived by the predicate baseline, which considers only predicates and does not modify the structure of a query. The next lowest syntactic distance is provided by the A*-search, which first considers less changed queries and aims to generate explanations with limited deviations to an original query. The topological baseline exhibits the largest syntactic distances caused by strong structural changes. This method does not reject any explanation with a high syntactic value like TRAVERSESEARCHTREE does. The distribution of the result distance for the TRAVERSESEARCHTREE algorithm demonstrates the expected behavior. For too-many-answers problems, the TRAVERSESEARCHTREE method restricts queries such that they return only a part of original results. Therefore, considering the cardinality factor of 0.01, in general we expect to deliver only 1% of the original answers and the corresponding result distance approaches 1. The result distance decreases by looking for such a number of answers, which is close to the original cardinality and corresponds to the cardinality factors 2 and 5. With an increasing cardinality threshold, the number of results grows by acquiring new data vertices and edges and maybe by removing some structural parts.

To summarize, the lowest syntactic distance can be achieved if only predicate changes are allowed or refined queries are processed according to their syntactic distances in an ascending order. Any topological change strongly increases a syntactic difference of an explanation. The TRAVERSESEARCHTREE algorithm considers topological changes for high or low cardinality factors, which make the cardinality thresholds very different from the original cardinalities. It prefers small predicate modifications for low original cardinality distances.



(a) Baseline comparison: distribution of result distances



(b) Baseline comparison: distribution of syntactic distances

Figure 6.10: Baseline comparison: distribution of result and syntactic distances

Number of Iterations vs. Cardinality Factor

To evaluate the performance of generating fine-grained modification-based explanations, we use a number of iterations, i.e., a number of generated refined queries, which are processed before the best explanation is discovered. The evaluation results are illustrated in Figure 6.11, where like in the previous charts, the minimum, 0.25 quantile, median, 0.75 quantile, and maximum values of the evaluated metric are presented. The lowest numbers of iterations are derived by topological and predicate baselines, which consider only a limited number of changes, which does not exceed the size or the number of predicates in a query graph, respectively. The largest variations are detected by the A*-search and the TRAVERSESEARCHTREE algorithm. While the first method allows multiple changes of the same elements, the second one supports only a single modification of the same element in the same refined query. The TRAVERSESEARCHTREE algorithm requires a larger number of iterations according to the median than the A*-search which can be explained by several facts. First, it successfully rewrites up to 50% more queries than other methods. For these cases, the compared approaches have

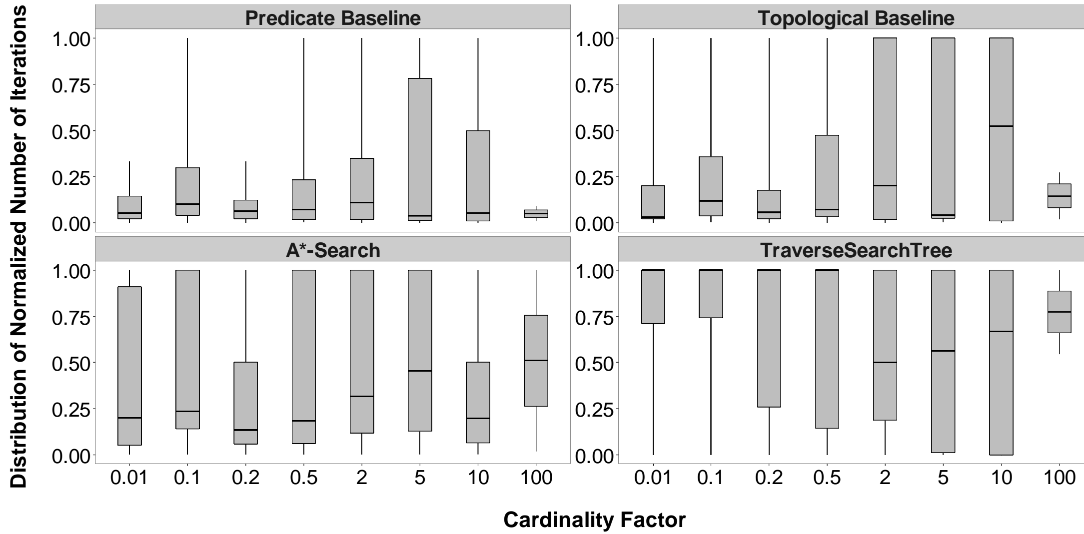
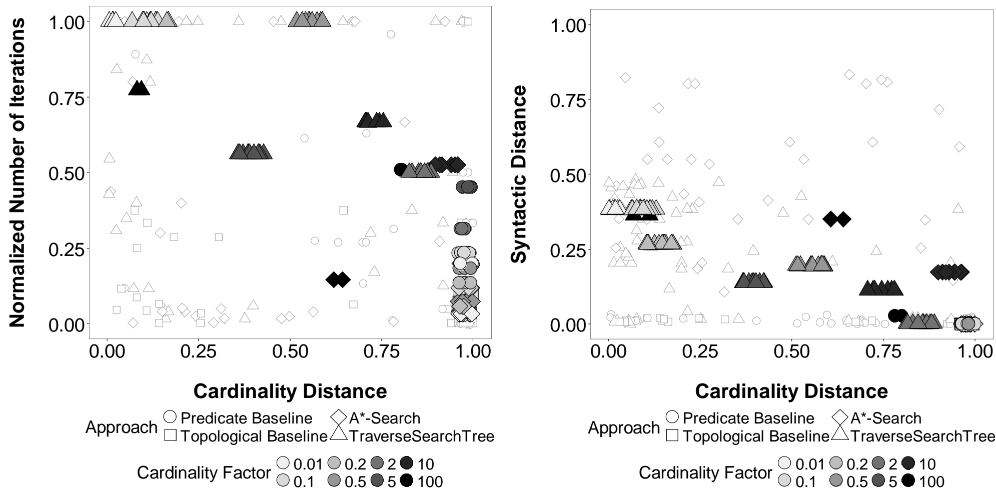


Figure 6.11: Baseline comparison: distribution of normalized number of iterations



(a) Baseline comparison: cardinality distance vs. number of iterations

(b) Baseline comparison: cardinality distance vs. syntactic distance

Figure 6.12: Baseline comparison: dependencies between evaluated metrics

the number of iterations equals 0, while the TRAVERSESEARCHTREE has at least one iteration. Second, the TRAVERSESEARCHTREE also provides better results according to the cardinality, which might require additional efforts. To consider the dependencies between different metrics, we present individual evaluation points in Figure 6.12.

In Figure 6.12a, the x axis describes the cardinality distance of generated explanations and the y axis introduces the normalized number of iterations. We denote different methods with specific shapes of the points and highlight the medium values for cardinality factors with various colors. The median points are also larger than the individual experimental points, which are filled with a white color. As it can be seen, most of the median points for three baselines approach the cardinality distance of 1, which means that they are not successful in refining the queries. Most of the explanations delivered by the TRAVERSESEARCHTREE algorithm have a cardinality distance which is half as large as the original cardinality distance. By considering individual experiments, the TRAVERSESEARCHTREE algorithm tends to consume the largest num-

ber of iterations by generating results with low cardinality distances and requires less iterations, otherwise.

In Figure 6.12b, we also provide cardinality and syntactic distances for specific evaluation runs. As in the previous chart, the large colored points describe medians for different cardinality factors. For the TRAVERSESEARCHTREE algorithm, generated explanations with higher syntactic distances have results with lower cardinality distances and vise versa. Considering only structural changes, we generate explanations with higher syntactic distances as presented for the topological baseline.

To summarize, the topological and predicate baselines consume a small number of iterations, which can be explained by their demands: they consider only one change per query element. The A*-search and TRAVERSESEARCHTREE require a larger number of iterations to produce explanations. Although TRAVERSESEARCHTREE consumes in some cases the maximum number of iterations among all methods, it is compensated by the high quality of generated explanations: it can successfully rewrite twice so many queries as competitors do and reduce the cardinality distance by up to 50% for specific cardinality factors in respect to other approaches.

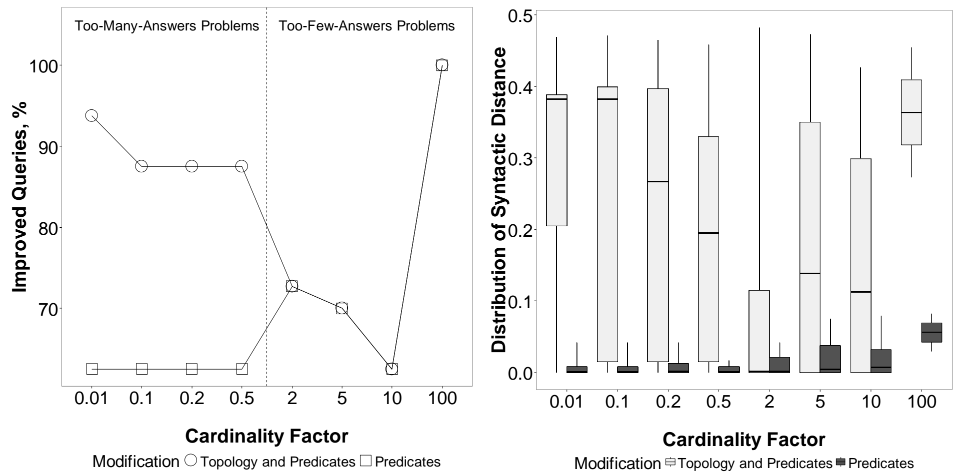
In this set of experiments, we compared the TRAVERSESEARCHTREE algorithm with three baselines for generating fine-grained modification-based explanations for why-so-few and why-so-many queries. We analyzed the following metrics: the number of improved queries, syntactic, cardinality, and result distances, and the number of iterations. The TRAVERSESEARCHTREE algorithm out-performs other approaches according to the cardinality distance, which is the main goal of a rewriting process, and can successfully modify twice so many queries as compared methods do. However, it requires a higher number of iterations for solving the too-many-answers problems.

6.4.3 Topology Consideration

In the previous section, we compared the TRAVERSESEARCHTREE algorithm with three baselines. We concluded that our method produces the best explanations in terms of the cardinality distance and the number of improved queries. However, we also noticed that in some cases it requires a large number of iterations, which can be observed for the too-many-answers problems. This fact can be explained in many cases with the topological changes applied to a query graph, which generate multiple query candidates with too high query distances that are later rejected. Topological changes also can affect several operational nodes at once, which can reduce the success of used prefix optimization. Therefore, in this evaluation, we consider two configurations of the TRAVERSESEARCHTREE algorithm: with and without topological changes. As a performance metric, we use the number of iterations. The quality metrics are represented by the number of successfully rewritten queries, syntactic, cardinality, and result distances as discussed in Section 3.2.

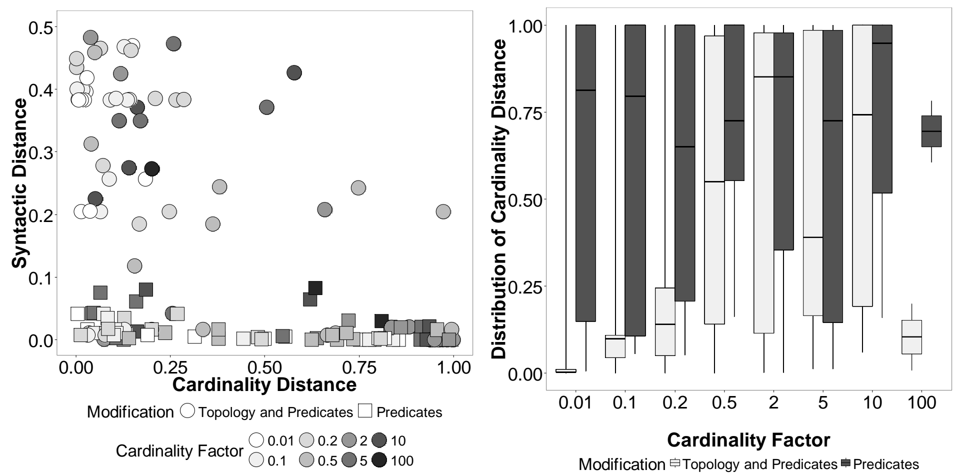
In Figure 6.13, we present the complete evaluation results for these experiments. We evaluate the LDBC queries presented in Table 6.2 for eight cardinality factors. Figure 6.13a describes the percentage of queries improved by the TRAVERSESEARCHTREE algorithm according to cardinality factors annotated on the x axis. The TRAVERSESEARCHTREE algorithm with topological changes refines up to 30% more queries than the setup with only predicate changes for the too-many-answers problems. For the too-few-answers problems, the algorithm improves the same amount of queries independent of its setup. However, from these results we cannot say how good the generated explanations are in terms of cardinality and syntactic distances.

Figure 6.13b illustrates distributions of syntactic distances for different cardinality factors with minimum, 0.25 quantile, median, 0.75 quantile, and maximum values.



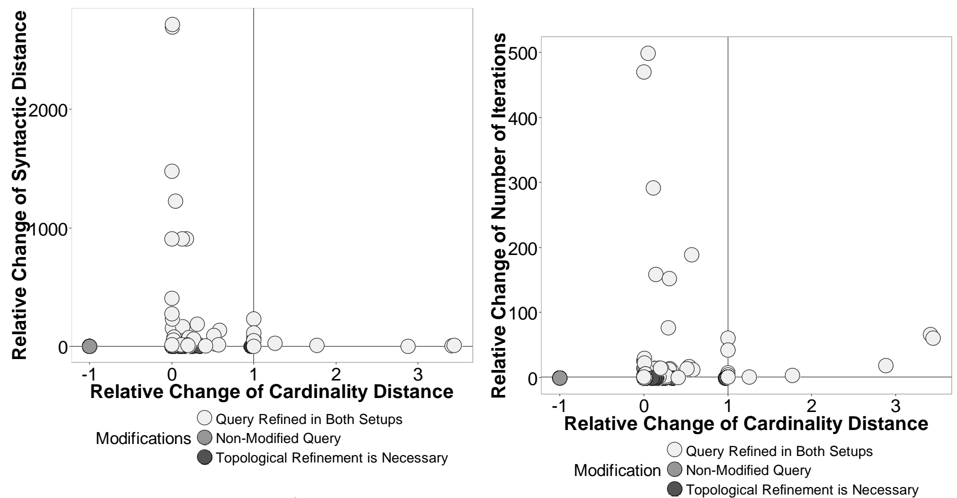
(a) Percentage of improved queries vs. cardinality factors

(b) Distribution of syntactic distance vs. cardinality factors



(c) Syntactic distance vs. cardinality distance

(d) Distribution of cardinality distance vs. cardinality factors



(e) Relative changes of syntactic distance in respect to cardinality distance

(f) Relative changes of number of iterations in respect to cardinality distance

Figure 6.13: Consideration of topological changes in TRAVERSESEARCHTREE algorithm

We distinguish between two evaluated configurations of the TRAVERSESEARCHTREE algorithm with specific colors. First, without considering topological changes, the algorithm delivers some explanations with less changes and smaller syntactic distances. Second, the median of the syntactic distance for the setup with topological modifications achieves the maximum value of 0.38 for the cardinality factor of 0.01. According to Figure 6.13a, the numbers of improved queries differ for too-many-answers problems, while it is the same for the too-few-answers problems. However, considering the syntactic distances in Figure 6.13b, we can conclude that both setups produce different explanations with non-identical syntactic distances. To clarify this situation, we have to consider the quality of derived explanations in terms of cardinality distances they have.

The cardinality distances of produced solutions are illustrated along the x axis in Figure 6.13c, where the y axis denotes the syntactic distances of the best generated explanations. We distinguish between two configurations of the TRAVERSESEARCHTREE algorithm with different shapes of the points. The colors represent the evaluated cardinality factors. The syntactic distances produced by considering only the predicate changes lie in the low half of the chart and correspond to the values, which do not exceed 0.1. However, the cardinality distances for these solutions are distributed along the whole x axis, which show their poor behavior in order to achieve the desired cardinality. If the TRAVERSESEARCHTREE algorithm supports topological changes then the situation improves dramatically: most of the produced explanations have a cardinality distance less than the value of 0.5. Only a few points exceed this value. Considering different factors represented by the colors, we can conclude that although both setups can improve the same number of queries, they produce different results and topological changes allow to deliver better cardinality. For example, for the cardinality factor with a value of 100, topological changes lead to a cardinality distance, which is almost 50% better than that of the explanation derived by the TRAVERSESEARCHTREE algorithm with only predicate changes. We show the distributions of cardinality distances for different cardinality factors in Figure 6.13d. As it can be seen, in most cases both setups deliver different results, which correspond to different cardinality distances. Such topological changes allow not only to improve more queries, but also to deliver better explanations with lower cardinality distances. According to the median values, the topological setup improves the cardinality distance by at most 70%.

In Figure 6.13e, we present the relative difference of syntactic and cardinality distances by considering topological modifications with respect to the configuration without structural changes. We distinguish three kinds of scenarios with colors: a query can be refined in both setups, a query cannot be improved, and a query requires topological modifications. The horizontal and vertical lines represent the axes, where both setups deliver explanations either with the same cardinality distance or the same syntactic distance. As it can be seen, there are several queries which require topological changes. There are a few queries which are rewritten very similarly and therefore are placed on the intersection of lines. We also see at least four points having similar syntactic distances in both setups, but which return results of up to 3.5 times closer to the cardinality threshold with topological modifications. In such scenarios, structural rewritings are of most interest. However, we also see seven points that have similar cardinality distances, but different syntactic differences. For such cases, predicate modifications are preferable, because they deliver less-changed queries and faster.

We use a similar representation for comparing the performance of both setups in Figure 6.13f, where the y axis represents the relative change of the numbers of iterations, and the x axis describes differences in cardinality changes. By comparing two graphs,

we see that if both setups deliver explanations with the same cardinality distances, but with different syntactic distances and number of iterations, then some of the applied modifications are redundant: they only increase the syntactic distance and the number of iterations without giving a clear improvement in the cardinality. Therefore, such modifications should be avoided. If we consider the points right in the chart, which increase the cardinality distance by several times, they reduce the performance only slightly, and therefore topological changes in such cases should be preferred.

To summarize, in this evaluation, we analyzed two configurations for the TRAVERSESEARCHTREE algorithm: the first one considers both topological and predicate changes, the second one conducts only predicate modifications. Rewriting a query topology, we can improve up to 30% more queries for the too-many-answers problems. Although for the too-few-answers problems, the number of improved queries is the same, the quality of generated explanations differs. The topological changes produce refined queries with cardinality distances, which are smaller by up to 3.5 times. Considering cardinality improvements for different cardinality factors, we can conclude that topological changes are preferred for those cardinality thresholds, which differ from original cardinalities in one or two orders of magnitude. For small changes in cardinality, predicate modifications are more preferable.

6.4.4 Evaluation Summary

In this section, we compared the TRAVERSESEARCHTREE algorithm with three baselines for generating fine-grained modification-based explanations: topological and predicate baselines and A*-search. The evaluation results showed that our proposal allows to improve up to 50% more queries than considered baselines. The produced explanations also out-perform the competitors in terms of lower cardinality distances, which are the goal of the rewriting process. The TRAVERSESEARCHTREE algorithm controls the modification process and discards strong modifications which generate refined queries with the syntactic distances larger than 0.5. It provides very similar performance in reference to the A*-search. In some cases, it is still higher, which can be tolerated by the better quality of generated explanations. Considering this fact, we also evaluated two configurations for the TRAVERSESEARCHTREE algorithm: the first one considers both topological and predicate changes, which is used in the first evaluation, and the second one allows only predicate changes. If we change the structure of a query, we can refine up to 30% more queries for the too-many-answers problems and decrease the cardinality distance by up to 3.5 times. However, for the cardinality factors of $\{0.5; 2; 5\}$ we observed that the topological changes increase the number of considered iterations, but still provide very similar syntactic distances. Therefore, in order to deliver the best combination of quality and performance, we should allow topological changes for large cardinality distances and consider only predicate modifications, otherwise.

6.5 Summary

In this chapter, we proposed the TRAVERSESEARCHTREE algorithm for generating fine-grained modification-based explanations for solving too-many- and too-few-answers problems. The presented algorithm is an iterative procedure, which traverses the modification tree that comprises all changes applied to a query in order to generate explanations, ensures propagation of applied changes to the output cardinality, and prohibits non-propagating changes. The TRAVERSESEARCHTREE algorithm makes use of the operational graph of a query for constructing the modification tree and changing

a failed query. In contrast to subgraph-based explanations proposed in Chapter 4 and modification-based explanations in Section 5, we did not present a new model for generating user-aware answers. We believe that the user-integration methods can be used, which were previously introduced in Chapters 4 – 5. 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 the quality of generated explanations and the quantity of modified queries and shows very similar performance to the A*-search. To improve the performance, we studied the role of topological changes in the explanation generation process. We came to the conclusion that if a cardinality threshold differs from an original cardinality in several orders of magnitude the topological changes have to be considered. For small cardinality distances, predicate modifications are good enough to derive better explanations.

7 Conclusion and Future Work

Graph databases implementing the property-graph model allow to store information of a 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 requests that deliver no, too few, or too many results. A user 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 on the example of 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 ones. 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 described in Section 2 include efficient generation of explanations, user integration, generation of different kinds of explanations, discovery of the reason

of unexpectedness, and query refinement. These aspects were then revised for graph databases in Chapters 3 – 6.

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 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. In Section 4, 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 a 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 the 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, in Section 5 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, in Section 6 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 as described in Section 3.2 including the syntactic, cardinality, and result distances. The syntactic distance describes how different an 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 as described in Section 4.4. The user integration is easily done by choosing the most relevant vertices and edges to process. The second approach as presented in Section 5.4, 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.

Future Work

In this thesis, we have set the foundations for making graph databases more user-friendly by introducing debugging capabilities in the form of why-queries. We opened up a new research field for cardinality-driven why-queries over property graphs. There are a lot of interesting open challenges, which can be studied in this area. In the following, we present the most important ones.

Developing Content-Based Why-Queries over Property Graphs In this thesis, we focused on cardinality-based why-queries and proposed several methods for generating explanations for unexpected answers. Content-based queries introduce a new aspect in the generation of explanations. For these queries, a user specifies which results are unexpectedly missing or present in a result set. In case of pattern-matching queries, such unexpected answers can be not just single vertices, but they can represent a query (sub)graph annotated with additional constraints, which express user interest. As a consequence, instead of items of interest known from RDBMS as described in Section 2.2 we have to deal with subgraphs of interests. This new aspect of the content-based why-queries opens several directions for future work including how to specify unexpected (sub)graphs, how to prioritize specific edges, vertices, and their combinations, and how to generate explanations efficiently. All these questions should be investigated in order to provide debugging functionality for content-based why-queries on property graphs.

Considering Graph-Specific Queries This thesis mainly focuses on pattern-matching queries. The study of other graph-specific queries such as shortest path or reachability queries would be of great interest. It is also necessary to keep in mind that not all queries can suffer from all kinds of unexpected results or they can suffer from new kinds of unexpectedness. For example, a why-not query for calculating the shortest path between two nodes can ask why the path does not cross a specific node or why it is so long or short. Such queries require the formal definitions for unexpected-results problems and efficient methods for generating explanations for them.

Studying Usability Issues This thesis mainly focuses on how to generate explanations and optimize the generation process. For the proposed approaches, we also developed a demonstration [139] that represents a typical debugging session starting with defining a query, its debugging, and visualizing the results of the derived explanation. However, what is still missing is the study of a user interaction with the tool, automatic extraction of a user-preference model from his actions, result and explanation representation, and adaptation of a debugging session based on the collected information. With respect to this fact, future work might investigate all these aspects of communication with the debugging tool.

Generating Query-Specific Explanations and Considering Dependencies between Explanations In this thesis, we focused on different debugging aspects including generating multiple explanations, which we supported with subgraph- and modification-based explanations. As we have pointed out in the survey about the state-of-the-art solutions in Chapter 2, also other kinds of explanations can exist, which are problem-specific. For example, why-so-many queries can provide also result-based explanations, which change neither query nor data, but modify the representation of a result set via reordering, clustering, faceting, etc. Investigation of such problem-specific explanations would be an interesting research topic. Moreover, we generate all explanations independent of each other. In addition to investigating different explanations, it would be interesting to combine

them and to re-use already gained knowledge. As we have already mentioned, modification-based explanations can profit from subgraph-based explanations, which can provide the upper limit of applied modifications. All these aspects could be considered by designing and testing the debugging tool.

Considering Intra-Graph Dependencies and Graph-Specific Characteristics Graph queries over property graphs consist of multiple correlated constraints, whose dependencies can affect the generation of explanations. We account for these dependencies in subgraph-based explanations implicitly by traversing the graph. In why-empty queries, we model them explicitly in the form of path(n) cardinalities. In modification-based explanations for why-so-few and why-so-many queries, we provide even stronger consideration of them by re-arranging search branches according to neighboring operators and backtracking. We think our methods can be even more optimized by integrating graph-specific statistics into the process of rewriting predicates or modifying the query topology. It might also be advantageous to extend the graph query with new vertices and edges in order to change the cardinality of the result.

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.

A

Evaluation Setup

This section provides technical details about the evaluation setup used in this thesis. In Section A.1, an overview of the used evaluation system is described in detail. In Section A.2, evaluated data sets and their queries are introduced.

A.1 System Overview

All algorithms and their optimizations are implemented as a C++-based extension of the prototypical graph database GRAPHITE [114], which stores data as a property graph in two separate tables for vertices and edges. In the first table, vertices are represented by a set of columns for their attributes and unique identifiers. In the second table, edges are modeled as simplified adjacency lists with an additional set of columns for attributes. In GRAPHITE, both edges and vertices have unique identifiers. Unique identifiers on edges allow modeling of temporal behavior, which is realized by using timestamps as attributes. To enable efficient graph processing, GRAPHITE provides optimized flexible tables [21, 124], which support insertion of new attributes for edges and vertices. The proposed algorithms and the graph database are run on a single server machine equipped with SUSE Linux Enterprise Server 11 (64 bit) with an Intel Xeon Processor E5-2643 (24 CPUs and 96 GB RAM).

A.2 Data Sets

For the evaluation, two data sets have been used: LDBC and DBPEDIA. The first one is a generated graph for a social network, which is used for benchmarking graph databases. The second one represents RDF data describing general knowledge about famous people, cities and countries, etc. Both data sets are used for evaluating subgraph- and modification-based explanations for no, too few, and too many answers.

A.2.1 LDBC Data Set and Its Queries

The LDBC data graph evaluated in this thesis represents a social network with 3.7 M vertices and 21.7 M edges. This graph provides general information about persons, blogs, forums, and places and considers typical social-network connections like *knows*, *hasMembers*, etc. The schema of the graph is illustrated in Figure A.1. The LDBC data graph allows multiple attributes on vertices and edges and represents a typical property graph without a rigid schema.

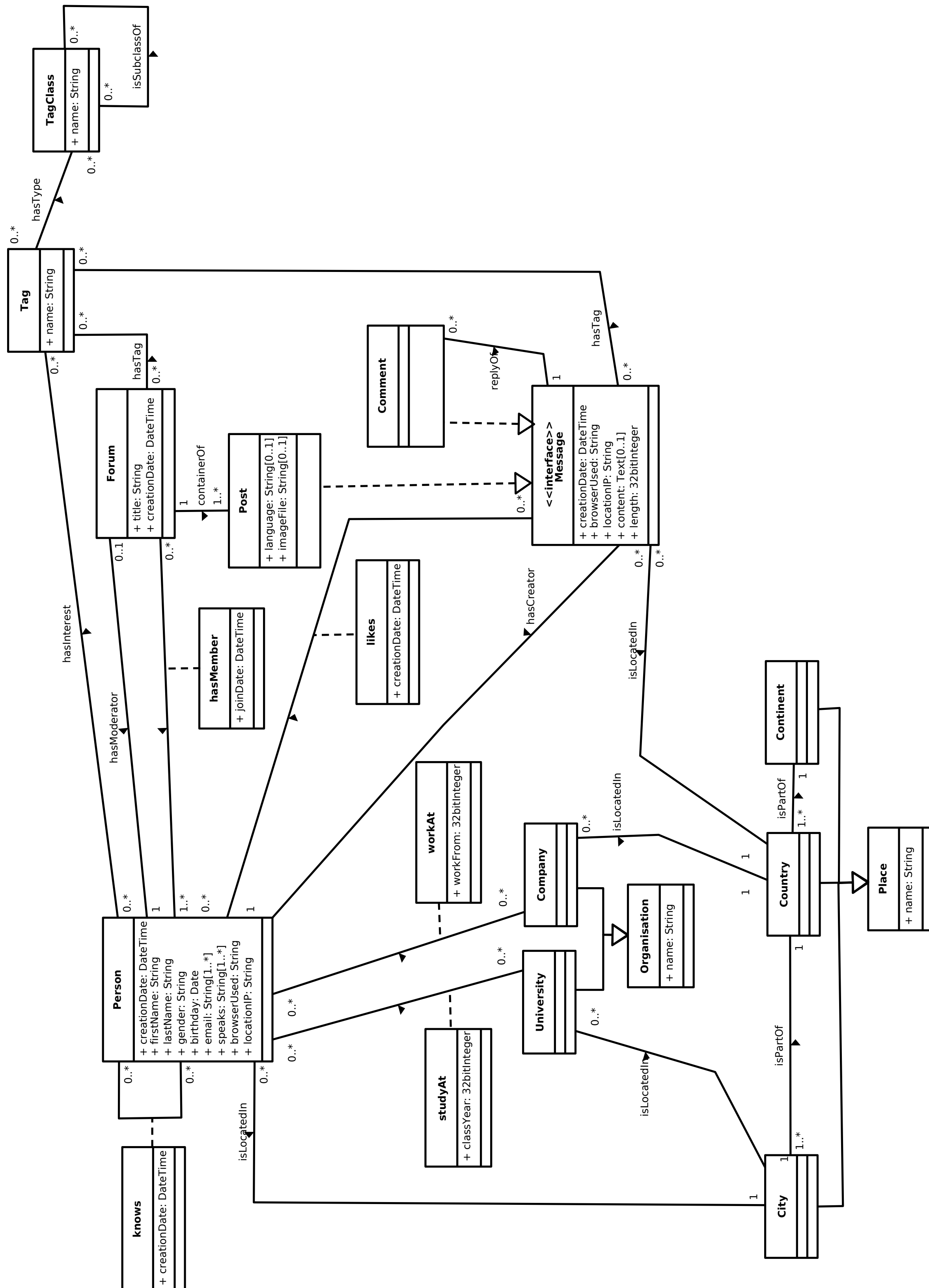


Figure A.1: LDBC social network benchmark schema (Source: Prat et al. [118])

For evaluating LDBC, query templates inspired by the queries Q_1, Q_2, Q_3, Q_5 from the LDBC interactive workload [20] have been created which are illustrated in Fi-

Name	LDBC	$\|V_q\|$	$\|E_q\|$	$\|A_q\| + \|T_q\|$	C_0	C_1	C_2	C_3	C_4
LDBC QUERY 1	Q_1	6	7	16	0	21	262	1687	6991
LDBC QUERY 2	Q_2	3	3	11	0	39	627	1626	5933
LDBC QUERY 3	Q_3	8	8	19	0	188	654	2139	8871
LDBC QUERY 4	Q_5	4	5	12	0	195	775	5020	5044

Table A.1: Evaluated queries and their original cardinalities for LDBC data set

Name	$\|V_q\|$	$\|E_q\|$	$\|A_q\| + \|T_q\|$	C_0
DBPEDIA QUERY 1	4	3	3	0
DBPEDIA QUERY 2	3	2	3	0
DBPEDIA QUERY 3	5	4	4	0
DBPEDIA QUERY 4	7	6	8	0
DBPEDIA QUERY 5	3	2	1	0
DBPEDIA QUERY	10	9	9	217,944

Table A.2: Evaluated queries and their original cardinalities for DBPEDIA data set

gure A.2. For evaluating the empty-answers problems, we configured the predicates for these queries in such a way that they deliver empty results. For testing too-few- and too-many-answers problems, for each query template, four different configurations have been generated by modifying different sets of predicates. The sizes of the result sets for these generated queries are given in Table A.1, which provides the query names used in the evaluation in Chapters 3 – 6, original names from the benchmark, numbers of query vertices, edges, and total numbers of defined predicates and attributes, and five cardinality classes $C_0 - C_4$ with cardinalities of their result sets. The cardinality class C_0 represents those instances of LDBC queries, which deliver empty results and therefore are used for evaluating the why-empty queries. The cardinality classes $C_1 - C_4$ are used to test why-so-few and why-so-many queries.

A.2.2 DBPEDIA Data Set and Its Queries

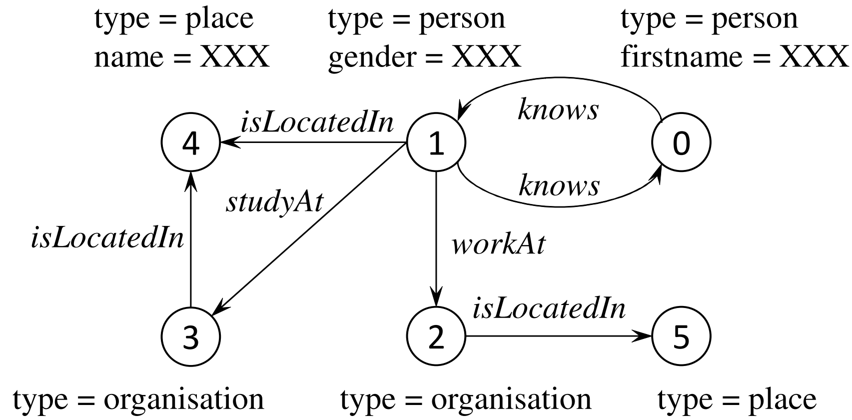
In contrast, the DBPEDIA graph describes mainly a topology with a limited number of attributes, because it is produced from the DBPEDIA RDF graph. The DBPEDIA project extracts the data from Wikipedia that comprises general knowledge of multiple contributors, which post Wikipedia articles, and represents the derived information as a cross-domain knowledge base. The data can be in multiple languages; however, in this thesis, we use only the data set in English. For storing the data, DBPEDIA uses RDF, which allows to query it using SPARQL. In this thesis, we created a property graph from the DBPEDIA tables, which are publicly available[1]. The generated graph consists of 213 K edges and 30 K vertices with 740 attributes, which are sparsely distributed across the graph.

A typical query over RDF data represents a star topology with a few nodes, where the central one has the largest degree and connects to all remaining nodes that typi-

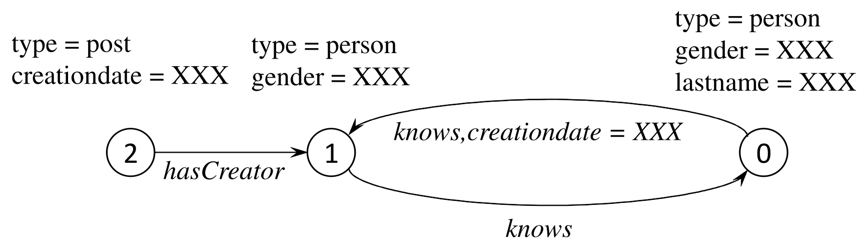
[1] http://wiki.dbpedia.org/services-resources/downloads/dbpedia-tables

cally have only one connection. The queries for evaluating the DBPEDIA are inspired by the real queries to DBPEDIA SPARQL endpoint taken from the public logs[2], which we expressed in the subgraph isomorphism language of GRAPHITE. The evaluated queries represent stars, include only a few vertices, and are illustrated in Figure A.3 and described in Table A.2.

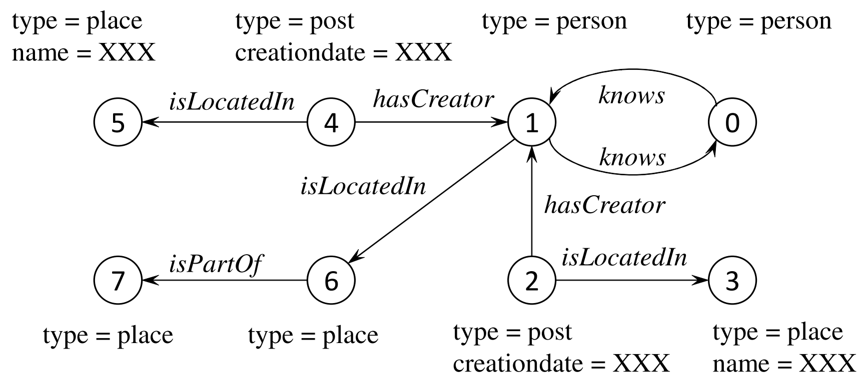
[2]ftp://download.openlinksw.com/support/dbpedia/



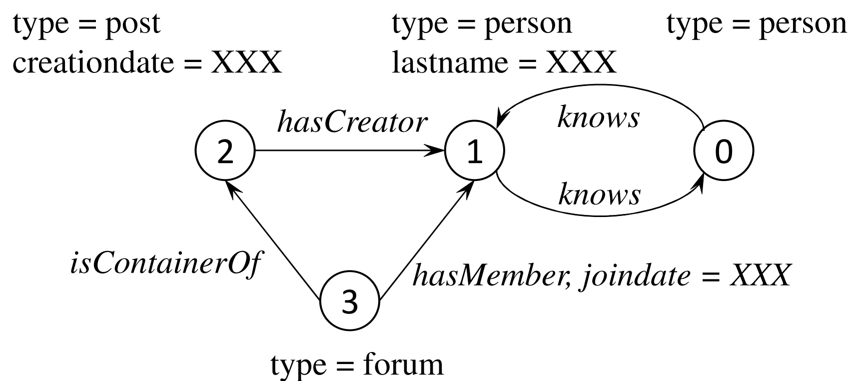
(a) LDBC QUERY Q_1



(b) LDBC QUERY Q_2

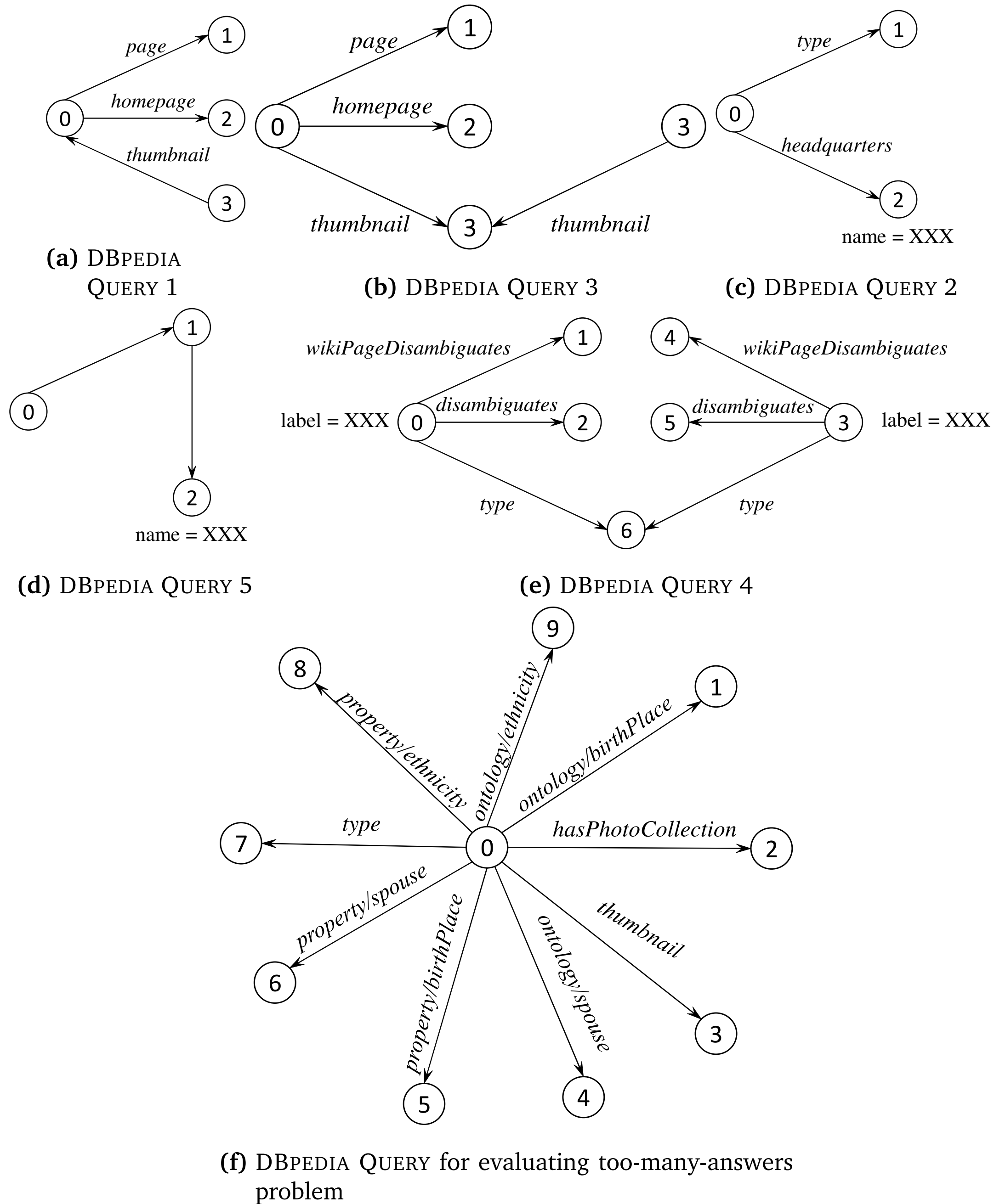


(c) LDBC QUERY Q_3



(d) LDBC QUERY Q_4

Figure A.2: Evaluated queries for LDBC data graph



(a) DBpedia Query 1

(b) DBpedia Query 3

(c) DBpedia Query 2

(d) DBpedia Query 5

(e) DBpedia Query 4

(f) DBpedia Query for evaluating too-many-answers problem

Figure A.3: Evaluated queries for DBpedia graph

B

Additional Evaluation Results

In this section, we present detailed evaluation results for user integration and resource consumption of the why-empty system introduced in Chapter 5.

B.1 User Integration in Why-Empty Query Rewriting

In this section, in Figure B.1, relevance distributions for first 100 refined queries are presented for three LDBC queries which originally fail to deliver any result. These detailed results belong to the evaluation of the user-integration model for coarse-grained modification explanations described in Section 5.5.4. Two different configurations are used: with considering user intention and without it.

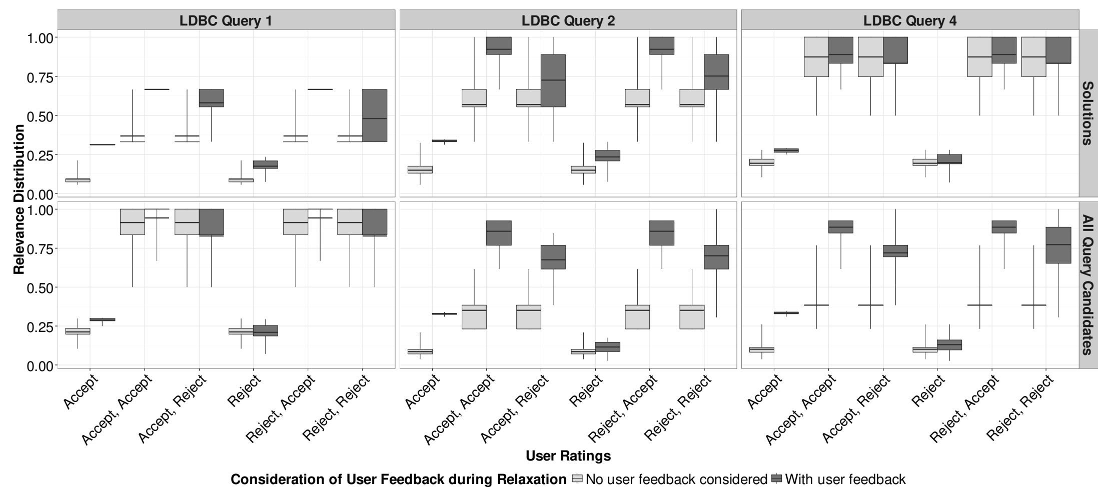


Figure B.1: Evaluation of user integration in the rewriting process for LDBC queries

In Figure B.2, we also present relevances of non-failed rewritten queries generated in first 100 iterations for both experiments: with and without considering the user feedback.

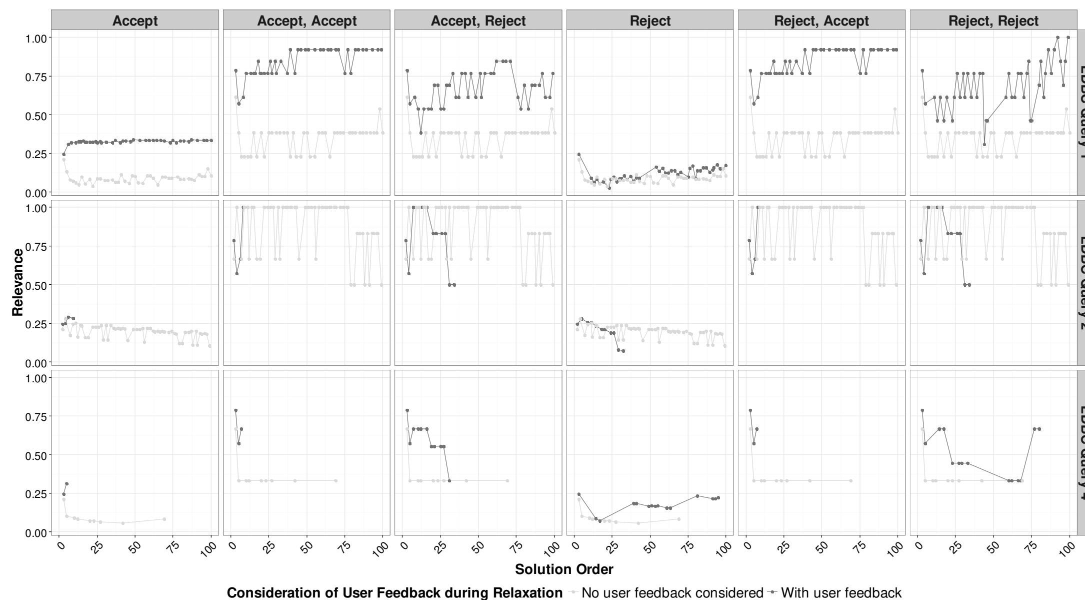


Figure B.2: Evaluation of relevance

B.2 Resource Consumption for Why-Empty Query Rewriting

In Table B.1, we describe the absolute numbers for processing time in ms and size of collected query-dependent statistics, which were collected by evaluating the length of a path to be considered in Section 5.5.3.

			Time (ms)				Statistics Size (B)	
Data Set	Query	Path Size	Total	Statistics Initialize	Statistics Extend	Calculate Paths	Final	Initial
LDBC	QUERY 1	1	41,594	107	1,187	0	4,425	2,356
		2	42,295	108	1,206	21.06	5,312	2,356
		3	41,377	105	1,195	52.8	7,192	2,356
	QUERY 2	1	6,998	98.4	1,756	0	4,074	1,304
		2	6,209	69.5	1,688	14.7	4,710	1,304
		3	6,257	70.7	1,683	15.4	4,710	1,304
	QUERY 3	1	2,779	113	1,201	0	5,185	3,136
		2	2,828	112	1,211	20.1	5,989	3,136
		3	2.917	112	1,195	55.7	8,079	3,136
	QUERY 4	1	15,312	143	2,057	0	4,859	1,828
		2	16,550	174	2,153	29.2	6,146	1,828
		3	15,010	157	2,098	66.8	8,646	1,828
DBPEDIA	QUERY 1	1	35	1.85	14.1	0	1,265	860
		2	36	1.80	17.1	1.7	1,414	860
		3	36	1.86	17.2	1.7	1,414	860
	QUERY 2	1	1,118	2.08	39.5	0	979	660
		2	1,122	2.08	41.8	0.07	1,055	660
		3	1,127	2.11	41.7	0.08	1,055	660
	QUERY 3	1	41	2.65	25.6	0	1,664	1,124
		2	183	2.73	41.1	47.9	1,813	1,124
		3	104	2.65	34.4	52.4	1,921	1,124
	QUERY 4	1	3,150	5.23	282.3	0	3,436	1,780
		2	3,246	5.19	327.3	71.6	3,842	1,780
		3	3,450	5.23	360.9	192.4	4,436	1,780
	QUERY 5	1	8,724	0.78	207.3	0	888	660
		2	10,391	0.78	204.7	2,973.27	964	660
		3	10,389	1.16	180.1	2,989.94	964	660

Table B.1: Memory consumption and response time of why-empty rewriting system for different queries and path lengths

Bibliography

[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] Parag Agrawal, Omar Benjelloun, Anish Das Sarma, Chris Hayworth, Shubha Nabar, Tomoe Sugihara, and Jennifer Widom. Trio: A system for data, uncertainty, and lineage. In *Proceedings of the 32Nd International Conference on Very Large Data Bases*, VLDB '06, pages 1151–1154. VLDB Endowment, 2006.

[3] Rakesh Agrawal, Ralf Rantzau, and Evimaria Terzi. Context-sensitive ranking. In *Proceedings of the 2006 ACM SIGMOD International Conference on Management of Data*, SIGMOD '06, pages 383–394, New York, NY, USA, 2006. ACM.

[4] Sanjay Agrawal, Surajit Chaudhuri, Gautam Das, and Aristides Gionis. Automated ranking of database query results. In *CIDR*, 2003.

[5] Boanerges Aleman-Meza, C Halaschek-Weiner, Cartic Ramakrishnan, Amit P Sheth, et al. Ranking complex relationships on the semantic web. *Internet Computing, IEEE*, 9(3):37–44, 2005.

[6] Laurent Amsaleg, Michael J. Franklin, Anthony Tomasic, and Tolga Urhan. Improving responsiveness for wide-area data access. *IEEE Data Engineering Bulletin*, 20:3–11, 1997.

[7] Kemafor Anyanwu, Angela Maduko, and Amit Sheth. Semrank: Ranking complex relationship search results on the semantic web. In *Proceedings of the 14th International Conference on World Wide Web*, WWW '05, pages 117–127, New York, NY, USA, 2005. ACM.

[8] Akanksha Baid, Wentao Wu, Chong Sun, AnHai Doan, and Jeffrey F. Naughton. On debugging non-answers in keyword search systems. In *Proceedings of the 18th International Conference on Extending Database Technology, EDBT 2015, Brussels, Belgium, March 23-27, 2015.*, pages 37–48, 2015.

[9] Egon Balas and Chang Sung Yu. Finding a maximum clique in an arbitrary graph. *SIAM J. Comput.*, 15(4):1054–1068, November 1986.

[10] Senjuti Basu Roy, Haidong Wang, Gautam Das, Ullas Nambiar, and Mukesh Mohania. Minimum-effort driven dynamic faceted search in structured databases. In *Proceedings of the 17th ACM Conference on Information and Knowledge Management*, CIKM '08, pages 13–22, New York, NY, USA, 2008. ACM.

[11] Mounir Bechchi. *Clustering-based Approximate Answering of Query Result in Large and Distributed Databases*. PhD thesis, Université de Nantes, 2009.

[12] Omar Benjelloun, Anish Das Sarma, Alon Halevy, and Jennifer Widom. Uldbs: Databases with uncertainty and lineage. In *Proceedings of the 32Nd International Conference on Very Large Data Bases*, VLDB '06, pages 953–964. VLDB Endowment, 2006.

[13] Philip A. Bernstein and Thomas Bergstraesser. Meta-data support for data transformations using microsoft repository. *IEEE Data Eng. Bull.*, 22(1):9–14, 1999.

[14] Deepavali Bhagwat, Laura Chiticariu, Wang-Chiew Tan, and Gaurav Vijayvargiya. An annotation management system for relational databases. In *Proceedings of the Thirtieth International Conference on Very Large Data Bases - Volume 30*, VLDB '04, pages 900–911. VLDB Endowment, 2004.

[15] Nicole Bidoit, Melanie Herschel, and Aikaterini Tzompanaki. Efficient computation of polynomial explanations of why-not questions. In *Proceedings of the 24th ACM International on Conference on Information and Knowledge Management*, CIKM '15, pages 713–722, New York, NY, USA, 2015. ACM.

[16] Nicole Bidoit, Melanie Herschel, and Katerina Tzompanaki. Immutably answering why-not questions for equivalent conjunctive queries. In *6th USENIX Workshop on the Theory and Practice of Provenance (TaPP 2014)*, Cologne, June 2014. USENIX Association.

[17] Nicole Bidoit, Melanie Herschel, and Katerina Tzompanaki. Query-Based Why-Not Provenance with NedExplain. In *Proceedings of the 17th International Conference on Extending Database Technology, EDBT 2014, Athens, Greece, March 24-28, 2014.*, pages 145–156, 2014.

[18] Nicole Bidoit, Melanie Herschel, and Katerina Tzompanaki. Efficient computation of polynomial explanations of Why-Not questions. In *31ème Conférence sur la Gestion de Données - Principes, Technologies et Applications - BDA 2015*, Île de Porquerolles, France, September 2015.

[19] Nicole Bidoit, Melanie Herschel, and Katerina Tzompanaki. Efq: Why-not answer polynomials in action. *Proc. VLDB Endow.*, 8(12):1980–1983, August 2015.

[20] Peter Boncz. Ldbc: Benchmarks for graph and rdf data management. In *Proceedings of the 17th International Database Engineering & Applications Symposium*, IDEAS '13, pages 1–2, New York, NY, USA, 2013. ACM.

[21] Christof Bornhövd, Robert Kubis, Wolfgang Lehner, Hannes Voigt, and Horst Werner. Flexible information management, exploration and analysis in SAP HANA. In *DATA 2012 - Proceedings of the International Conference on Data Technologies and Applications, Rome, Italy, 25-27 July, 2012*, pages 15–28, 2012.

[22] Peter Buneman, Alin Deutsch, and Wang-Chiew Tan. A deterministic model for semistructured data. In *Workshop on Query Processing for Semistructured Data and Non-Standard Data Formats*, pages 14–19, 1999.

[23] Peter Buneman, Sanjeev Khanna, and Wang Chiew Tan. Why and where: A characterization of data provenance. In *Proceedings of the 8th International Conference on Database Theory*, ICDT '01, pages 316–330, London, UK, UK, 2001. Springer-Verlag.

[24] Peter Buneman, Sanjeev Khanna, and Wang-Chiew Tan. On propagation of deletions and annotations through views. In *Proceedings of the Twenty-first ACM SIGMOD-SIGACT-SIGART Symposium on Principles of Database Systems*, PODS '02, pages 150–158, New York, NY, USA, 2002. ACM.

[25] Rafael Caballero, Yolanda García-Ruiz, and Fernando Sáenz-Pérez. Declarative debugging of wrong and missing answers for sql views. In Tom Schrijvers and Peter Thiemann, editors, *Functional and Logic Programming*, volume 7294 of *Lecture Notes in Computer Science*, pages 73–87. Springer Berlin Heidelberg, 2012.

[26] Michael J. Cafarella, Christopher Re, Dan Suciu, Oren Etzioni, and Michele Banko. Structured querying of web text. In *3rd Biennial Conference on Innovative Data Systems Research (CIDR), Asilomar, California, USA*, 2007.

[27] Andrea Cali, Domenico Lembo, and Riccardo Rosati. Query rewriting and answering under constraints in data integration systems. In *Proceedings of the 18th International Joint Conference on Artificial Intelligence*, IJCAI'03, pages 16–21, San Francisco, CA, USA, 2003. Morgan Kaufmann Publishers Inc.

[28] Diego Calvanese, Magdalena Ortiz, Mantas Simkus, and Giorgio Stefanoni. The complexity of explaining negative query answers in dl-lite. In *Proc. of the 13th Int. Conf. on the Principles of Knowledge Representation and Reasoning (KR 2012)*, pages 583–587. AAAI Press, 2012.

[29] Diego Calvanese, Magdalena Ortiz, Mantas Simkus, and Giorgio Stefanoni. Reasoning about explanations for negative query answers in dl-lite. *J. Artif. Intell. Res. (JAIR)*, 48:635–669, 2013.

[30] Kaushik Chakrabarti, Surajit Chaudhuri, and Seung-won Hwang. Automatic categorization of query results. In *Proceedings of the 2004 ACM SIGMOD International Conference on Management of Data*, SIGMOD '04, pages 755–766, New York, NY, USA, 2004. ACM.

[31] Hans Chalupsky and Thomas A. Russ. Whynot: Debugging failed queries in large knowledge bases. In *Proceedings of the Eighteenth National Conference on Artificial Intelligence and Fourteenth Conference on Innovative Applications of Artificial Intelligence, July 28 - August 1, 2002, Edmonton, Alberta, Canada.*, pages 870–877, 2002.

[32] 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.

[33] Surajit Chaudhuri. Generalization and a framework for query modification. In *Proceedings of the Sixth International Conference on Data Engineering*, pages 138–145, Washington, DC, USA, 1990. IEEE Computer Society.

[34] Surajit Chaudhuri, Gautam Das, Vagelis Hristidis, and Gerhard Weikum. Probabilistic ranking of database query results. In *Proceedings of the Thirtieth International Conference on Very Large Data Bases - Volume 30*, VLDB '04, pages 888–899. VLDB Endowment, 2004.

[35] Surajit Chaudhuri, Gautam Das, Vagelis Hristidis, and Gerhard Weikum. Probabilistic information retrieval approach for ranking of database query results. *ACM Trans. Database Syst.*, 31(3):1134–1168, September 2006.

[36] Lei Chen, Xin Lin, Haibo Hu, Christian S Jensen, and Jianliang Xu. Answering why-not questions on spatial keyword top-k queries. In *Data Engineering (ICDE), 2015 IEEE 31st International Conference on*, pages 279–290. IEEE, 2015.

[37] Zhiyuan Chen and Tao Li. Addressing diverse user preferences: A framework for query results navigation. *IEEE Data Eng. Bull.*, 32(4):41–48, 2009.

[38] Zhiyuan Chen, Tao Li, and Yanan Sun. A learning approach to sql query results ranking using skyline and users' current navigational behavior. *IEEE Trans. on Knowl. and Data Eng.*, 25(12):2683–2693, December 2013.

[39] Sean Chester and Ira Assent. Explanations for skyline query results. In *Proceedings of the 18th International Conference on Extending Database Technology, EDBT 2015, Brussels, Belgium, March 23-27, 2015.*, pages 349–360, 2015.

[40] Laura Chiticariu, Wang-Chiew Tan, and Gaurav Vijayvargiya. Dbnotes: A post-it system for relational databases based on provenance. In *Proceedings of the 2005 ACM SIGMOD International Conference on Management of Data*, SIGMOD '05, pages 942–944, New York, NY, USA, 2005. ACM.

[41] Wesley W. Chu, Hua Yang, Kuorong Chiang, Michael Minock, Gladys Chow, and Chris Larson. Cobase: A scalable and extensible cooperative information system. *Journal of Intelligent Information Systems*, 6(2-3):223–259, 1996.

[42] Thomas H. Cormen, Clifford Stein, Ronald L. Rivest, and Charles E. Leiserson. *Introduction to Algorithms*. McGraw-Hill Higher Education, 2nd edition, 2001.

[43] Yingwei Cui and Jennifer Widom. Lineage tracing for general data warehouse transformations. *The VLDB Journal*, 12(1):41–58, May 2003.

[44] Yingwei Cui, Jennifer Widom, and Janet L. Wiener. Tracing the lineage of view data in a warehousing environment. *ACM Trans. Database Syst.*, 25(2):179–227, June 2000.

[45] 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.

[46] Paul J Durand, Rohit Pasari, Johnnie W Baker, and Chun-che Tsai. An efficient algorithm for similarity analysis of molecules. *Internet Journal of Chemistry*, 2(17):1–16, 1999.

[47] Efthimis N Efthimiadis. Query expansion. *Annual review of information science and technology*, 31:121–187, 1996.

[48] Wenfei Fan, Xin Wang, and Yinghui Wu. Diversified top-k graph pattern matching. *Proc. VLDB Endow.*, 6(13):1510–1521, August 2013.

[49] Géraud Fokou, Stéphane Jean, and Allel Hadjali. Endowing semantic query languages with advanced relaxation capabilities. In Troels Andreasen, Henning

Christiansen, Juan-Carlos Cubero, and ZbigniewW. Raś, editors, *Foundations of Intelligent Systems*, volume 8502 of *Lecture Notes in Computer Science*, pages 512–517. Springer International Publishing, 2014.

[50] Xinbo Gao, Bing Xiao, Dacheng Tao, and Xuelong Li. A survey of graph edit distance. *Pattern Anal. Appl.*, 13(1):113–129, January 2010.

[51] Yunjun Gao, Qing Liu, Gang Chen, Baihua Zheng, and Linlin Zhou. Answering why-not questions on reverse top-k queries. *Proc. VLDB Endow.*, 8(7):738–749, February 2015.

[52] Boris Glavic, Sven Köhler, Sean Riddle, and Bertram Ludäscher. Towards constraint-based explanations for answers and non-answers. In *Proceedings of the 7th USENIX Conference on Theory and Practice of Provenance*, TaPP'15, pages 13–13, Berkeley, CA, USA, 2015. USENIX Association.

[53] Parke Godfrey. Minimization in cooperative response to failing database queries. *International Journal of Cooperative Information Systems*, 6(02):95–149, 1997.

[54] Todd J. Green. Containment of conjunctive queries on annotated relations. In *Proceedings of the 12th International Conference on Database Theory*, ICDT '09, pages 296–309, New York, NY, USA, 2009. ACM.

[55] Todd J. Green, Grigoris Karvounarakis, and Val Tannen. Provenance semirings. In *Proceedings of the Twenty-sixth ACM SIGMOD-SIGACT-SIGART Symposium on Principles of Database Systems*, PODS '07, pages 31–40, New York, NY, USA, 2007. ACM.

[56] Torsten Grust and Jan Rittinger. Observing sql queries in their natural habitat. *ACM Trans. Database Syst.*, 38(1):3:1–3:33, April 2013.

[57] Steve Harris, Andy Seaborne, and Eric Prudh́ommeaux. Sparql 1.1 query language. *W3C Recommendation*, 21, 2013.

[58] Zhian He and Eric Lo. Answering why-not questions on top-k queries. In *IEEE 28th International Conference on Data Engineering (ICDE 2012), Washington, DC, USA (Arlington, Virginia), 1-5 April, 2012*, pages 750–761, 2012.

[59] Zhian He and Eric Lo. Answering why-not questions on top-k queries. *IEEE Trans. Knowl. Data Eng.*, 26(6):1300–1315, 2014.

[60] Melanie 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 & knowledge management*, CIKM '13, pages 2213–2218, New York, NY, USA, 2013. ACM.

[61] Melanie Herschel. A hybrid approach to answering why-not questions on relational query results. *J. Data and Information Quality*, 5(3):10:1–10:29, March 2015.

[62] Melanie Herschel and Hanno Eichelberger. The nautilus analyzer: Understanding and debugging data transformations. In *Proceedings of the 21st ACM International Conference on Information and Knowledge Management*, CIKM '12, pages 2731–2733, New York, NY, USA, 2012. ACM.

[63] Melanie Herschel and Torsten Grust. Transformation lifecycle management with nautilus. In *VLDB Workshop on the Quality of Data (QDB)*, 2011.

[64] Melanie Herschel and Mauricio A. Hernández. Explaining missing answers to spjua queries. *Proc. VLDB Endow.*, 3(1-2):185–196, September 2010.

[65] Melanie Herschel, Mauricio A. Hernández, and Wang-Chiew Tan. Artemis: A system for analyzing missing answers. *Proc. VLDB Endow.*, 2(2):1550–1553, August 2009.

[66] Hai Huang, Chengfei Liu, and Xiaofang Zhou. Approximating query answering on rdf databases. *World Wide Web*, 15(1):89–114, 2012.

[67] Jiansheng Huang, Ting Chen, AnHai Doan, and Jeffrey F. Naughton. On the provenance of non-answers to queries over extracted data. *Proc. VLDB Endow.*, 1(1):736–747, August 2008.

[68] Y. Huhtala, J. Karkkainen, P. Porkka, and H. Toivonen. Efficient discovery of functional and approximate dependencies using partitions. In *Data Engineering, 1998. Proceedings., 14th International Conference on*, pages 392–401, Feb 1998.

[69] Ihab F. Ilyas, George Beskales, and Mohamed A. Soliman. A survey of top-k query processing techniques in relational database systems. *ACM Comput. Surv.*, 40(4):11:1–11:58, October 2008.

[70] Tomasz Imieliński and Witold Lipski, Jr. Incomplete information in relational databases. *J. ACM*, 31(4):761–791, September 1984.

[71] Md. Saiful Islam. On answering why and why-not questions in databases. In *Workshops Proceedings of the 29th IEEE International Conference on Data Engineering, ICDE 2013, Brisbane, Australia, April 8-12, 2013*, pages 298–301, 2013.

[72] 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.

[73] Md. Saiful Islam, Chengfei Liu, and Rui Zhou. On modeling query refinement by capturing user intent through feedback. In *Proceedings of the Twenty-Third Australasian Database Conference - Volume 124*, ADC '12, pages 11–20, Darlinghurst, Australia, Australia, 2012. Australian Computer Society, Inc.

[74] Md. Saiful Islam, Chengfei Liu, and Rui Zhou. A framework for query refinement with user feedback. *J. Syst. Softw.*, 86(6):1580–1595, June 2013.

[75] Md. Saiful Islam, Chengfei Liu, and Rui Zhou. Flexiq: A flexible interactive querying framework by exploiting the skyline operator. *Journal of Systems and Software*, 97:97–117, 2014.

[76] Md. Saiful Islam, Rui Zhou, and Chengfei Liu. On answering why-not questions in reverse skyline queries. In *Data Engineering (ICDE), 2013 IEEE 29th International Conference on*, pages 973–984. IEEE, 2013.

[77] 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.

[78] Dietmar Jannach. Techniques for fast query relaxation in content-based recommender systems. In *Proceedings of the 29th Annual German Conference on Artificial Intelligence*, KI'06, pages 49–63, Berlin, Heidelberg, 2007. Springer-Verlag.

[79] Ulrich Junker. Quickxplain: Preferred explanations and relaxations for over-constrained problems. In *Proceedings of the 19th National Conference on Artifical Intelligence*, AAAI'04, pages 167–172. AAAI Press, 2004.

[80] S. Jerrold Kaplan. Cooperative responses from a portable natural language query system. *Artificial Intelligence*, 19(2):165 – 187, 1982.

[81] S. Jerrold Kaplan. Designing a portable natural language database query system. *ACM Trans. Database Syst.*, 9(1):1–19, March 1984.

[82] Nishant Kapoor, Gautam Das, Vagelis Hristidis, S. Sudarshan, and Gerhard Weikum. STAR: A system for tuple and attribute ranking of query answers. In *Data Engineering, 2007. ICDE 2007. IEEE 23rd International Conference on*, pages 1483–1484, April 2007.

[83] Grigoris Karvounarakis and Todd J. Green. Semiring-annotated data: Queries and provenance? *SIGMOD Rec.*, 41(3):5–14, October 2012.

[84] Abhijith Kashyap, Vagelis Hristidis, and Michalis Petropoulos. Facetor: Cost-driven exploration of faceted query results. In *Proceedings of the 19th ACM International Conference on Information and Knowledge Management*, CIKM '10, pages 719–728, New York, NY, USA, 2010. ACM.

[85] Sven Köhler, Bertram Ludäscher, and Daniel Zinn. First-order provenance games. In Val Tannen, Limsoon Wong, Leonid Libkin, Wenfei Fan, Wang-Chiew Tan, and Michael Fourman, editors, *In Search of Elegance in the Theory and Practice of Computation*, volume 8000 of *Lecture Notes in Computer Science*, pages 382–399. Springer Berlin Heidelberg, 2013.

[86] Werner Kießling. Foundations of preferences in database systems. In *Proceedings of the 28th International Conference on Very Large Data Bases*, VLDB '02, pages 311–322. VLDB Endowment, 2002.

[87] Nick Koudas, Chen Li, Anthony K. H. Tung, and Rares Vernica. Relaxing join and selection queries. In *Proceedings of the 32Nd International Conference on Very Large Data Bases*, VLDB '06, pages 199–210. VLDB Endowment, 2006.

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

[89] Thomas Lee, Stéphane Bressan, and Stuart E Madnick. Source attribution for querying against semi-structured documents. In *Workshop on Web Information and Data Management*, pages 33–39, 1998.

[90] Alon Y. Levy. Obtaining complete answers from incomplete databases. In *Proceedings of the 22th International Conference on Very Large Data Bases*, VLDB '96, pages 402–412, San Francisco, CA, USA, 1996. Morgan Kaufmann Publishers Inc.

[91] Bin Liu and H. V. Jagadish. Using trees to depict a forest. *Proc. VLDB Endow.*, 2(1):133–144, August 2009.

[92] Federica Mandreoli, Riccardo Martoglia, Giorgio Villani, and Wilma Penzo. Flexible query answering on graph-modeled data. In *Proceedings of the 12th International Conference on Extending Database Technology: Advances in Database Technology*, EDBT '09, pages 216–227, New York, NY, USA, 2009. ACM.

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

[94] James J. McGregor. Backtrack search algorithms and the maximal common subgraph problem. *Software: Practice and Experience*, 12(1):23–34, 1982.

[95] Alexandra Meliou, Wolfgang Gatterbauer, Joseph Y Halpern, Christoph Koch, Katherine F Moore, and Dan Suciu. Causality in databases. *IEEE Data Eng. Bull.*, 33(EPFL-ARTICLE-165841):59–67, 2010.

[96] Alexandra Meliou, Wolfgang Gatterbauer, Katherine F. Moore, and Dan Suciu. The complexity of causality and responsibility for query answers and non-answers. *Proc. VLDB Endow.*, 4(1):34–45, October 2010.

[97] Sergey Melnik, Hector Garcia-Molina, and Erhard Rahm. Similarity flooding: a versatile graph matching algorithm and its application to schema matching. In *Data Engineering, 2002. Proceedings. 18th International Conference on*, pages 117–128, 2002.

[98] Eduardo Mena, Vipul Kashyap, Arantza Illarramendi, and Amit Sheth. Imprecise answers in distributed environments: Estimation of information loss for multi-ontology based query processing. *International Journal of Cooperative Information Systems*, 09(04):403–425, 2000.

[99] Matthew Merzbacher and Wesley W. Chu. Pattern–based clustering for database attribute values. In *in Proceedings of AAAI Workshop on Knowledge Discovery*, 1993.

[100] Eric Miller. An introduction to the resource description framework. *Bulletin of the American Society for Information Science and Technology*, 25(1):15–19, 1998.

[101] Chaitanya Mishra and Nick Koudas. Stretch 'n' shrink: Resizing queries to user preferences. In *Proceedings of the 2008 ACM SIGMOD International Conference on Management of Data*, SIGMOD '08, pages 1227–1230, New York, NY, USA, 2008. ACM.

[102] Chaitanya Mishra and Nick Koudas. Interactive query refinement. In *Proceedings of the 12th International Conference on Extending Database Technology: Advances in Database Technology*, EDBT '09, pages 862–873, New York, NY, USA, 2009. ACM.

[103] Amihai Motro. Query generalization: A method for interpreting null answers. In *Proceedings from the First International Workshop on Expert Database Systems*, pages 597–616, Redwood City, CA, USA, 1986. Benjamin-Cummings Publishing Co., Inc.

[104] Amihai Motro. SEAVE: A Mechanism for Verifying User Presuppositions in Query Systems. *ACM Trans. Inf. Syst.*, 4(4):312–330, December 1986.

[105] Amihai Motro. FLEX: a tolerant and cooperative user interface to databases. *Knowledge and Data Engineering, IEEE Transactions on*, 2(2):231–246, Jun 1990.

[106] 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.

[107] Davide Mottin, Alice Marascu, Senjuti Basu Roy, Gautam Das, Themis Palpanas, and Yannis Velegrakis. Iqr: An interactive query relaxation system for the empty-answer problem. In *Proceedings of the 2014 ACM SIGMOD International Conference on Management of Data*, SIGMOD '14, pages 1095–1098, New York, NY, USA, 2014. ACM.

[108] 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.

[109] Ion Muslea. Machine learning for online query relaxation. In *Proceedings of the Tenth ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, KDD '04, pages 246–255, New York, NY, USA, 2004. ACM.

[110] Ion Muslea and Thomas J. Lee. Online query relaxation via bayesian causal structures discovery. In *Proceedings of the 20th National Conference on Artificial Intelligence - Volume 2*, AAAI'05, pages 831–836. AAAI Press, 2005.

[111] Ullas Nambiar and Subbarao Kambhampati. Mining approximate functional dependencies and concept similarities to answer imprecise queries. In *Proceedings of the 7th International Workshop on the Web and Databases: Colocated with ACM SIGMOD/PODS 2004*, WebDB '04, pages 73–78, New York, NY, USA, 2004. ACM.

[112] Xiaomin Ning, Hai Jin, and Hao Wu. Rss: A framework enabling ranked search on the semantic web. *Inf. Process. Manage.*, 44(2):893–909, March 2008.

[113] Michael Ortega-Binderberger, Kaushik Chakrabarti, and Sharad Mehrotra. An approach to integrating query refinement in sql. In ChristianS. Jensen, Simonas Šaltenis, KeithG. Jeffery, Jaroslav Pokorny, Elisa Bertino, Klemens Böhn, and Matthias Jarke, editors, *Advances in Database Technology — EDBT 2002*, volume 2287 of *Lecture Notes in Computer Science*, pages 15–33. Springer Berlin Heidelberg, 2002.

[114] Marcus Paradies, Wolfgang Lehner, and Christof Bornhövd. Graphite: An extensible graph traversal framework for relational database management systems. In *Proceedings of the 27th International Conference on Scientific and Statistical Database Management*, SSDBM '15, pages 29:1–29:12, New York, NY, USA, 2015. ACM.

[115] Marcus Paradies, Elena Vasilyeva, Adrian Mocan, and Wolfgang Lehner. Robust cardinality estimation for subgraph isomorphism queries on property graphs. In *Big-O(Q) 2015 (co-located with VLDB 2015)*, 2015.

[116] 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.

[117] 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.

[118] Arnau Prat, Peter Boncz, Josep Lluìs Larriba, Renzo Angles, Alex Averbuch, Orri Erling, Andrey Gubichev, Mirko Spasic, Minh-Duc Pham, and Norbert Martìnez. Ldbc social network benchmark (snb) - v0.2.2 first public draft release v0.2.2. Technical report, 2015.

[119] 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.

[120] Cartic Ramakrishnan, William H. Milnor, Matthew Perry, and Amit P. Sheth. Discovering informative connection subgraphs in multi-relational graphs. *SIGKDD Explor. Newsl.*, 7(2):56–63, December 2005.

[121] Sean Riddle, Sven Köhler, and Bertram Ludäscher. Towards constraint provenance games. In *6th USENIX Workshop on the Theory and Practice of Provenance (TaPP 2014)*, Cologne, June 2014. USENIX Association.

[122] 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.

[123] Senjuti Basu Roy, Haidong Wang, Ullas Nambiar, Gautam Das, and Mukesh Mohania. DynaCet: Building Dynamic Faceted Search Systems over Databases. In *Data Engineering, 2009. ICDE '09. IEEE 25th International Conference on*, pages 1463–1466, March 2009.

[124] Michael Rudolf, Marcus Paradies, Christof Bornhövd, and Wolfgang Lehner. The graph story of the SAP HANA database. In *Datenbanksysteme für Business, Technologie und Web (BTW), 15. Fachtagung des GI-Fachbereichs "Datenbanken und Informationssysteme" (DBIS), 11.-15.3.2013 in Magdeburg, Germany. Proceedings*, pages 403–420, 2013.

[125] Nikos Sarkas, Nilesh Bansal, Gautam Das, and Nick Koudas. Measure-driven keyword-query expansion. *Proc. VLDB Endow.*, 2(1):121–132, August 2009.

[126] Kostas Stefanidis, Evaggelia Pitoura, and Panos Vassiliadis. A context-aware preference database system. *Int. J. Pervasive Computing and Communications*, 3(4):439–460, 2007.

[127] Weifeng Su, Jiying Wang, Qiong Huang, and Fred Lochovsky. Query result ranking over e-commerce web databases. In *Proceedings of the 15th ACM International Conference on Information and Knowledge Management*, CIKM '06, pages 575–584, New York, NY, USA, 2006. ACM.

[128] Wang-Chiew Tan. Containment of relational queries with annotation propagation. In *Database Programming Languages*, pages 37–53. Springer, 2004.

[129] Tao Tao and ChengXiang Zhai. Best-k queries on database systems. In *Proceedings of the 15th ACM International Conference on Information and Knowledge Management*, CIKM '06, pages 790–791, New York, NY, USA, 2006. ACM.

[130] Aditya Telang, Chengkai Li, and Sharma Chakravarthy. One size does not fit all: Toward user- and query-dependent ranking for web databases. *IEEE Trans. on Knowl. and Data Eng.*, 24(9):1671–1685, September 2012.

[131] Balder ten Cate, Cristina Civili, Evgeny Sherkhonov, and Wang-Chiew Tan. High-level why-not explanations using ontologies. In *Proceedings of the 34th ACM Symposium on Principles of Database Systems*, PODS '15, pages 31–43, New York, NY, USA, 2015. ACM.

[132] Quoc Trung Tran and Chee-Yong Chan. How to conquer why-not questions. In *Proceedings of the 2010 ACM SIGMOD International Conference on Management of Data*, SIGMOD '10, pages 15–26, New York, NY, USA, 2010. ACM.

[133] Quoc Trung Tran, Chee-Yong Chan, and Srinivasan Parthasarathy. Query by output. In *Proceedings of the 2009 ACM SIGMOD International Conference on Management of Data*, SIGMOD '09, pages 535–548, New York, NY, USA, 2009. ACM.

[134] Thanh Tran, Haofen Wang, Sebastian Rudolph, and Philipp Cimiano. Top-k exploration of query candidates for efficient keyword search on graph-shaped (rdf) data. In *Proceedings of the 2009 IEEE International Conference on Data Engineering*, ICDE '09, pages 405–416, Washington, DC, USA, 2009. IEEE Computer Society.

[135] Julian R. Ullmann. An algorithm for subgraph isomorphism. *Journal of the ACM (JACM)*, 23(1):31–42, January 1976.

[136] Arthur H Van Bunningen, Maarten M Fokkinga, Peter MG Apers, and Ling Feng. Ranking query results using context-aware preferences. In *Data Engineering Workshop, 2007 IEEE 23rd International Conference on*, pages 269–276. IEEE, 2007.

[137] Manasi Vartak, Venkatesh Raghavan, and Elke A. Rundensteiner. Qrelx: Generating meaningful queries that provide cardinality assurance. In *Proceedings of the 2010 ACM SIGMOD International Conference on Management of Data*, SIGMOD '10, pages 1215–1218, New York, NY, USA, 2010. ACM.

[138] Elena Vasilyeva. Why-query support in graph databases. In *2016 IEEE 32nd International Conference on Data Engineering Workshops (ICDEW)*, pages 221–225, May 2016.

[139] Elena Vasilyeva, Thomas Heinze, Maik Thiele, and Wolfgang Lehner. DebEAQ - debugging empty-answer queries on large data graphs. In *2016 IEEE 32nd International Conference on Data Engineering (ICDE)*, pages 1402–1405, May 2016.

[140] Elena Vasilyeva, Maik Thiele, Christof Bornhövd, and Wolfgang Lehner. GraphMCS: Discover the Unknown in Large Data Graphs. In *Proceedings of the Workshops of the EDBT/ICDT 2014 Joint Conference (EDBT/ICDT 2014), Athens, Greece, March 28, 2014.*, pages 200–207, 2014.

[141] Elena Vasilyeva, Maik Thiele, Christof Bornhövd, and Wolfgang Lehner. Answering "Why Empty?" and "Why So Many?" queries in graph databases. *Journal of Computer and System Sciences*, 82(1):3–22, 2016.

[142] Elena Vasilyeva, Maik Thiele, Christof Bornhövd, and Wolfgang Lehner. Top-k differential queries in graph databases. In Yannis Manolopoulos, Goce Trajcevski, and Margita Kon-Popovska, editors, *Advances in Databases and Information Systems*, volume 8716 of *Lecture Notes in Computer Science*, pages 112–125. Springer International Publishing, 2014.

[143] Elena Vasilyeva, Maik Thiele, Adrian Mocan, and Wolfgang Lehner. Relaxation of subgraph queries delivering empty results. In *Proceedings of the 27th International Conference on Scientific and Statistical Database Management*, SSDBM '15, pages 28:1–28:12, New York, NY, USA, 2015. ACM.

[144] 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.

[145] Marcos R. Vieira, Humberto L. Razente, Maria C. N. Barioni, Marios Hadjieleftheriou, Divesh Srivastava, Caetano Traina, and Vassilis J. Tsotras. On query result diversification. In *Proceedings of the 2011 IEEE 27th International Conference on Data Engineering*, ICDE '11, pages 1163–1174, Washington, DC, USA, 2011. IEEE Computer Society.

[146] Aleksa Vukotic, Nicki Watt, Tareq Abedrabbo, Dominic Fox, and Jonas Partner. *Neo4j in Action*. Manning, 2015.

[147] Y. Richard Wang and Stuart E. Madnick. A polygon model for heterogeneous database systems: The source tagging perspective. In *Proceedings of the Sixteenth International Conference on Very Large Databases*, pages 519–533, San Francisco, CA, USA, 1990. Morgan Kaufmann Publishers Inc.

[148] Jennifer Widom. Trio: A system for integrated management of data, accuracy, and lineage. In *CIDR*, pages 262–276, 2005.

[149] Garrett Wolf, Aravind Kalavagattu, Hemal Khatri, Raju Balakrishnan, Bhaumik Chokshi, Jianchun Fan, Yi Chen, and Subbarao Kambhampati. Query processing over incomplete autonomous databases: Query rewriting using learned data dependencies. volume 18, pages 1167–1190, Secaucus, NJ, USA, October 2009. Springer-Verlag New York, Inc.

[150] 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.

[151] Xintao Wu and Daniel Barbará. Learning missing values from summary constraints. *SIGKDD Explor. Newsl.*, 4(1):21–30, June 2002.

[152] Shengqi Yang, Yinghui Wu, Huan Sun, and Xifeng Yan. Schemaless and structureless graph querying. *Proc. VLDB Endow.*, 7(7):565–576, March 2014.

[153] Siyu Yao, Jun Liu, Meng Wang, Bifan Wei, and Xuelu Chen. ANNA: answering why-not questions for SPARQL. In *Proceedings of the ISWC 2015 Posters & Demonstrations Track co-located with the 14th International Semantic Web Conference (ISWC-2015), Bethlehem, PA, USA, October 11, 2015.*, 2015.

[154] Elad Yom-Tov, Shai Fine, David Carmel, and Adam Darlow. Learning to estimate query difficulty: Including applications to missing content detection and distributed information retrieval. In *Proceedings of the 28th Annual International ACM SIGIR Conference on Research and Development in Information Retrieval*, SIGIR '05, pages 512–519, New York, NY, USA, 2005. ACM.

[155] Lei Zou, Lei Chen, and Yansheng Lu. Top-k subgraph matching query in a large graph. In *Proceedings of the ACM First Ph.D. Workshop in CIKM*, PIKM '07, pages 139–146, New York, NY, USA, 2007. ACM.

List of Figures

1.1 Thesis outline 5

2.1 Overview of why-queries 8
2.2 Provenance hierarchy (Source: Green et al. [55]) 10
2.3 Example of provenance calculation (Source: Karvounarakis et al. [83]) . 11
2.4 Query-based solution for why-not query: Why is family Clarke not in result set? Original query: SELECT Surname FROM Family AS F INNER JOIN Apartments AS A ON F.Surname=A.Surname WHERE F.Children = 3 AND A.Flat < 5 15
2.5 Example for database and query tree (Source: Bidoit et.al [17]) 16
2.6 Why-not provenance for *3hop(c,a)* using provenance games (Source: Riddle et al. [121]) 19
2.7 Example of database table with train connections and ontology (Source: ten Cate et al. [131]) 21
2.8 Example of query representation in Meta Query Language 26
2.9 Query relaxation lattice 27
2.10 Classification of top-k solutions for querying RDBMS (Source: Ilyas et al. [69]) 32

3.1 Holistic support of different cardinality-based problems 45
3.2 Classification of complex modification operations 48
3.3 Set-based query model, where ovals define operations such as union and disjunctions of values and rectangles describe graph elements and their properties. For reasons of simplicity, all elements are drawn only once and dashed lines represent multiplicity of relations, for example: vertex set can consist of multiple vertices 49
3.4 Point-set distance $d(a_i, B)$ 51
3.5 Example query and its modification-based explanation 54
3.6 Example of two result subgraphs for calculating result distance 55
3.7 Ordered syntactic distances for randomly generated explanations 59
3.8 Ordered result distances for randomly generated explanations 60
3.9 Ordered cardinality distances for randomly generated explanations . . . 61
3.10 Distribution of average result distances for randomly generated explanations 62

4.1 Example of why-empty query, data graph, and corresponding partial answers 66
4.2 Depth-first search 67

4.3 Original query delivering empty result and its subgraph-based explanation: which two vertices are from same country and play in same club? . 70
4.4 Why-so-many query and its answer: which two players of same nationality play in different clubs? 72
4.5 Weakly connected graphs 73
4.6 All-covering spanning tree and backtracking procedure 75
4.7 Example of missing largest subgraph caused by splitting query in unreachable components and traversal in its smaller part 76
4.8 Why-empty query example with user preferences 78
4.9 Relevance-based search 79
4.10 Relevance flooding: gray relevance weights correspond to non-resetted weights 81
4.11 Comparison of heuristics for choosing single traversal path for DiscoverMCS 84
4.12 Relative change of response time for DiscoverMCS algorithm with construction of spanning tree 85
4.13 Relative response time of DiscoverMCS algorithm for restart with minimal-cardinality heuristic 86
4.14 Evaluation results of DiscoverMCS algorithm for restart with minimal in-degree heuristic 87
4.15 Evaluation results of BoundedMCS algorithm with single traversal for LDBC data graph 88
4.16 Evaluation results of BoundedMCS algorithm with single traversal for DBpedia data graph 88

5.1 System architecture for why-empty query rewriting 96
5.2 Example query and its query-dependent statistics 98
5.3 Configuration masks for vertices, edges, and paths(1) 99
5.4 Example of vertex bitvector calculation 101
5.5 Examples for path calculations 104
5.6 Placement of new queries in priority queue 105
5.7 Hierarchy of induced cardinality changes 106
5.8 Influence of vertex deletion on degrees of neighboring vertices 107
5.9 Example query delivering empty result and its two explanations 110
5.10 Average rank for priority functions 113
5.11 Evaluation results for different priority strategies across evaluated queries 115
5.12 Evaluation results for top-3 priority functions 116
5.13 Syntactic distance of first five discovered explanations 117
5.14 Relative change of number of iterations for path(2) and path(3) with respect to path(1) 118
5.15 Time distribution for evaluated queries 120
5.16 Relevance distribution of discovered explanations 122

6.1 Modification process for why-so-few and why-so-many queries 126
6.2 Example query and its operational representations 128
6.3 Construction of modification tree for running example 130
6.4 Interval modification 135
6.5 Topological removal for operator $FP(e_2, 1)$ 137
6.6 Acquisition of neighbors for leaves $GV(v_1)$ and $GV(v_2)$ 138
6.7 Subtree removal from modification tree 139
6.8 Baseline comparison: distribution and percentage of refined queries . . . 142

6.9 Baseline comparison: distribution of cardinality distance 144
6.10 Baseline comparison: distribution of result and syntactic distances 145
6.11 Baseline comparison: distribution of normalized number of iterations . . 146
6.12 Baseline comparison: dependencies between evaluated metrics 146
6.13 Consideration of topological changes in TRAVERSESEARCHTREE algorithm 148

A.1 LDBC social network benchmark schema (Source: Prat et al. [118]) . . . 158
A.2 Evaluated queries for LDBC data graph 161
A.3 Evaluated queries for DBPEDIA graph 162

B.1 Evaluation of user integration in the rewriting process for LDBC queries . 163
B.2 Evaluation of relevance 164

List of Tables

2.1 Overview of why-so methods 12
2.2 Overview of why-not methods 24
2.3 Overview of why-empty methods 30
2.4 Overview of why-so-many methods 39
2.5 Common properties of why-queries 40

3.1 Basic modification operations 48

4.1 Relevance weights produced at different iterations of relevance flooding 81
4.2 Evaluated heuristics for choosing traversal paths 83

5.1 User-preference model derived from ratings of two solutions S_1 and S_2 . 110

6.1 Graph processing operators used in operational graphs 127
6.2 Original cardinalities and cardinality thresholds of evaluated LDBC queries141

A.1 Evaluated queries and their original cardinalities for LDBC data set . . . 159
A.2 Evaluated queries and their original cardinalities for DBPEDIA data set . 159

B.1 Memory consumption and response time of why-empty rewriting system for different queries and path lengths 165

Confirmation

I confirm that I independently prepared the thesis and that I used only the references and auxiliary means indicated in the thesis.

Dresden, November 8, 2016