Elena Vasilyeva, Maik Thiele, Adrian Mocan, Wolfgang Lehner

Relaxation of Subgraph Queries Delivering Empty Results

Erstveröffentlichung in / First published in:

SSDBM 2015: International Conference on Scientific and Statistical Database Management, La Jolla 29.06. – 01.07.2015. ACM Digital Library, Art. Nr. 28. ISBN 978-1-4503-3709-0

DOI: https://doi.org/10.1145/2791347.2791382

Final edited form was published in "SSDBM 2015: International Conference on Scientific and Statistical Database Management. La Jolla 2015", Art. Nr. 28, ISBN 978-1-4503-3709-0
https://doi.org/10.1145/2791347.2791382

Relaxation of Subgraph Queries Delivering Empty Results

Elena Vasilyeva[1] Maik Thiele[2] Adrian Mocan[3] Wolfgang Lehner[2]

[1]SAP SE
Walldorf, Germany
elena.vasilyeva@sap.com

[2]Database Technology Group
Technische Universität Dresden, Germany
firstname.lastname@tu-dresden.de

[3]SAP SE
Dresden, Germany
adrian.mocan@sap.com

ABSTRACT

Graph databases with the property graph model are used in multiple domains including social networks, biology, and data integration. They provide schema-flexible storage for data of a different degree of a structure and support complex, expressive queries such as subgraph isomorphism queries. The flexibility and expressiveness of graph databases make it difficult for the users to express queries correctly and can lead to unexpected query results, e.g. empty results. Therefore, we propose a relaxation approach for subgraph isomorphism queries that is able to automatically rewrite a graph query, such that the rewritten query is similar to the original query and returns a non-empty result set. In detail, we present relaxation operations applicable to a query, cardinality estimation heuristics, and strategies for prioritizing graph query elements to be relaxed. To determine the similarity between the original query and its relaxed variants, we propose a novel cardinality-based graph edit distance. The feasibility of our approach is shown by using real-world queries from the DBpedia query log.

Keywords

Query Relaxation, Empty-Answer Problem, Graph Database, “Why Empty?”-Query

1. INTRODUCTION

The empty-answer problem is well-known in database research as a problem of queries delivering an empty result set [4, 10, 11, 13, 14]. In many scenarios a user is forced to debug a query himself to discover a predicate, which overspecifies a query and therefore is responsible for an empty result. This process implies iterative rewriting of a query and checking if it produces any answer. Although such a query usually includes only several predicates, its debugging can be a complicated task because of multiple combinations of predicates that can be relaxed. To support a user in such cases, automatic and navigational approaches are provided, where a system relaxes a query automatically till a modified query returns a non-empty result, or a user navigates the relaxation by choosing a query candidate to be proven on the delivery of a non-empty answer in each iteration.

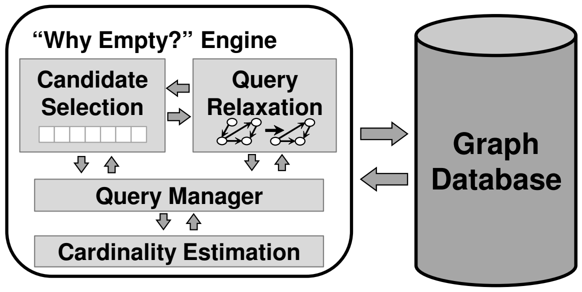


Figure 1: System architecture

The empty-answer problem in graph databases becomes even more complicated than in relational databases. Tackling a graph database, we assume that data is stored in a form of a property graph [19]: It has multiple edges, vertices, and attributes, where a vertex represents an entity, an edge shows a relationship between entities, and an attribute describes a specific property of a vertex or an edge. Queries to a graph database are modeled as property graphs and include vertices, edges, and predicates for their attributes. A fundamental graph query is a subgraph isomorphism query that describes a graph pattern to be discovered in a data graph and delivers matching subgraphs as a result.

Following the rewriting methods for relational databases, if a graph query has delivered an empty result set, it has to be rewritten in such a way that a result set of a revised query is non-empty. The full relaxation has exponential complexity for graph databases. To support a user in such scenarios, the rewriting process has to consider the specifics of graph queries and its complexity has to be reduced. In this paper, we propose an approach for explanations why a query has delivered an empty result for subgraph isomorphism queries. To the best of our knowledge, this is the first rewriting solution for an empty-result problem in graph databases.

Solution Overview

In Figure 1 we present the architecture of our system for “Why Empty?”-queries. The “Why Empty?”-engine is an extension of a graph database that is activated by a user, after a graph database delivered an empty result set. It includes the following components:

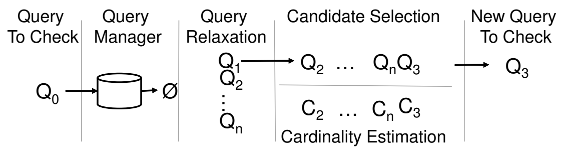


Figure 2: A single iteration of a relaxation process

- **Query Relaxation Component** rewrites a user query delivering an empty result set and generates several query candidates, which are then redirected to Candidate Selection Component.
- **Candidate Selection Component** consists of a sorted pool of query candidates as well as cardinality-based and distance-based comparators for them. This component receives relaxed candidates from the query relaxation component, and sorts them.
- **Query Manager** chooses the best query candidate from the set and requests it from the graph database. It is also responsible for querying cardinalities.
- **Cardinality Estimation Module** maintains, calculates, and estimates cardinalities for cardinality queries received by the query manager.

After a query 'failed', a user can activate the "Why Empty?"-engine responsible for relaxation of the query. The query is redirected to the query relaxation component (see a single iteration of the relaxation process in Figure 2) and is used to generate multiple query candidates. In the worst case for the first relaxation iteration, the total number of generated query candidates is $(m + n) * k$, where m is a number of edges, n is a number of vertices in a query graph, and k is a number of possible relaxation operations. We optimize this step with relaxation strategies reducing the size of the candidate space by choosing the most promising vertex and edge to be relaxed based on the estimation of cardinality change of a result set after their relaxation. The query relaxation component redirects generated candidates to the candidate selection component that ranks and stores them in the candidate pool. The ranking ensures the prioritization of more promising query candidates and therefore is a crucial point in the discovery of a query candidate delivering a non-empty answer. For this step, we propose a comparator based on distance and cardinality estimation. After the query generation the query manager selects the best candidate from the pool and sends it to the graph database. If this candidate delivers again an empty result set, the query manager forwards it to the query relaxation component for its further relaxation. The process terminates if a query delivering a non-empty answer is found. In general, our relaxation process is based on A* search, where in each step we generate a set of query candidates that are stored in a sorted list of candidates. We always select the best candidate from this list to be tested on the delivery of non-empty results.

Contributions

In this paper we make the following contributions and propose:

- Relaxation operations that are specific for property graphs.
- A cardinality-based graph edit distance that is used for comparison of any two graph queries. It is based on the cardinality estimation of the result sets produced by the compared queries.
- A relaxation process based on A* search allowing to quickly identify a more reliable query candidate first.
- Relaxation strategies that reduce the search space of query candidates.
- Heuristics for sorting and selecting query candidates according to how likely a query candidate will deliver a non-empty result set.

We introduce relaxation operations and our cardinality-based graph edit distance in Section 2. We then describe a relaxation process, candidate selection, and cardinality estimation in Sections 3 – 5. We introduce related work in Section 7 and evaluate our solution in Section 6.

2. CARDINALITY-BASED GRAPH EDIT DISTANCE

A graph query can be modified by standard graph edit operations [5] like vertex/edge insertion, vertex/edge deletion, and vertex/edge substitution, which are qualified by their implied costs. Based on these costs, we can determine a graph edit distance that measures the difference between two graphs. A graph edit distance is an aggregated cost of all graph edit operations that have to be applied to one graph in order to transform it into another one. In this section we extend the graph edit operations with operations suitable for the used data model, a property graph, and propose a cardinality-based graph edit distance based on them. As a preparation step, we introduce several definitions.

2.1 Prerequisites

As the underlying data model we use a property graph.

Definition 1 (Property Graph). *We define a property graph as a directed graph $G = (V, E, u, f, g)$ 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.*

A data graph G_d is a property graph consisting of M_d edges and N_d vertices. A query graph G_q is a property graph with M_q edges and N_q vertices.

Definition 2 (Cardinality). *We define cardinality $C(v)$ of query vertex v as a number of data vertices it matches.*

We denote the cardinality of vertex v_i as $C(v_i)$ and the cardinality of edge e_j as $C(e_j)$. The cardinality of a vertex can be annotated by predicate a_k and vertex degree deg; the cardinality of an edge can be annotated by type T, direction D, or predicate a_h. We express the cardinality for vertices without any predicate as $C(V) = N_d$ and for edges without any constraints as $C(E) = M_d$. Therefore, $C(V|a_k)$ describes the cardinality of a vertex with only single predicate a_k – the number of data vertices matching to predicate a_k. The difference in cardinality introduced by a predicate a_k is denoted as $\Delta C(V, a_k)$ and shows the maximum cardinality increase of a result set by discarding a predicate a_k.

Type	Target	Relaxation Operation
Topology	Edge	Edge Deletion
	Vertex	Vertex Deletion
	Edge	Direction Deletion
Notation	Edge, Vertex	Predicate Deletion
	Edge	Type Deletion

Table 1: Relaxation operations for property graphs

In addition, we define $src(e_j)$ a vertex where edge e_j starts and $tgt(e_j)$ a vertex where edge e_j ends.

Definition 3 (Path). *A **path** with n steps is a property graph consisting of n edges.*

A single path represents a path with only one edge and its two adjacent vertices. The cardinality of a query path represents the number of paths in the data graph matching this query path. Assume we have k paths in query G_q then the average path cardinality of G_q is the average number of data instances of these k query paths.

2.2 Graph Edit Operations

We extend the list of standard graph edit operations [5] including vertex/edge insertion, vertex/edge deletion, and vertex/edge substitution with new operations specific for property graphs like predicate deletion/insertion, type deletion/insertion, and direction deletion/insertion. Each operation has an inverse operation.

Furthermore, we denote operations that reduce the graph in terms of a number of edges, vertices, predicates, types, or directions *relaxation operations*. We call the inverse operations *augmentation operations*, because they enrich the description of a query with additional information. Both relaxation and augmentation operations describe minimal modifications that can be applied to a graph. Since we specifically focus on the empty-answer problem, we apply only relaxation operations to a query graph. We also classify the relaxation operations according to their target graph elements and types as notational and topological operations (see Table 1).

The introduced relaxation operations may have different impact on a query graph: a small query change can increase the size of a result set dramatically. To assess the extent of the changes they imply, we introduce a novel cardinality-based graph edit distance.

2.3 Purpose of Cardinality-Based Graph Edit Distance

To compare two graphs, standard graph edit distance approaches (see Section 7) can be applied that describe the difference between two graphs in terms of different numbers of edges, vertices, labels, etc. Since our system deals with graph queries delivering empty results, we strive to systematically increase the cardinality of a result set. To achieve this, two graph queries have to be compared by the estimated cardinalities of their result sets. We introduce a cardinality-based graph edit distance as a measure of a difference between two graph queries. It is expressed by the maximum expected difference in the result cardinality between these two queries introduced by transforming

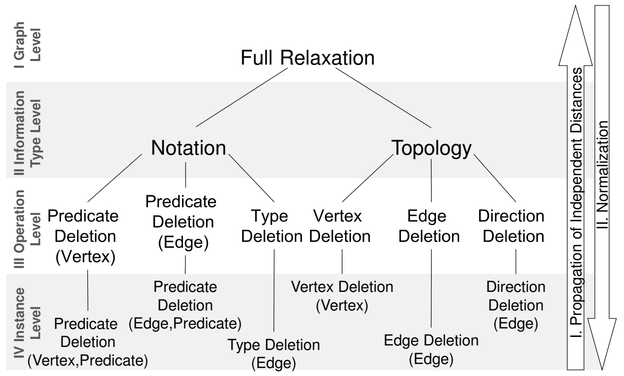


Figure 3: Cardinality-based graph edit distance

one graph query into another. We model this distance as a tree with leaves describing the distances introduced by the instances of relaxation operations and vertices representing aggregated distances. This tree exposes all upper bounds for distances for all operations and graph elements. A tree consists of notational and topological parts, and has four levels: (I) a root node – the full relaxation (the total graph edit distance), (II) aggregated notational and topological impacts, (III) types of relaxation operations, and (IV) operations' instances (see Figure 3).

By applying a relaxation operation to a specific graph element, its corresponding leaf's distance is used as the measure of cardinality change it implies. At the same time, this relaxation operation changes the distances for other leaves as well. This leads to a complexity of $O(mnk)$, where n, m are number of vertices and edges in a query graph, and k is a number of candidates. We avoid this complexity by introducing a heuristic, which models the distances to be independent and calculates them only for the single leaf element. We model the distances as the upper bound of cardinality changes caused by a relaxation operation. As a result, we get a complexity of $O(n + m)$.

The tree is constructed in two steps before the relaxation takes place: First, independent distances are calculated that show the distances of operations on independent graph elements. Second, distances are normalized by the aggregated impact. In the following we present how to derive such distances at the leaves and give some examples of applying our model to different use cases.

2.4 Calculation of Independent Distances

An independent distance shows the upper bound of the cardinality change of an element or its neighboring elements caused by a relaxation operation without considering the correlation between graph elements and their predicates. Independent distances are calculated bottom up along the tree presented in Figure 3. First, the distances at the leaves are calculated. Then they are propagated and aggregated on the parental nodes. The relaxation of notational and topological parts has different types of impact on a query graph, which we will discuss in the following.

2.4.1 Notational Distances

The predicate deletion for edges and vertices and the type deletion for edges are notational operations that can be applied to relax a query graph (see Table 1). We distinguish a type as an important attributes for an edge in a property graph and therefore we consider its deletion separately from a predicate deletion. The notational operations only modify the notational description of a graph query: no vertices or edges are removed. In order to outline the calculation of notational weights, we start from the bottom of the tree and calculate independent distances at the leaves.

Each notational operation describes the distance between relaxed and original queries caused by this operation in terms of a maximum cardinality change of a relaxed element. The predicate deletion $PD(v_i, a_k)$ of predicate a_k for vertex v_i is a cardinality change of vertex v_i caused by the relaxation of predicate a_k:

$$PD(v_i, a_k) = PD(V, a_k) = \frac{C(V) - C(V|a_k)}{C(V)} = \frac{\Delta C(V, a_k)}{N_d} \tag{1}$$

In the same way we calculate the distance for the predicate deletion $PD(e_j, a_h)$ of a predicate a_h of an edge e_j:

$$PD(e_j, a_h) = PD(E, a_h) = \frac{C(E) - C(E|a_h)}{C(E)} = \frac{\Delta C(E, a_h)}{M_d} \tag{2}$$

The similar equation is used for the type deletion $TD(e_j)$ for an edge e_j:

$$TD(e_j) = TD(E, T) = \frac{C(E) - C(E|T)}{C(E)} = \frac{\Delta C(E, T)}{M_d} \tag{3}$$

The distances at the leaves are propagated to the next level – an operation's type level, where we sum the distances of the same operation's type up, namely:

$$\begin{aligned} PD(V) &= \sum_{i=0}^{N_q} \sum_{k=0}^{K} PD(v_i, a_k), \\ PD(E) &= \sum_{j=0}^{M_q} \sum_{h=0}^{H} PD(e_j, a_h), \\ TD(E) &= \sum_{j=0}^{M_q} TD(e_j) \end{aligned} \tag{4}$$

On the information type level (level II), these aggregated distances are summed up into the aggregated distance of the notational subtree:

$$dist(notation) = PD(V) + PD(E) + TD(E) \tag{5}$$

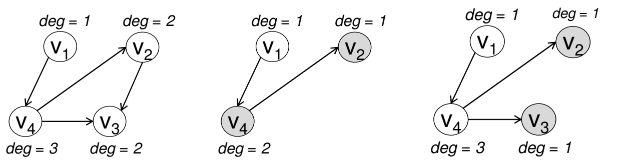
2.4.2 Topological Distances

Topological operations are represented by a vertex deletion, an edge deletion, and a direction deletion. While notational influence can be expressed by the relative cardinality of the relaxed elements themselves, topological distances for a vertex and an edge deletion are difficult to express, because if the elements are removed, their cardinalities cannot be calculated. We therefore interpret the topological distances as cardinality changes of direct neighbors of a removed element for an edge deletion and a vertex deletion. The distance for a direction deletion is expressed by the cardinality change of a relaxed edge. We start from the bottom of the tree and, first, calculate independent distances at the leaves.

A direction deletion $DD(e_j)$ of an edge e_j removes its direction. This operation increases the cardinality of a relaxed edge twice, because a relaxed edge can be considered during graph traversal in both directions.

$$DD(e_j) = DD(E) = \frac{C(E) - C(E|D)}{C(E)} = \frac{\Delta C(e_j, D)}{M_d} = 0.5 \tag{6}$$

By applying a vertex or an edge deletion, the cardinality of neighboring vertices changes in terms of their degrees. Assume we have a graph query as presented in Figure 4(a).



(a) Initial graph (b) Without v_3 (c) Without $v_2 - v_3$

Figure 4: Example of changing degree of neighbors with deletion

We have four vertices with a degree deg $\in [1, 3]$. The deletion of a vertex v_3 also removes its adjacent edges. Therefore, the degrees of direct neighbors v_2, v_4 are reduced by the number of removed adjacent edges like presented in Figure 4(b). If we delete the edge between vertices v_2, v_3, then their degrees reduce as well (see Figure 4(c)).

If we delete an edge e_j by the edge deletion operation $ED(e_j)$ the degrees of its source and target will decrease. We express the distance of an edge deletion as a cardinality change of an edge's ends derived from the change of the number of instances with a specific degree:

$$ED(e_j) = \frac{C(V|deg(src(e_j)) - 1) - C(V|deg(src(e_j)))}{C(V|deg(src(e_j)) - 1)} + \frac{C(V|deg(tgt(e_j)) - 1) - C(V|deg(tgt(e_j)))}{C(V|deg(tgt(e_j)) - 1)} \tag{7}$$

$$VD(v_i) = \sum_{p=0}^{N_{adj}} \frac{C(V|deg(v_p) - 1) - C(V|deg(v_p))}{C(V|deg(v_p) - 1)} \tag{8}$$

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

The upper levels are calculated like for the notational weight: the impact is aggregated bottom up.

2.5 Normalization of Distances

The independent distances show only cardinality changes of specific graph elements. Therefore, we need to sum up the notational and topological distances. The overall distance describes the full relaxation of the query (level I in Figure 3) and equals to 100 %:

$$dist(total) = dist(notation) + dist(topology) = 100\% \tag{9}$$

At this level we calculate the normalization factor and propagate it top down the tree (see Figure 3).

Application to Different Use Cases.

The proposed cardinality-based graph edit distance is general and not limited to any specific use case. We provide a possibility for a user to adapt the distance to a particular use case by introducing weights $\alpha, 100\% - \alpha$ for notational and topological subtrees, respectively. In our approach, we consider property graphs, for which the notational and topological descriptions are equally important and, thereby, we can assign the same weights $\alpha = 100\% - \alpha = 50\%$. Therefore, we calculate corresponding normalization coefficients like

$$\begin{aligned} unit(notation) &= \frac{dist(notation)}{\alpha} \\ unit(topology) &= \frac{dist(topology)}{100\% - \alpha} \end{aligned} \tag{10}$$

Assume we have a corporate database including information about company partners, business processes, etc. and we fill it with additional information about data flows in the company, which is extracted from internal text documents. The quality of a created graph depends strongly on the extractors and cannot be fully correct. Based on the quality of extractors, a user can define different weights for notational and topological parts. In this example, the topology is created from unreliable data and includes mistakes and, therefore, a root for the topological subtree can be annotated by a smaller distance $100\% - \alpha \ll \alpha$, which would describe how much this information is trusted. An advanced user can adapt the normalization factor even finer by specifying it on the operation level (level III).

3. QUERY RELAXATION COMPONENT

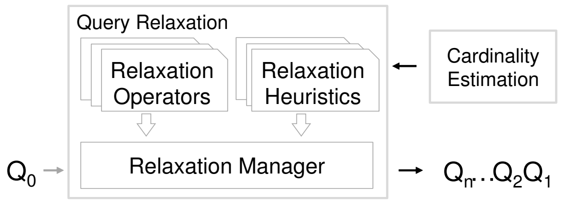


Figure 5: Query relaxation component

The main component of "Why Empty?"-engine is a query relaxation component illustrated in Figure 5. Its *Relaxation Manager* implements the relaxation process. As an input it receives a query, which has to be relaxed, and produces a set of query candidates (relaxed versions of the query) as an output. To modify a query, *Relaxation Operations* and *Relaxation Heuristics* are used. Relaxation operations have been already presented in Section 2. Relaxation heuristics select graph query elements to be relaxed based on the cardinality derived by the cardinality estimation module.

3.1 Relaxation Heuristics

In general, the number of relaxation candidates depends on the number of edges m_q, vertices n_q, and relaxation operations k and equals to $(m + n)k$. This candidate space represents the full relaxation space for a query and a relaxation iteration. With an increasing number of iterations l, the candidate space grows and can include up to $(m + n)kl$ candidates. To reduce the candidate space, we propose two heuristics for selection of edges and vertices to be relaxed.

H1: Minimum Cardinality of Edges and Vertices

This strategy selects those elements for modification that promise to have the lowest cardinality in the data set. For this, the cardinality estimation module provides estimated cardinalities for each query vertex and edge. Those elements are chosen for the relaxation, which have the lowest cardinality. At least one edge and vertex are selected by this heuristic. If several elements have the same number of data instances, then all of them are chosen.

H2: Maximum Impact & Minimum Vertex Cardinality

This heuristic is based on the calculation of minimum vertex cardinality and path cardinality. Vertices are selected according to the relaxation strategy H1. To choose an edge, we observe how cardinalities of its direct neighbors change by relaxing the edge and estimate impact of a relaxation operation for this particular edge as a 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 of a single edge). Edges with the maximum impact are chosen for the further relaxation. This heuristic accounts for the correlation between edges and their vertices in a query graph and allows to relax less representative elements first. If several vertices have the same number of data instances or several edges have the same maximum impact, then all of them are considered to be relaxed. This heuristic includes as much information as possible and allows the quick discovery of a candidate delivering a non-empty answer.

In Section 6 we will compare both heuristics with the baseline where the whole candidate space is produced. The heuristics H1 and H2 require independent cardinalities for edges, vertices, and cardinalities for paths. This information is provided by the cardinality estimation component.

4. CANDIDATE SELECTION

When a new candidate is produced, the candidate selection component has to rate it by comparing with other candidates kept in the system and to store it correspondingly. This component consists of three parts: a candidate pool and two comparators for a distance and a path. The candidate pool represents an ordered list of query candidates, where more preferable candidates are located at the beginning of the pool. To add a new candidate, a two-step comparison is done: a new candidate is compared with other candidates from head to tail of the list based on the average path cardinality and on the distance (Figure 6).

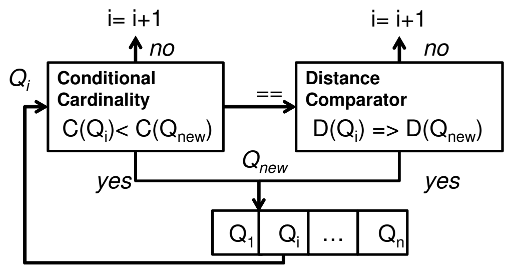


Figure 6: Candidate selection component

The comparison of path cardinalities is hierarchical. First, we choose the maximum path level available for two candidates and compare them. If average path cardinalities are equal, then we decrement the level and compare them again. The last available level is a single path. If a new candidate

Final edited form was published in "SSDBM 2015: International Conference on Scientific and Statistical Database Management. La Jolla 2015", Art. Nr. 28, ISBN 978-1-4503-3709-0
https://doi.org/10.1145/2791347.2791382

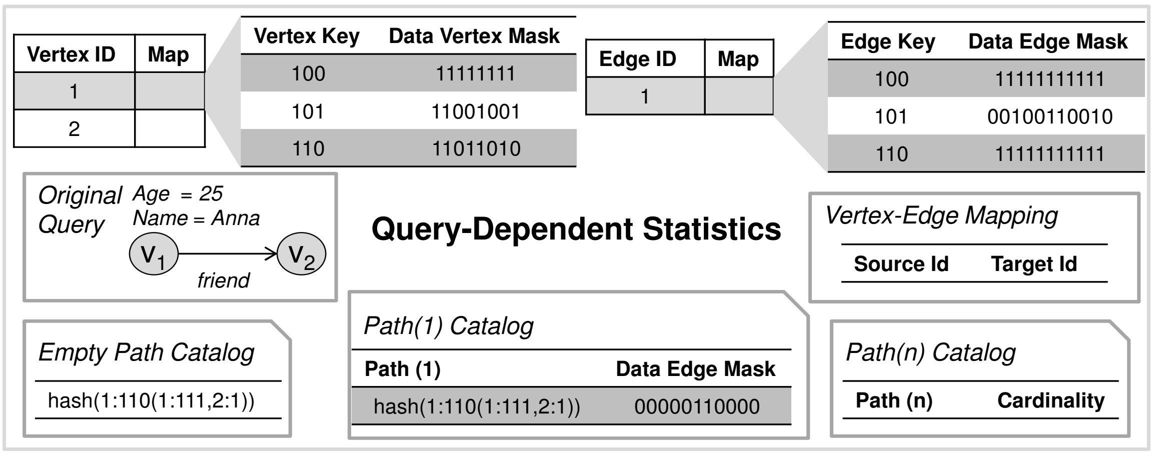


Figure 7: Query-dependent statistics

has higher average path cardinality then it is placed before a current candidate, otherwise, after it. If new and current candidates have the same average single path cardinality, then their distances are compared. The first rule – path cardinality – considers the cardinality of edges and vertices in terms of correlation. Therefore, additional cardinality-relevant parameters are not required. The second rule – distance – shows how strong the impact on cardinality changes is. In other words, the first rule chooses the most relevant changes (from small cardinality to higher cardinality), the second rule keeps the changes minimal. From these rules, the distance is a stronger comparator: it leads us to a better solution by passing all possible relaxations. Although we can get better results in term of distance, it increases the search time strongly. Based on these observations, we set it to be the second rule. If we would use it on the first place, the changes to the original query would be too small and this would increase the number of iterations and, as a consequence, the response time.

5. CARDINALITY ESTIMATION

To relax a query and to sort query candidates, relaxation strategies and comparators rely on cardinality estimation, which is done by the cardinality estimation module that receives requests for the number of data edges or data vertices matching to e_j or v_i, respectively. This module keeps general information about a query like the number of edges, vertices, predicates, and general statistics about the data graph like a number of edges, vertices, and vertex-edge mapping. To speed up the estimation of requested statistics, the module also maintains earlier requested statistics.

5.1 Query-Dependent Statistics

The query-dependent statistics is an underlying data structure that keeps cardinalities of elements of a query graph and is represented in Figure 7. It keeps an original query (shown on the left side) and supports three kinds of cardinality (see Figure 7): independent cardinalities for edges and vertices, which are stored in the cardinality maps on the top of the figure, as well as cardinalities of path expressions, which are kept in the path catalogs on the bottom of the figure.

The maps for independent vertex and edge cardinalities represented on the top of the figure have two levels. On the first level, the key is a query vertex id or query edge id. The value of this map is a map itself binding a key (mask for a specific configuration of a query element) with a data mask that has bits set for vertices or edges matching the configuration mask. The size of a key for an internal map is derived from the description of the original query. Each bit of a key represents a particular property of a vertex (edge). Any key for a vertex consists of a reservation bit and predicate bits (see Figure 8). Any key for an edge consists of at least three elements (see Figure 8): a reservation bit, a direction bit, and a type bit. In addition, it can include bits for predicates, if available.

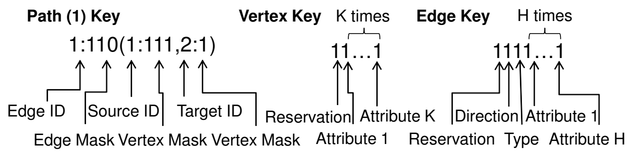


Figure 8: Components of the keys for statistics

Example. Assume the original query and its statistics presented in Figure 7. The vertex v_1 has two predicates, therefore, its key includes two bits for predicates and one reservation bit. The vertex v_2 has no predicates and its mask consists of a single reservation bit. On the top in Figure 7 we provide keys and data masks for vertex v_1 and edge e_1 in the cardinality maps.

Path statistics are collected in the catalogs responsible for storing empty paths, and paths up to size n. A key for any path catalog is a hash of a string descriptor that is a set of simple path descriptors, each of them consisting of three components: a bit mask for a query edge together with masks for its source and target. In our example on the bottom in Figure 7, path(1) for edge e_1 can be encoded as $1 : 111(1 : 111, 2 : 1)$ and stored in the path(1) catalog.

The empty path catalog on the bottom in Figure 7 keeps hashes of empty paths. The catalog for paths with length 1 maps hashes of simple paths to data edge masks. Based on this information and the corresponding vertex-edge mapping, cardinalities for n-paths can be estimated without requesting them from the graph database.

Final edited form was published in "SSDBM 2015: International Conference on Scientific and Statistical Database Management. La Jolla 2015", Art. Nr. 28, ISBN 978-1-4503-3709-0
https://doi.org/10.1145/2791347.2791382

Algorithm 1 Calculate Cardinality Vector

```
function CALCULATECARDINALITY($mask, candidates[]$)
    $resultCandidates \leftarrow \varnothing$
    $finalMask \leftarrow \varnothing$
    for all $candidate \in candidates[]$ do
        for all $bit \in mask$ do
            if ($bit == 0$) AND ($candidate[bit] <> 0$) then
                break
            if $bit$ is last bit then
                $resultCandidates \leftarrow candidate$
    for all $candidate \in resultCandidates$ do
        if $candidate$ is first then
            $finalMask \leftarrow$ mask for $candidate$
        else
            $finalMask$ &= mask for $candidate$
    return $finalMask$
```

5.2 Statistics Initialization

Before relaxation starts we fill the statistics with independent cardinalities for vertices and edges together with the description of an original query delivering an empty result set. For each predicate of a vertex and vertices without any predicates, we ask the graph database, which vertices have this predicate or the whole vertex set, respectively.

As a result set, we get a bit mask that has set bits indicating that a particular vertex matches to this particular query vertex. In such a way, we collect all independent bit masks for all vertices. Assume our previous example in Figure 7, for vertices we have to ask four queries: three for vertex v_1 and one for vertex v_2. We can reduce this number of queries by the number of vertices, because each query for a vertex without any predicates delivers the same bit mask with all bits set. We initialize the statistics with edges in the same way like with vertices. The only difference is that we include two additional queries: for the type and direction of an edge.

5.3 Statistics Maintenance

During relaxation, the statistics are requested for particular query elements. They depend on used relaxation heuristic and comparators sorting the buffer of query candidates. If statistics cannot be delivered, a miss is happened.

Any independent miss refers to a failed attempt to read an independent edge or vertex. Such a miss forces the statistics controller to derive a missing cardinality mask from available masks. This is possible because we collected all the cardinality masks for each single bit from a key in advance. To solve an independent miss, first, the statistics controller chooses all masks for the required element. Then it selects from them all matching candidates for a required key mask and derives the final mask. This is presented in Algorithm 1. First, the statistics controller selects only matching candidates based on the unset bits in the keys on lines 4 – 9. Second, the data masks for selected candidates are bitwise summed up on lines 10 – 14. Finally, the calculated data mask is returned at line 14.

A path miss happens when a path is not presented in the statistics. In this situation a corresponding data mask has to be acquired from the graph database or estimated based on the already collected information.

Empty Path Resolution.

If no bit is set in a data edge mask for a path, then we store the hash of the key of this path in the empty path catalog. Each time a mask for a path is requested the hash of its key is matched to hashes stored in the empty path catalog. If the hash is found, then this path has no representatives in a data set and as a consequence its cardinality equals to 0.

In the following, we will present how paths with multiple steps can be estimated from the existing statistics.

5.4 Estimating n-Paths

An n-path is a path consisting of n single paths. Cardinality for an n-path with $n > 1$ can be estimated from the single paths, which it consists of, with the help of the vertex-edge mapping. The vertex-edge mapping (see on the right side in Figure 7) provides source and target vertices for data edges representing with their data identifiers. For the cardinality estimation, we encode each single path from the n-path as a sparse matrix in the coordinate format. This coordinate format consists of three fields: an id of source and target vertices as well as a number of data edges between these two data vertices. First, data masks for single paths are collected for each single path in an n-path from the path(1) catalog in Figure 7. Second, all bit set in retrieved data masks are substituted by corresponding source and target vertices from the vertex-edge mapping. If a single path is specified as undirected then edges in the opposite direction have to be retrieved. For this, entries are transposed and added to directed edges. As a result, each entry in a map represents a single data edge between two vertices. Finally, we sum the number of edges between the same source and target vertices. In this way we have encoded each single path in the coordinate format. Our system supports acyclic paths with mixed directions allowing to calculate larger paths for queries. To estimate the cardinality of an n-path, we have to transpose paths for the opposite direction and then multiply produced matrices along the path.

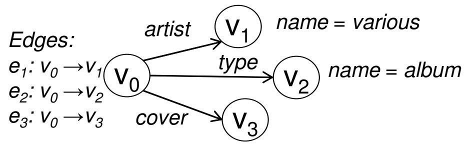

Edges:
e_1: $v_0 \rightarrow v_1$
e_2: $v_0 \rightarrow v_2$
e_3: $v_0 \rightarrow v_3$
artist
type
cover
v_0
v_1
v_2
v_3
name = various
name = album

Figure 9: Query example

Example Assume an example in Figure 9, we need to calculate the path e_1, e_2. This query describes an item of type *album*, which has a cover and is specified by an artist who has produced it. To calculate n-path, first, we prepare sparse matrices in the coordinate format. As next, we select vertices that can be used as a start or an end of a path. Such vertices should have $degree = 1$. Assume we take vertex v_1 with an incoming edge e_1. From vertex v_1 we traverse backward to vertex v_0. For this, we transpose the sparse matrix for edge e_1. Vertex v_0 in path e_1, e_2 has two outgoing edges: already processed edge e_1 and not-processed edge e_2. We take edge e_2 and traverse to its target vertex v_2. The direction of this edge corresponds to the direction of traversal. Therefore, we do not have to transpose its sparse matrix and can just simply multiply both matrices. The sum of all non-zero elements of the produced matrix is the estimated cardinality of 2-path e_1, e_2.

Final edited form was published in "SSDBM 2015: International Conference on Scientific and Statistical Database Management. La Jolla 2015", Art. Nr. 28, ISBN 978-1-4503-3709-0
https://doi.org/10.1145/2791347.2791382

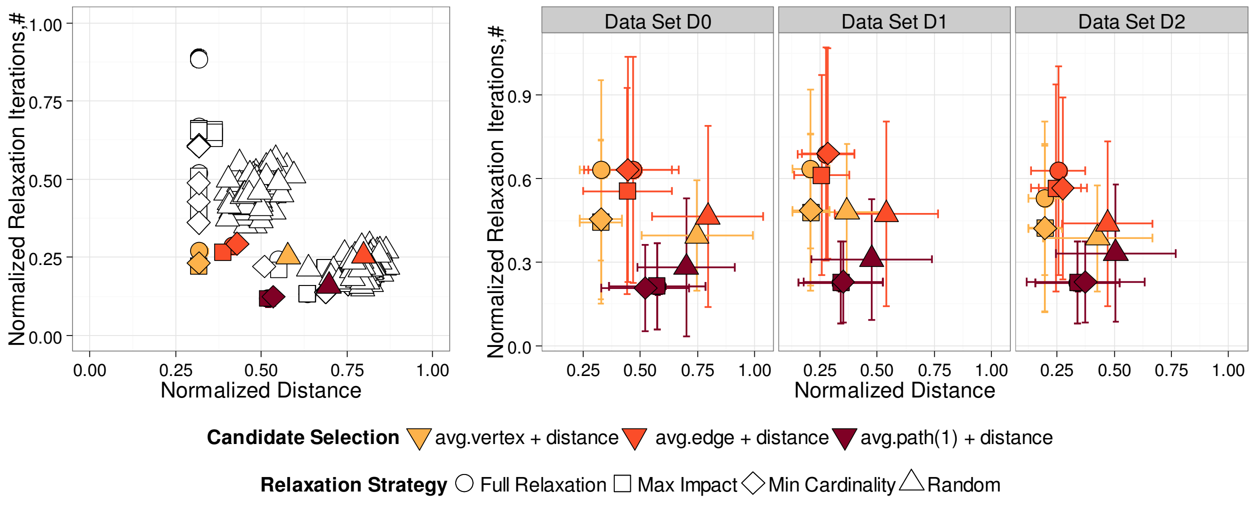


Figure 10: Comparison of strategies for candidate selection

6. EVALUATION

In this section we evaluate our framework for relaxation of subgraph isomorphism queries. We describe the evaluation setup and used data sets and evaluate different relaxation strategies and comparator configurations. We also show how the chosen candidate selection strategies converge and evaluate weight specification for topological and notational parts for cardinality-based graph edit distance.

6.1 Evaluation Setup

We implemented our "Why Empty?"-engine in C++ as an extension to a prototypical graph database [18] implementing the property graph model. It stores a graph in separate tables for vertices and edges, where vertices are represented by a set of columns for their attributes, and edges are simplified adjacency lists with attributes in a table. Both edges and vertices have unique identifiers. To enable efficient graph processing, the database provides optimized flexible tables [2, 20]. This enables schema-flexible storage of data without a predefined rigid schema. The "Why Empty?"-engine and the graph database were running on the same system. In a query, each graph element is described with predicates for attribute values.

For the evaluation we constructed three property graphs from DBpedia RDF triples, where labels represent attribute values of entities. The first data graph D_0 has $100K$ edges, $49K$ vertices, and 163 attributes. The second data graph D_1 consists of $213K$ edges, $30K$ vertices, and 740 attributes. The third data graph D_2 has $819K$ edges, $182K$ vertices, and 1542 attributes. The edges of the graph D_2 are included in the graph D_3. We have evaluated 20 real queries from the DBpedia SPARQL query log endpoint[1], which we expressed in a subgraph isomorphism language of the used graph database. As a quality measure we use the distance between an original query and a query candidate delivering a non-empty result expressed by the number of changes between them. As a performance measure we use the number of relaxation iterations required for the discovery of a query candidate with a non-empty answer. We normalize both measures by their maximal values.

[1]ftp://download.openlinksw.com/support/dbpedia/

6.2 Relaxation Heuristics

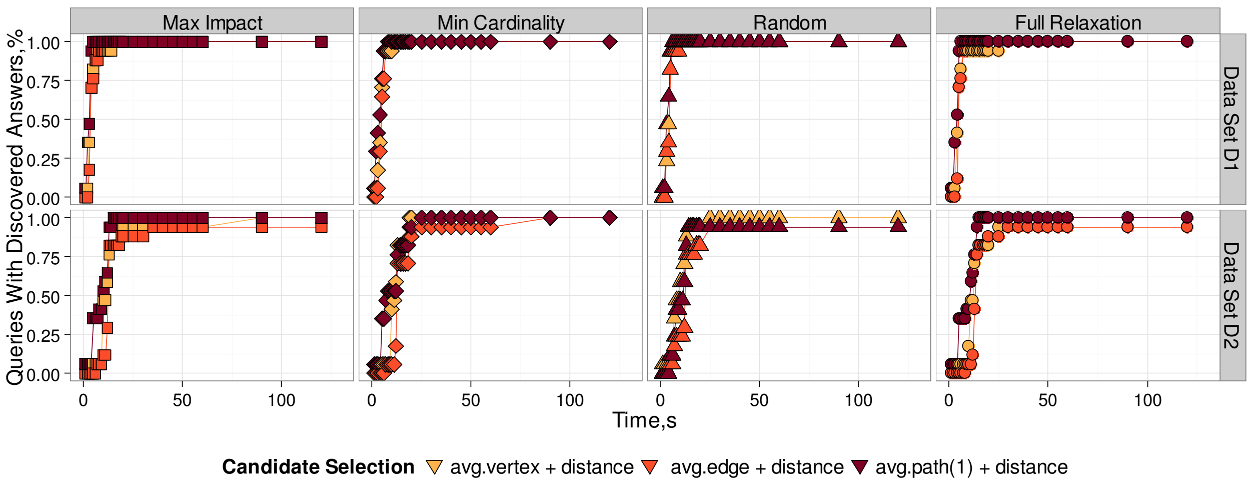
First, we evaluated different comparator configurations and the two relaxation strategies H1 (minimum cardinality), H2 (maximum impact) introduced in Section 3 with two baselines: random relaxation and full relaxation.

We have created 240 comparator configurations for sorting query candidates as follows. Each configuration has two parts: rule-based and aggregated parts (sum), each of them consists of at most three comparators based on: an average edge cardinality, an average vertex cardinality, an average and minimum single path cardinality, a number of query vertices, a number of query edges, or a distance between a candidate and an original query. Each comparator participates only in a rule-based or in an aggregated part. If no difference between two query candidates exists based on a rule-based part, then they are compared by the sum of comparator answers from the aggregated part.

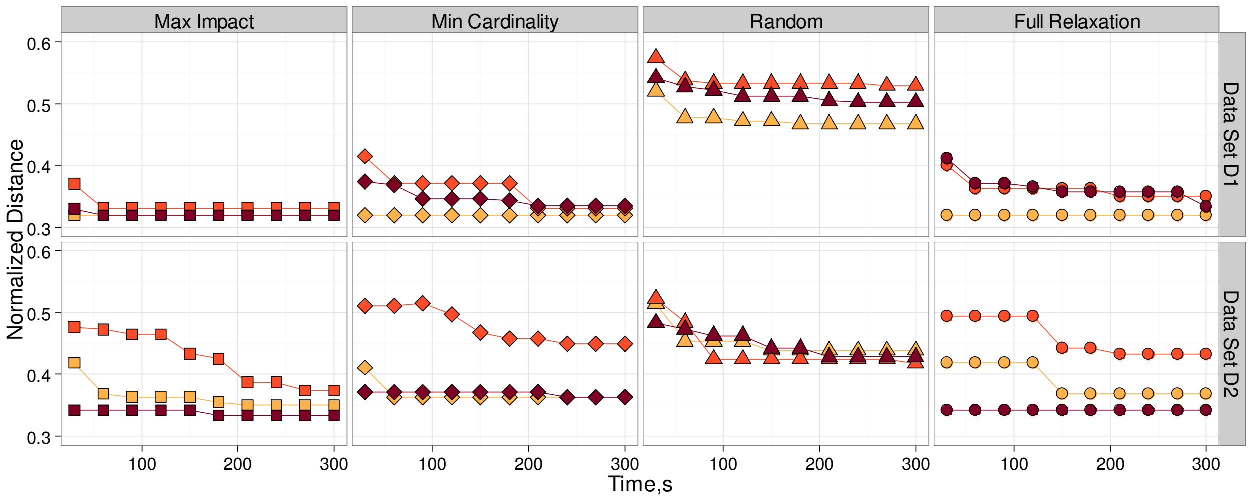
We conducted this experiment on the D_1 data set and evaluated the average numbers of relaxation iterations, which are required to discover a query candidate with a non-empty answer, normalized to the maximum number of relaxation iterations over all configurations for a single query and the average normalized distances. As we can see on the left in Figure 10, most of the comparators show high distances and number of relaxation iterations, before a query delivering a non-empty result set was found. We selected three comparators exhibiting the highest quality and the best performance: (1) avg.vertex + distance: first, average vertex cardinalities, then distances are compared; (2) avg.edge + distance: first, average edge cardinalities, then distances are compared; (3) avg.path(1) + distance: first, average single path cardinalities, then distances are compared.

We studied these three comparators in detail by applying four relaxation strategies for three data sets D_0, D_1, and D_2 (see on the right in Figure 10). For this, we normalized the number of relaxation iterations and distances to the highest values among these three data sets. The worst quality and performance are shown by the random strategy. Compara-

Final edited form was published in "SSDBM 2015: International Conference on Scientific and Statistical Database Management. La Jolla 2015", Art. Nr. 28, ISBN 978-1-4503-3709-0
https://doi.org/10.1145/2791347.2791382



(a) Discovery of the first answer



(b) Quality evaluation

Figure 11: Convergence of relaxation strategies for candidate comparators

tors $avg.vertex + distance$ and $avg.edge + distance$ based on cardinality without considering correlation between edges and vertices deliver candidates of better quality because of small changes they apply to a query, but worse performance caused by a high number of iterations. In contrast, comparator $avg.path(1) + distance$ considers as much information as possible and leads to the faster discovery of a local minimum by lacking quality in comparison to the first two comparators. In addition, $avg.path(1) + distance$ exhibits less variance in its results. Figure 10 also demonstrates that the impact of a relaxation strategy is weaker than the influence of a comparator. Our proposed comparator $avg.path(1)+distance$ with maximum impact relaxation strategy shows the best combination of distance and relaxation iterations for all data sets. We further evaluate our comparators for paths $1, 2, 3$ in Figure 12. We normalized the results to the maximum values of relaxation iterations and distances among paths. As we can see, the number of relaxations remains the same, but the distance can increase

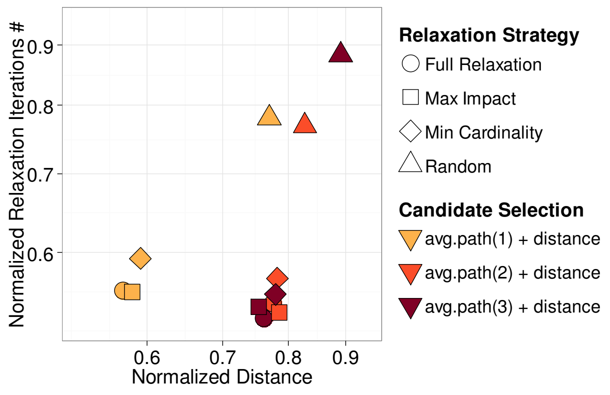


Figure 12: Path evaluation

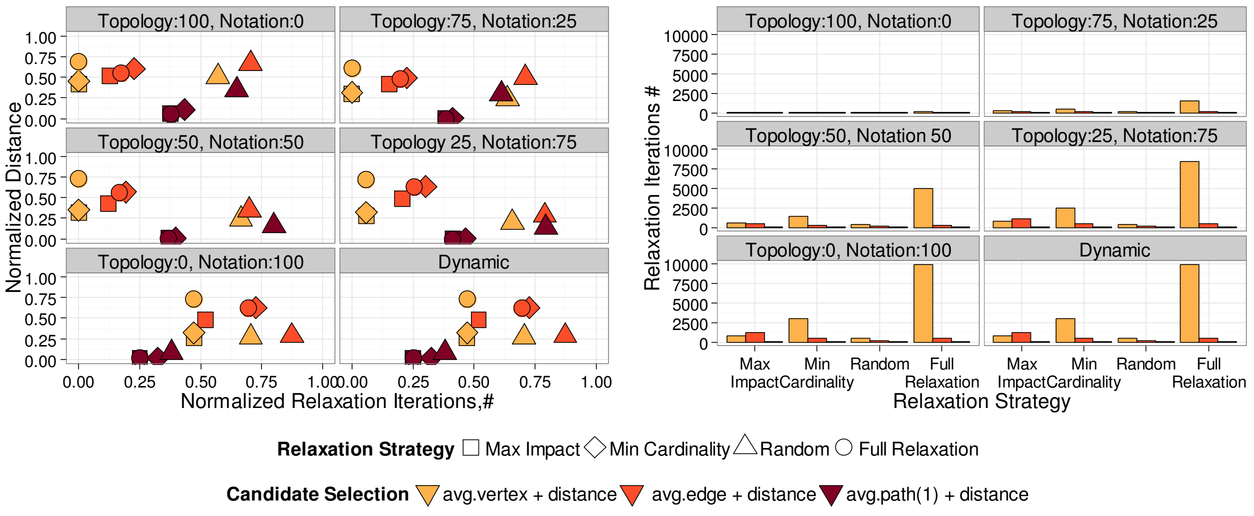


Figure 13: Weights selection for topological and notational parts

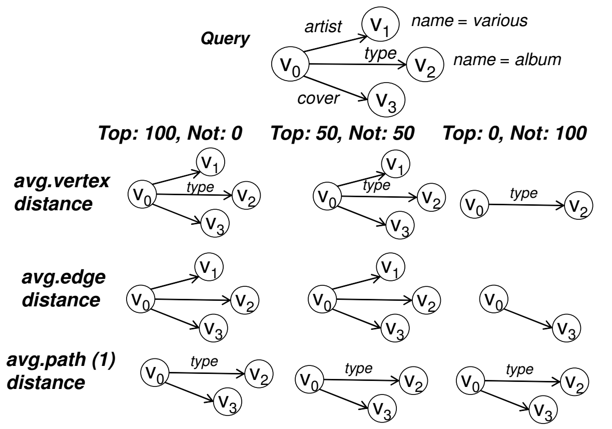


Figure 14: Query candidates delivering non-empty answers

a little. This happens because longer paths bring some noise into the results. Based on this observation, we can conclude that the use of single paths is enough, and in the following experiments we use only single paths.

6.3 Convergence

Second, we evaluate convergence of relaxation strategies and comparators in terms of the discovery of the first candidate delivering a non-empty result and time for discovering candidates of better quality. For this purpose, we take a time budget of five minutes and plot the ratio of 20 evaluated queries with discovered the first query candidate delivering a non-empty result set in Figure 11(a) and the best discovered solution (in terms of the average normalized distance) at the end of every 30 seconds in Figure 11(b). We conducted this set of experiments on two data sets D_1 and D_2. The convergence to the first answer occurs faster for our path-based solution (see in Figure 11(a)). The convergence time increases with the increasing size of a data set. As we can see in Figure 11(b), the maximum impact relaxation strategy shows the best results for all three strategies (comparator configurations $avg.path(1)+distance$ and $avg.edge+distance$ show the same results and are overlapped for D_1). The maximum impact relaxation strategy with the path-based comparator outperforms both baselines: random and full relaxations, and also converges faster than minimum cardinality relaxation strategy. With an increasing size of the data graph (see bottom charts $D2$ in Figure 11), the impact of comparators becomes stronger and requires a path-based solution for fast convergence. Random relaxation exhibits the worst results as the distance of produced query candidates is very high and the relaxation process does not converge to the best solution during these five minutes. The evaluation shows that it is possible to discover results of higher quality if we continue the search after the discovery of the first candidate with a non-empty answer. Still these solutions do not out-perform the maximum impact strategy.

6.4 Graph Edit Distance

In this section we evaluate the weight distribution for the notational and topological parts. For this, we change them between $[0, 100]$ and additionally compare them with dynamic weights (this is a general use case: no separation of weights for notation and topology, weights are propagated to the root and then normalized). We normalize the measurements between the minimum and maximum number of relaxation iterations and distances for each query and then calculate average. The results are presented in Figure 13.

We can see that by using our cardinality-based distance we can get the fastest result of the best quality (see "Dynamic" case). Regarding the weight propagation, our comparator $avg.path(1) + distance$ provides the fastest results. The topology of evaluated queries already exists in our data graph, but suitable notation is missing. This can be seen from the best distance for the strategies based on independent cardinality for cases where the notational part has a low weight. For these cases, queries with relaxed notational parts are preferred.

In Figure 14 we present a query delivering an empty result set and its solutions for maximum impact relaxation strategy. This evaluation supports our conclusions: our strategy is stable for different weight configurations. The candidate comparator $avg.edge + distance$ based on edge cardinalities relaxes the whole query description in order to improve the independent cardinality.

7. RELATED WORK

In this section we present state of the art research in explaining unexpected answers and graph edit distance.

7.1 Solutions for Empty Answer Problem

The empty-answer problem is being extensively studied for conjunctive queries in traditional relational settings. A typical example of such queries is a web form, where a user enters predicates for attributes of the search, which are interesting for him in order to discover some information, for example: flight information like departure time, duration of travel etc. There are two common ways to support a user in case a query delivered an empty result: automatic and navigational solutions. In the first group [4, 10, 11], a system generates several query candidates by relaxing some of the attributes. This approach suffers from a large number of candidates that complicates the selection of the best candidate. Chaudhuri [4] defines extended queries that provide additional constraints on the result set and examines some query constrains as flexible constraints. Jannach et al. [10] calculate user-optimal query relaxations by evaluating subqueries and detecting conflicts for fast relaxation of preferred conflicts on demand. The second group [13, 14] overcomes the problem of too many query candidates by involving a user into the relaxation process. Instead of recommending a lot of proposals, a system provides only a small portion and a user chooses his favorite if required. In such systems a user navigates the search, this is the reason why they are called navigational. 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.

These approaches solve the empty-answer problem for relational databases and conjunctive queries. In our use case we work with subgraph isomorphism queries delivering empty answers. We need to consider specifics of a graph query and predicates on edges and vertices. These proposed solutions represent only a partial problem for predicates. In our previous work [22, 23], we mainly focused on determining which parts of a graph query are represented in a data set without considering partial representations for edges and vertices. Therefore, in this current work we focus on rewriting and consider specifics of graph queries like a direction, predicates, edges' types, edges, and vertices.

7.2 Solutions for Missing Answers

The problem of empty answers can also be modeled as a problem of missing answers. In this case "Why Not?"-queries are used that explain why items of interest are missing from a result set. "Why Not?"-queries can be classified into two groups according to explanations they provide: provenance-based and query rewriting. The first group, provenance-based methods, generates one of the following explanations: query-based [1, 3], instance-based [7, 8, 9], or hybrid explanations [6]. To provide query-based explanations [3], a query tree is traversed and operations, which reject the tuples of interest are delivered to a user as explanations. Instance-based explanations [7, 8, 9] show how a data source has to be modified in order to deliver missing answers. This kind of explanations can be used in data integration tasks, where the extraction process can fail and deliver not fully correct data. Like in the previous section, these approaches are relation-oriented and do not consider specifics of a graph.

7.3 Flexible Query Answering

To prevent an empty-answer problem, flexible query answering can be used. Approximate queries [12] can include approximation for topology and vocabularies. This approach calculates notational relatedness between vertices to identify if the connection is meaningful and deals with incorrect annotations for edges and vertices. In comparison to this approach, we assume that vertices and edges are described grammatically correctly, but the construction of a query and assignment of predicates is wrong. The approach of uncertain predicates [26] is similar to our assumptions, where a user has to point out which predicates can be uncertain and relaxed. For this, a user has to annotated a set of predicates with probability (relevance for a user). Another approach for flexible query answering is realized by keyword-search. Originally, 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. [21] compute query proposals from the keywords and provide them to a user, who chooses a candidate to be processed by a query processor. These methods help a user to create a meaningful query and prevent him from creating a query delivering empty answers. In contrast, we give help to a user when he already knows what he wants exactly to get from a graph database and to debug the query by providing query candidates delivering non-empty results.

7.4 Graph Edit Distance

Graph edit distance [5] measures similarity between two graphs by transforming one graph into another with a sequence of edit operations (vertex deletion, edge deletion, and vertex substitution). Every operation is characterized by its cost. The least-cost edit operation sequence is used as a graph edit distance. Algorithms for graph edit distance are classified into two groups according to a graph type they consider: non-attributed graphs and attributed graphs [5]. Graph edit distance for a non-attributed graph describes topological difference. In this case, compared graphs are coded as strings and a string distance is used as a graph edit distance, for example: Hamming [25], Levenstein [15], and Markov [24] edit distances. The dissimilarity of two attributed graphs is calculated according to their attributes. If a vertex has k attributes and an edge has h attributes, then k-dimensional and h-dimensional attribute space is constructed, which is used to calculate the distances between attributes of graphs. This can be done with self-organizing maps [17], probability approach estimating the frequency of edit operations and hereby constructing the distribution of edit operations [16]. For further reading we recommend the survey on a graph edit distance [5] to interested readers. In comparison to these solutions, our approach calculates the difference between a query candidate and an original query based on the number of instances of their edges and vertices in a data graph. We do not compare the structure and label similarity for these two graphs, because a query candidate is already a subgraph of an original query.

8. CONCLUSION

The empty-answer problem known from the database research on conjunctive queries becomes a difficult problem in graph databases explained by the complexity of graph data model and supported graph queries. In this paper we propose a solution for this problem for subgraph isomorphism queries based on "Why Empty?"-queries that are triggered by a user in case of getting an empty result set. These queries are processed by the "Why Empty?"-engine that rewrites an original query in such a way that a new query delivers a non-empty result set. The full relaxation process has exponential complexity because of the variety of combinations of elements to be relaxed and relaxation sequences. We reduce the complexity heuristically by providing comparators considering path cardinalities and choosing only a subset of elements to be relaxed based on the impact of their relaxation on the neighbors in a graph query. We also introduce relaxation operations for property graphs and our cardinality-based graph edit distance that expresses the difference between two query candidates in terms of maximum estimated cardinality difference between them. We evaluate our system on three data sets with real queries from the DBpedia log. Our approach based on the cardinality-based graph edit distance with the maximum impact relaxation strategy shows the best combination of quality and performance and scales with an increasing size of a data graph.

Acknowledgment

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

9. REFERENCES

[1] N. Bidoit, M. Herschel, and K. Tzompanaki. Query-based Why-Not provenance with NedExplain. In *Proc. EDBT*, pages 145–156, 2014.

[2] C. Bornhövd, R. Kubis, W. Lehner, H. Voigt, and H. Werner. Flexible information management, exploration and analysis in SAP HANA. In *DATA*, pages 15–28, 2012.

[3] A. Chapman and H. V. Jagadish. Why Not? In *Proc. SIGMOD*, pages 523–534. ACM, 2009.

[4] S. Chaudhuri. Generalization and a framework for query modification. In *Proc. ICDE*, pages 138–145. IEEE Computer Society, 1990.

[5] X. Gao, B. Xiao, D. Tao, and X. Li. A survey of graph edit distance. *Pattern Anal. Appl.*, pages 113–129, 2010.

[6] M. Herschel. Wondering why data are missing from query results?: Ask conseil why-not. In *Proc. CIKM*, pages 2213–2218. ACM, 2013.

[7] M. Herschel and M. A. Hernández. Explaining missing answers to SPJUA queries. *Proc. VLDB Endow.*, pages 185–196, 2010.

[8] M. Herschel, M. A. Hernández, and W.-C. Tan. Artemis: A system for analyzing missing answers. *Proc. VLDB Endow.*, pages 1550–1553, 2009.

[9] J. Huang, T. Chen, A. Doan, and J. F. Naughton. On the provenance of non-answers to queries over extracted data. *Proc. VLDB Endow.*, pages 736–747, 2008.

[10] D. Jannach. Techniques for fast query relaxation in content-based recommender systems. In *Proc. KI*, pages 49–63. Springer-Verlag, 2007.

[11] U. Junker. QUICKXPLAIN: preferred explanations and relaxations for over-constrained problems. In *AAAI*, pages 167–172, 2004.

[12] F. Mandreoli, R. Martoglia, G. Villani, and W. Penzo. Flexible query answering on graph-modeled data. In *Proc. EDBT*, pages 216–227. ACM, 2009.

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

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

[15] R. Myers, R. Wison, and E. R. Hancock. Bayesian graph edit distance. *IEEE Trans. Pattern Anal. Mach. Intell.*, pages 628–635, 2000.

[16] M. Neuhaus and H. Bunke. A probabilistic approach to learning costs for graph edit distance. In *Proc. ICPR*, pages 389–393. IEEE, 2004.

[17] M. Neuhaus and H. Bunke. Self-organizing maps for learning the edit costs in graph matching. *Systems, Man, and Cybernetics, Part B: Cybernetics, IEEE Transactions on*, pages 503–514, 2005.

[18] M. Paradies, M. Rudolf, C. Bornhövd, and W. Lehner. GRATIN: Accelerating graph traversals in main-memory column stores. In *GRADES*, pages 9:1–9:6. ACM, 2014.

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

[20] M. Rudolf, M. Paradies, C. Bornhövd, and W. Lehner. The graph story of the SAP HANA database. In *Datenbanksysteme für Business, Technologie und Web (BTW). Proceedings*, pages 403–420, 2013.

[21] T. Tran, H. Wang, S. Rudolph, and P. Cimiano. Top-k exploration of query candidates for efficient keyword search on graph-shaped (RDF) data. In *Proc. ICDE*, pages 405–416, 2009.

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

[23] E. Vasilyeva, M. Thiele, C. Bornhovd, and W. Lehner. Top-k differential queries in graph databases. In *Advances in Databases and Information Systems*, Lecture Notes in Computer Science, pages 112–125. Springer International Publishing, 2014.

[24] J. Wei. Markov edit distance. *IEEE Trans. Pattern Anal. Mach. Intell.*, pages 311–321, 2003.

[25] R. C. Wilson and E. R. Hancock. Structural matching by discrete relaxation. *IEEE Trans. Pattern Anal. Mach. Intell.*, pages 634–648, 1997.

[26] H. Zhou, A. A. Shaverdian, H. Jagadish, and G. Michailidis. Querying graphs with uncertain predicates. In *Proc. Eighth Workshop on Mining and Learning with Graphs*, pages 163–170. ACM, 2010.