Elena Vasilyeva, Maik Thiele, Christof Bornhövd, Wolfgang Lehner

Top-k Differential Queries in Graph Databases

Erstveröffentlichung in / First published in:

Advances in Databases and Information Systems: 18th East European Conference. Ohrid, 07.-10.09.2014. Springer, S. 112–125. ISBN 978-3-319-10933-6.

DOI: http://dx.doi.org/10.1007/978-3-319-10933-6_9

Final edited form was published in "Advances in Databases and Information Systems: 18th East European Conference. Ohrid 2014", S. 112–125, ISBN 978-3-319-10933-6
http://dx.doi.org/10.1007/978-3-319-10933-6_9

Top-k Differential Queries in Graph Databases

Elena Vasilyeva[1], Maik Thiele[2], Christof Bornhövd[3], and Wolfgang Lehner[2]

[1] SAP AG, Chemnitzer Str. 48, 01187 Dresden, Germany
elena.vasilyeva@sap.com
[2] Technische Universität Dresden, Database Technology Group
Nöthnitzer Str. 46, 01187 Dresden, Germany
{maik.thiele,wolfgang.lehner}@tu-dresden.de
[3] SAP Labs, LLC, Palo Alto, USA
christof.bornhoevd@sap.com

Abstract. The sheer volume as well as the schema complexity of today's graph databases impede the users in formulating queries against these databases and often cause queries to "fail" by delivering empty answers. To support users in such situations, the concept of differential queries can be used to bridge the gap between an unexpected result (e.g. an empty result set) and the query intention of users. These queries deliver missing parts of a query graph and, therefore, work with such scenarios that require users to specify a query graph. Based on the discovered information about a missing query subgraph, users may understand which vertices and edges are the reasons for queries that unexpectedly return empty answers, and thus can reformulate the queries if needed. A study showed that the result sets of differential queries are often too large to be manually introspected by users and thus a reduction of the number of results and their ranking is required. To address these issues, we extend the concept of differential queries and introduce top-k differential queries that calculate the ranking based on users' preferences and therefore significantly support the users' understanding of query database management systems. The idea consists of assigning relevance weights to vertices or edges of a query graph by users that steer the graph search and are used in the scoring function for top-k differential results. Along with the novel concept of the top-k differential queries, we further propose a strategy for propagating relevance weights and we model the search along the most relevant paths.

Keywords: Graph databases, Top-k Differential Queries, Flooding.

1 Introduction

Following the principle "data comes first, schema comes second", graph databases allow to store data without having a predefined, rigid schema and enable a gradual evolution of data together with its schema. Unfortunately, schema flexibility impedes the formulation of queries. Due to the agile flavor of integration and interpretation processes, users very often do not possess deep knowledge of the data and its evolving schema. As a consequence, issued queries might return unexpected result sets, especially empty results. To support users to understand the reasons of an empty answer, we already proposed the notion of a differential query [15]; a graph query (for example see Figure 1(a)) that has a result consisting of two parts: (1) a discovered subgraph that is a part of a data graph isomorphic to a query subgraph like in Figure 1(b), and (2) a difference graph reflecting the remaining part of a query like in Figure 1(c). Differential queries work in scenarios, where users need to specify a query in the form of a graph, such as subgraph matching queries. Although the approach in [15] already supports users in the query answering process, it still has some limitations: the number of intermediate results can be very large, e.g. it can reach up to 150K subgraphs for a data graph consisting of 100K edges and a query graph with 10 edges. Optimization strategies reducing the number of traversals for a query based on cardinality and degree of a query's vertices could prune intermediate results, but, as a side effect, they also could remove important subgraphs, since these strategies do not consider the users' intention.

Contributions

To cope with this issue, we extend the concept of differential queries with a top-k semantic, resulting in so-called top-k differential queries that are the main contribution of this paper. These queries allow the user to mark vertices, edges, or entire subgraphs of a query graph with relevance weights showing how important specified graph elements are within a query. To make the search of a top-k differential query with multiple relevance weights possible, we present an algorithm for the propagation of relevance weights: relevance flooding. Based on the propagated weights, the system decides automatically how to conduct the search in order to deliver only the most relevant subgraphs to a user as an alternative result set of the original query. The initial weights are used to rank the results. The concept of top-k differential queries allows us to reduce processing efforts on the one side and allows to rank individual answers according to the user's interest on the other side.

The rest of the paper is structured as follows. In Section 2 we present the state of the art related work. The property graph model and differential queries are introduced in Section 3. Section 4 describes the relevance-based search and its application to top-k differential queries. We evaluate our approach in Section 5.

2 Related Work

In this section we present solutions for "Why Not?" queries and for the empty-answer problem, ranking of query results, and flexible query answering.

"Why Not?" Queries and Empty-Answer Problem

The problem of unexpected answers is generally addressed by "Why Not?" queries [3] determining why items of interest are not in the result set. It is assumed that the size and complexity of data prevent a user from manually studying the reasons in a feasible way. A user specifies the items of interest with attributes or key values and conducts a "Why Not?" query. The answers to such a query can be (1) an operator of a query tree, removing the item from processing [3], (2) a data source in provenance-based "Why Not?" queries like for example in [6], or (3) a refined query that contains the items of interest in the result set like in [14]. In contrast to approaches tailored for relational databases, we do not operate on a query tree constructed for a query execution plan, but we deal with a query graph, for which we search corresponding data subgraphs by a breadth-first or depth-first traversal considering user-defined restrictions with respect to vertices and edges based on their attribute values. It is important to understand, which query edges and vertices are responsible for the delivery of an empty result set.

Query rewriting for the empty-answer problem can also be enhanced by user interaction [10]. This interactive query relaxation framework for conjunctive queries [10] constructs a query relaxation tree from all possible combinations of attributes' relaxations. Following the tree top-down, a user receives proposals for query relaxations and selects preferred ones. This approach [10] has only a single objective function. In our settings, it would be only a single vertex of interest. To model multiple relevant elements and to detect the optimal path between them cannot be achieved by this approach proposed in [10].

Ranking of Query Results

The concept of top-k queries derives from relational database management systems, where the results are calculated and sorted according to a scoring function. In graph databases top-k queries are used for ranking (sub)graph matchers [16,17]. These ranking strategies differ in regard to how a data graph is stored in a graph database. If a database maintains multiple data graphs, for example chemical structures, then a similarity measure based on a maximum common subgraph between a query and an individual data graph can be used as a scoring function [16]. If a database maintains a single large data graph, then the approach of top-k subgraph matchers [17] can be applied. In this context, it is assumed that a data graph has naturally a hierarchical structure that can be used for index construction and clustering of data subgraphs enabling effective pruning. These solutions do not consider any relevance function for a query graph which is paramount in our setup.

To rank the results, an "interesting" function [5], relevance and distance functions [4], or estimation of confidence, informativeness, and compactness [7] can be used. In the first case [5], such an "interesting" function is defined in advance by a use case, for example, it can be a data transfer rate between computers in a network. Up front, we do not have any "interesting" function in a data graph. In the second case [4], the matching problem is revised by the concept of "output node", which presents the main part of a query answer to be delivered to a user. In our settings, this approach could be compared to a single vertex with a user-specified relevance weight. In the third case [7], additional semantic information is used to estimate scoring functions. In contrast, we assume that the data graph has the maximal confidence, our user is interested in subgraph matching queries without accounting for additional semantic information. The compactness of answers is not considered in our work, because we deal with exact matching, and the answers containing more relevant parts matching to the initial query are ranked higher. Our approach can be further improved by estimating the informativeness, which should be based on a user's preferences. This question is left for future work.

In [1] the top-k processing for XML repositories is presented. The authors relax the original query, calculate the score of a new query based on its content- and structure-based modifications, and search for the matches. While Amer-Yahia et al. relax the query and search for a matching document, we process a data graph without any changes to the original query. Instead we do search for exact subgraph matches. Subgraphs can also be matched and ranked by approximate matching and simulation-based algorithms, which can result in inaccurate answers with a wrong graph shape or non matching vertices. Since we provide exact matches, the class of inexact algorithms is not considered in our work.

Flexible Query Answering

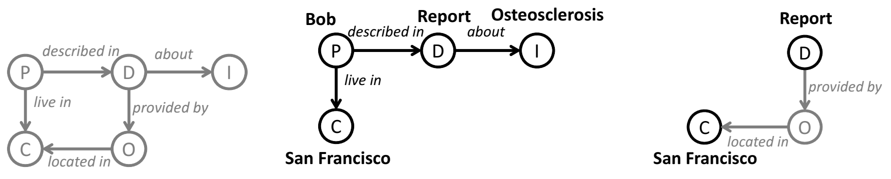
A different approach tackling the problem of overspecified queries can be modeled by the SPARQL language [11]. SPARQL provides the `OPTIONAL` clause, which allows to process a query graph if a statement or an entire subgraph is missing in a data graph. The `UNION` clause allows to specify alternative patterns. Defining a flexible query is not straight-forward: a user has to produce all possible combinations of missing edges and vertices in a query graph to derive results, this requires good knowledge of SPARQL. Moreover, this language does not support relevance weights on a query graph directly, and a user cannot have a direct impact on the search within the database. Furthermore, it does not support the calculation of difference graphs.

3 Preliminaries

In this section we present a general overview on the used graph model and differential queries in graph databases.

Property Graph

A graph database stores data as vertices and edges. Any query to a graph database and corresponding results may be understood as graphs themselves. As an underlying graph model we use the property graph model [12], a very general model, describing a graph as a directed multigraph. It models entities via vertices and relationships between them via edges. Each graph element can be characterized by attributes and their values, allowing the combination of data with different structures and semantics. The mathematical definition and the comparison of this model with other graph models are provided in [12,15].



(a) Differential query (b) Discovered subgraph (c) Difference graph

Fig. 1. Differential query and its results

Differential Queries

If a user receives an empty result set from a graph database, a differential query can be launched that investigates the reasons of an empty result [15]. The differential query is the initial graph query delivering an empty answer, marked by a specified keyword. In order to provide some insights into the "failure" of a query, a user receives intermediate results of the query processing consisting of two parts: a data subgraph and a missing part of the original query graph. The first part consists of a maximum common subgraph between a data graph and the query graph that was discovered by any maximum common subgraph algorithm suitable for property graphs. This can be for example the McGregor maximum common subgraph algorithm [8]. The second part reflects a difference graph - a "difference" between a query graph and a discovered maximum common subgraph. It shows the part of a differential query that is missing from a data graph and therefore displays the reason why the original query "failed". The difference graph is also annotated with additional constraints at the vertices, which are adjacent to the discovered subgraph as connecting points.

As an example, imagine a data graph derived from text documents that contains information about patients, their diagnoses, and medical institutions. We store the data graph 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 patients (P), their diseases (I), their cities of residence (C), medical institutions (O), and information documents (D) like in Figure 1(a). If the query does not deliver any answer, the doctor launches the query as a differential query and receives the following results:

– The discovered subgraph in Figure 1(b): A person, called Bob, living in San Francisco, whose information was described in "Report", which is about osteosclerosis.
– The difference graph in Figure 1(c): There is no information about any medical institution located in San Francisco, which provided the "Report".

Differential Query Processing

The processing of a differential query is based on the discovery of maximum common subgraphs between a query and a data graph as well as on the computation of difference graphs. Firstly, the system selects a starting vertex and edge from a query graph. Secondly, it searches a corresponding data subgraph in a breadth-first or depth-first manner. If a maximum possible data subgraph for a chosen starting vertex is found, then the system stores this intermediate result, chooses a next starting vertex, and searches again. This process is repeated with every vertex as a starting point. If the search is done only from a single starting vertex, then the largest maximum common subgraph might be missing, because not all edges exist in a data graph. In a final step, the system selects the maximum common subgraphs from all intermediate results, computes the corresponding difference graphs, and returns them to a user.

Due to the nature of the differential queries, redundant intermediate subgraphs and their multiple processing create a potentially significant processing overhead. The number of intermediate results can reach up to 150K subgraphs (Figure 5(d)) for a data graph of 100K edges. In order to cope with this issue, we already proposed different strategies for the selection of a starting vertex [15]: based on cardinality or degree of vertices. Although the number of answers is reduced, it can still remain large to be processed manually. As a side effect, some subgraphs, which are potentially relevant for a user, might be excluded from a search, because the strategies do not take a user's intention into account. To avoid this, we propose an extended concept of differential queries – top-k differential queries, which process a query graph and rank results according to user-defined relevance weights.

4 Top-k Differential Query Processing

In this section we describe the core of our approach – the relevance-based search with relevance flooding and the detection of an optimal traversal path through a differential query, and ranking of results.

4.1 Top-k Differential Queries

We define a **top-k differential query** as a directed graph $G_q^k = (V, E, u, f, g, k)$ over attribute space $A = A_V \dot{\cup} A_E$, where: (1) V, E are finite sets of N vertices and M edges, respectively; (2) $u : E \to V^2$ is a mapping between edges and vertices; (3) $f(V)$ and $g(E)$ are attribute functions for vertices and edges; (4) A_V and A_E are their attribute space, and (5) k is a number of required results.

The goal of a top-k differential query is to search subgraphs based on relevance weights and to rank the discovered subgraphs according to a relevance-based scoring function. For this, we introduce so-called *relevance weights* for vertices $\omega(v_i)$ and edges $\omega(e_j)$ in a query graph, which annotate graph elements, vertices and/or edges, in a query graph with float numbers $\in [0; 1]$. A weight $\omega = 0$ denotes low relevance and thus reflects the default of a vertex and an edge. In our work we do not concern negative evidence, because if a graph element is not interesting to a user, then it would not be included in the query. Graph elements with higher relevance weights in a query are more important to a user than those with lower values. The introduction of relevance weights does

Final edited form was published in "Advances in Databases and Information Systems: 18th East European Conference. Ohrid 2014", S. 112–125, ISBN 978-3-319-10933-6
http://dx.doi.org/10.1007/978-3-319-10933-6_9

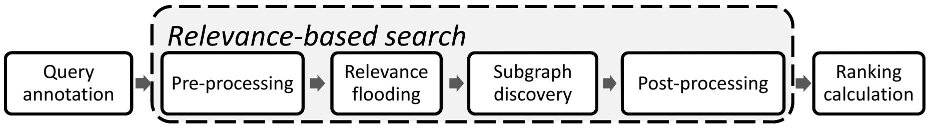


Fig. 2. Top-k differential query processing

not affect the definition of top-k differential queries, this is just an additional property for edges and vertices: $\omega(V) \subseteq f(V)$ and $\omega(E) \subseteq g(E)$.

The relevance weights are used for several purposes, e.g. (1) for steering our search in a more relevant direction, (2) for earlier processing of elements with higher relevance, and most importantly (3) in a scoring function for the ranking itself. The values facilitate the discovery of such subgraphs that are more interesting to a user, and the elimination of less relevant subgraphs.

The processing of top-k differential queries is performed as depicted in Figure 2. After a user has annotated a query with the relevance weights, a relevance-based search is started. When no new data subgraphs can be found, the system stops the search, calculates the rank of discovered subgraphs, and returns results to a user. In the following, we describe all these processing steps in more detail.

4.2 User and Application Origin of Relevance Weights

Relevance weights described in the previous paragraph can be determined based on a user's preferences or based on a particular use case. If relevance weights are assigned by a user, then the more important graph elements get higher weights. With reference to our running example (Figure 1(a)), if a doctor is more interested in the names of patients and their diseases, then he provides the highest relevance to corresponding vertices: $\omega(v_P) = 1$ and $\omega(v_I) = 1$.

If relevance weights are determined by a particular use case, then they are defined considering specific features of the use case – an *objective function*, which the use case tries to minimize or maximize. Some examples of objective functions would be the data transfer rate in networks of hubs or traffic in the road networks. If a user aims to maximize the objective function, then graph elements with higher values of the objective function are annotated by higher relevance weights. In our approach, we do not assume any specific use case and expect that relevance weights are defined by a user.

4.3 Relevance-Based Search

After a user has annotated the query graph, the relevance-based search is conducted, which is outlined in the dashed box in Figure 2. 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 query graph, if at least one vertex does not have a user-defined relevance weight. Then, relevant subgraphs are searched in a data graph. After the search the post-processing is executed over relevance weights to prepare them for the further ranking.

Pre-processing and Post-processing of Relevance Weights. Relevance flooding considers relevance weights only on vertices. To account for the relevance weights on edges, we transform them into the weights of incident vertices before the flooding. The pre-processing consists of two steps: assignment of missing relevance weights and transformation of relevance weights. If a graph element is not annotated by a relevance weight, then the default value is assigned to it. 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 ends: the source and target vertices. (2) Given a set of K incident edges to a vertex v_i, the relevance weight of a vertex $\omega(v_i)$ is the sum of the square root of edges' relevance weights $\omega(e_j)$, which are incident to the vertex v_i, and its initial relevance weight $\omega^{init}(v_i)$ (if any) like in Equation 1.

The post-processing is conducted after the subgraph search; it prepares the weights for the ranking. By default, the user-defined weights are used in the 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 (see Equations 2 – 4). If we want to use the relevance flooding weights for the ranking, we have to derive the weights for edges by multiplying the weights of their sources and targets (Equation 3).

$$\omega(v_i) = \sum_{j=1}^{K} \sqrt{\omega(e_j)} + \omega^{init}(v_i) \quad (1)$$

$$\omega^{min}(e_j) = 1/M \quad (2)$$

$$\omega(e_i) = \omega(e_i^{source}) * \omega(e_i^{target}) \quad (3)$$

$$\omega^{min}(v_i) = 1/N \quad (4)$$

Relevance Flooding. The goal of relevance flooding is to annotate all vertices in a query graph by relevance weights. It takes place if not all vertices of a query graph have user-defined relevance weights. This is necessary to allow the subgraph search based on relevance weights and to facilitate the early detection of the most relevant parts of a query graph, which are specified by relevance weights. The algorithm for relevance flooding is based on similarity flooding [9], where two schemes are matched by comparing the similarity of their vertices. We extend this algorithm to propagate the relevance weights to all vertices in a query graph and to keep the initial user-defined relevance weights.

The relevance flooding takes several observations into account: 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 the flooding.

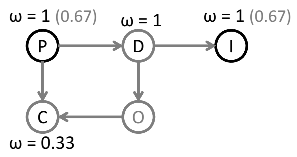
Final edited form was published in "Advances in Databases and Information Systems: 18th East European Conference. Ohrid 2014", S. 112–125, ISBN 978-3-319-10933-6
http://dx.doi.org/10.1007/978-3-319-10933-6_9

Algorithm 1. Relevance Flooding

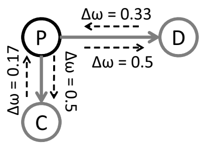
```
 1: for all vertex $v_i$ in query graph $G_q$ do
 2:     if $v_i.getWeight() > 0$ then                              ▷ if a vertex has a weight
 3:         $\omega = v_i.getWeight()$                              ▷ store a weight in $\omega$
 4:         $neighbors = getNeighbors(v_i)$                          ▷ take all direct neighbors
 5:         $\Delta\omega = \omega/neighbors.size()$                 ▷ calculate a propagation weight
 6:         for all $neighbors_j$ in $neighbors$ do
 7:             $neighbors_j.addPropWeight(\Delta\omega)$            ▷ store a propagation weight
 8: for all vertex $v_i$ in query graph $G_q$ do
 9:     $v_i.increaseWeight()$           ▷ increase all weights with propagation weights
10: $max(\omega) = 0$
11: for all vertex $v_i$ in query graph $G_q$ do
12:     if $max(\omega) < v_i.getWeight()$ then
13:         $max(\omega) = v_i.getWeight()$           ▷ find a vertex with the maximal weight
14: for all vertex $v_i$ in query graph $G_q$ do
15:     if $v_i.getInitWeight() > 0$ then       ▷ if a vertex has a user-defined weight
16:         $v_i.setWeight(v_i.getInitWeight())$                     ▷ reset to an initial weight
17:     else
18:         $v_i.setWeight(v_i.getWeight()/max(\omega))$             ▷ normalize a weight
19: $sum = 0$
20: for all vertices $v_i$ in $G_q$ do          ▷ calculate a difference between iterations
21:     $sum = sum + (v_i.getPrevWeight() - v_i.getWeight())^2$
22: if $sum <= \epsilon$ OR $\kappa >= longestPath$ then
23:     $terminateFlooding()$                                   ▷ check termination conditions
```

Relevance flooding works as described in Algorithm 1. In the main part 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 the termination condition we can use a threshold ϵ for the difference of relevance weights of two subsequent iterations or the number of iterations κ, which corresponds to the size of the longest path between two vertices in a query graph.

Following our example in Figure 1(a) and assigned relevance weights $\omega(v_P) = \omega(v_I) = 1$, at each iteration we propagate the equal relevance weights to all direct neighbors (an exemplary weight propagation during the second iteration is shown in Figure 3(b)). During the flooding we do not consider the direction of edges, because the processing of a graph can easily be done in both directions without any additional efforts. After the first iteration, vertices D, C get the propagated relevance weights from P, I according to the locality property (Figure 3(a)). Vertex O still remains without relevance weight. After each iteration, we normalize the relevance weights to the highest value and set those of them



(a) The first iteration



(b) Distribution

		Iteration			
Vertex	Initial	1st	2d	3d	4st
P	1	1	1	1	1
D	0	1	1	1	1
I	1	1	1	1	1
C	0	0.33	0.33	0.34	0.34
O	0	0	0.06	0.09	0.1

(c) Flooding

Fig. 3. Relevance flooding: gray relevance weights show the case, where the initial relevance weights are not set back

back to initial values that have weights defined by the user. The gray relevance weights in brackets for vertices P, I show the weights without reset. We repeat the process, until it converges according to the specified threshold ϵ or when the number of iterations κ has exceeded the longest path between two vertices ($\kappa = 4$). The results of relevance flooding are presented in Figure 3(c).

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. We treat a traversal path between all relevant elements in a query graph as a cost-based optimization, where we maximize the relevance of a path.

The search of subgraphs is modeled by the GraphMCS algorithm [15], a depth-first search for property graphs, discovering maximum common subgraphs between a query and a data graph. First, we choose the first vertex to process. The vertices with highest relevance weights are prioritized and processed first. Second, we process such an incident edge of the selected starting vertex that has a target vertex defined by the highest relevance weight. Finally, this process continues till all vertices and edges in a query graph are processed. If a query edge is missing in a data graph, then the system adjusts the search dynamically: it selects the incident edge with the next highest relevance weight or revises the search from all possible target vertices.

The relevance-based search chooses a next edge to process dynamically based on relevance weights of edges' ends. If several vertices have the same weight, then the edge that has a vertex with minimal cardinality or minimal degree is chosen to be processed. The proposed strategy steers the search in the most relevant direction first, guaranteeing the early discovery of the most relevant parts.

4.4 Rank Calculation

The ranking is based only on the discovered subgraphs, the difference graph does not influence the rating score. 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

Final edited form was published in "Advances in Databases and Information Systems: 18th East European Conference. Ohrid 2014", S. 112–125, ISBN 978-3-319-10933-6
http://dx.doi.org/10.1007/978-3-319-10933-6_9

$$rating(G'_d) = \sum_{i=1}^{i=N} \begin{cases} \omega(v_i) & \text{, if } v_i \in G'_d \\ 0 & \text{, otherwise} \end{cases} + \sum_{j=1}^{j=M} \begin{cases} \omega(e_j) & \text{, if } e_i \in G'_d \\ 0 & \text{, otherwise} \end{cases} \quad (5)$$

Following our example in Figure 1, the rating of the discovered subgraph in Figure 1(b) before normalization equals to $rating = 3$ by default or $rating = 5.68$ by using the relevance flooding weights from the fourth iteration (see Figure 3(c)).

5 Evaluation

In this section, we compare top-k differential queries and unranked differential queries. We describe the evaluation setup in Section 5.1 and compare both approaches in Section 5.2. Then, we present and interpret the scalability of the top-k differential queries in Section 5.3.

5.1 Evaluation Setup

We implemented a property graph model on the top of an in-memory column database system with 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 (new attributes can efficiently be added and removed) and compression for sparsely populated columns like in [2,13]. This enables schema-flexible storage of data without a predefined rigid schema. Our prototypical graph database supports insert, delete, update, filter based on attribute values, aggregation, and graph traversal in a breadth-first manner in backward and forward directions with the same performance.

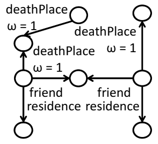
Data and queries are specified as property graphs. In a query, each graph element can be described with predicates for attribute values. To specify a dedicated vertex, we use its unique identifier.

As a data set, we use a property graph constructed from DBpedia RDF triples, where labels represent attribute values of entities. This graph consists of about $20K$ vertices and $100K$ edges. We have tested each case for each query ten times and have taken the average runtime as a measure.

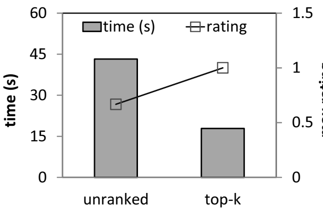
5.2 General Comparison

We constructed an exemplary query shown in Figure 4(a) and marked three edges of the type “deathPlace” with relevance weight $\omega = 1$. The unranked differential query delivers results with a lower maximal rating and exhibits longer response times than the top-k differential query (Figure 4(b)). The top-k differential query discovers more subgraphs of higher ratings than the unranked differential query (Figure 4(c)). The unranked query also discovers the graphs with low ratings.

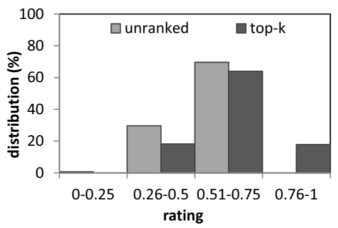
Final edited form was published in "Advances in Databases and Information Systems: 18th East European Conference. Ohrid 2014", S. 112–125, ISBN 978-3-319-10933-6
http://dx.doi.org/10.1007/978-3-319-10933-6_9



(a) Query graph



(b) Performance

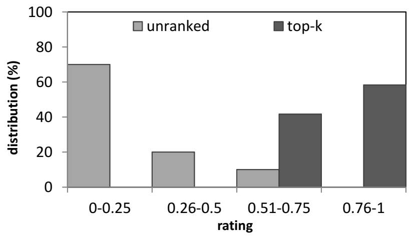


(c) Ranking

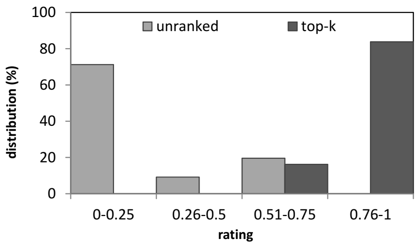
Fig. 4. Evaluation of unranked and top-k differential queries

5.3 Performance Evaluation

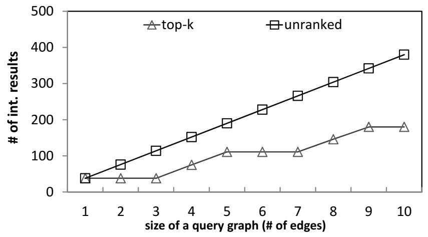
We evaluate two kinds of query graphs, one for the path topology and one for the zigzag topology. The query for the path topology consists of edges of the same type "successor", and the first edge is marked by a relevance weight. The



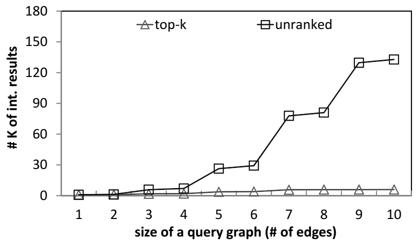
(a) Ranking for the path (10 edges)



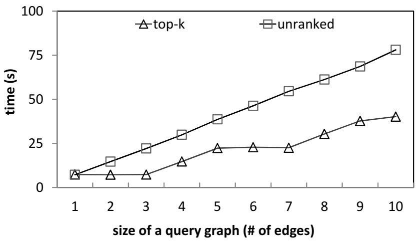
(b) Ranking for the zigzag (10 edges)



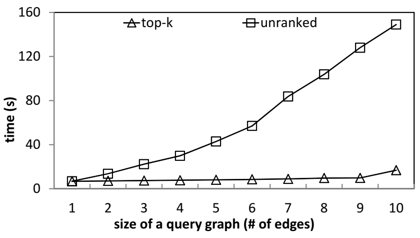
(c) Intermediate results for the path



(d) Intermediate results for the zigzag



(e) Response time for the path



(f) Response time for the zigzag

Fig. 5. Performance evaluation for the differential query and top-k differential query

Final edited form was published in "Advances in Databases and Information Systems: 18th East European Conference. Ohrid 2014", S. 112–125, ISBN 978-3-319-10933-6
http://dx.doi.org/10.1007/978-3-319-10933-6_9

query for the zigzag topology consists of edges of two types "birthPlace" and "deathPlace". It starts with "birthPlace" marked by a relevance weight and is extended incrementally by a new edge for "deathPlace", then "birthPlace" etc.

We compare rating distributions for the largest query (ten edges in a query graph) in Figures 5(a)-5(b). The most of the results delivered by the unranked differential query have low ratings, while the proposed solution provides at least 60% of its results with the highest ratings. We increase the size of a query graph from one edge up to ten edges and evaluate the scalability of the proposed solution. The size of intermediate results grows linearly with the number of edges in a query graph (Figures 5(c)-5(d)), and it is lower than at least one order of magnitude for the top-k differential query. This can be explained by the elimination of low-rated subgraphs from the search. The response time evaluation exhibits the steep decrease for the top-k differential query (Figures 5(e)-5(f)). From this we can conclude, the top-k differential query is more efficient than the unranked differential query: it delivers results with a higher rating score, omits low-rated subgraphs, and consumes less processing time.

6 Conclusion

Heterogeneous, evolving data requires a new kind of storage supporting evolving data schema and complex queries over diverse data. This requirement can be implemented by graph databases offering the property graph model [12]. To express graph queries correctly over diverse data without any deep knowledge of the underlying data schema is a cumbersome task. As a consequence, many queries might return unexpected or even empty results. To support a user in such cases, we proposed differential queries [15] that provide intermediate results of a query processing and difference graphs as the reasons of an empty answer.

In [15] we showed that the result of a differential query can be too large to be manually studied by a user. Therefore, the number of results has to be reduced, and the differential queries have to provide a ranking of their results based on a user's intention. To address these issues, we extend the concept of differential queries and introduce top-k differential queries that rank answers based on a user's preferences. These preferences are provided by a user in a form of relevance weights to vertices or edges of a query graph. Top-k differential queries (1) allow marking more relevant graph elements with relevance weights, (2) steer the search so that more relevant parts of a query graph are discovered first, (3) adjust the search dynamically in case of missing edges based on relevance weights, and (4) rank results according to the relevance weights of discovered elements. The evaluation results showed that more meaningful results are discovered first according to a user's preferences. Our proposed solution delivers results only with high rating scores and omits the graphs with low ratings. Our approach also shows good scalability results with an increasing number of edges in a query graph. In the future, we would like to speedup top-k differential queries with database techniques like indexing and pre-sorting to allow even faster processing. We also want to enhance the system with an online adaptive propagation of relevance weights based on a user's feedback.

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

References

1. Amer-Yahia, S., Koudas, N., Marian, A., Srivastava, D., Toman, D.: Structure and Content Scoring for XML. Proc. of VLDB Endow., 361–372 (2005)
2. Bornhövd, C., Kubis, R., Lehner, W., Voigt, H., Werner, H.: Flexible Information Management, Exploration and Analysis in SAP HANA. In: DATA, pp. 15–28 (2012)
3. Chapman, A., Jagadish, H.V.: Why not? In: Proc. of ACM SIGMOD, pp. 523–534. ACM, New York (2009)
4. Fan, W., Wang, X., Wu, Y.: Diversified top-k graph pattern matching. Proc. of VLDB Endow. 6, 1510–1521 (2013)
5. Gupta, M., Gao, J., Yan, X., Cam, H., Han, J.: Top-k interesting subgraph discovery in information networks. In: Proc. of ICDE, pp. 820–831. IEEE (2014)
6. Huang, J., Chen, T., Doan, A., Naughton, J.F.: On the provenance of non-answers to queries over extracted data. Proc. of VLDB Endow. 1, 736–747 (2008)
7. Kasneci, G., Suchanek, F., Ifrim, G., Ramanath, M., Weikum, G.: Naga: Searching and ranking knowledge. In: Proc. of ICDE, pp. 953–962. IEEE (2008)
8. McGregor, J.J.: Backtrack search algorithms and the maximal common subgraph problem. Software: Practice and Experience 12(1), 23–34 (1982)
9. Melnik, S., Garcia-Molina, H., Rahm, E.: Similarity flooding: A versatile graph matching algorithm and its application to schema matching. In: Proc. of ICDE, pp. 117–128. IEEE (2002)
10. Mottin, D., Marascu, A., Roy, S.B., Das, G., Palpanas, T., Velegrakis, Y.: A probabilistic optimization framework for the empty-answer problem. Proc. of VLDB Endow. 6, 1762–1773 (2013)
11. Prud'hommeaux, E., Seaborne, A.: SPARQL Query Language for RDF. W3C Recommendation (2008)
12. Rodriguez, M.A., Neubauer, P.: Constructions from dots and lines. Bulletin of the American Society for Inf. Science and Technology 36(6), 35–41 (2010)
13. Rudolf, M., Paradies, M., Bornhövd, C., Lehner, W.: The Graph Story of the SAP HANA Database. In: BTW, pp. 403–420 (2013)
14. Tran, Q.T., Chan, C.Y.: How to conquer why-not questions. In: Proc of ACM SIGMOD, pp. 15–26. ACM, New York (2010)
15. Vasilyeva, E., Thiele, M., Bornhövd, C., Lehner, W.: GraphMCS: Discover the Unknown in Large Data Graphs. In: EDBT/ICDT Workshops, pp. 200–207 (2014)
16. Zhu, Y., Qin, L., Yu, J.X., Cheng, H.: Finding top-k similar graphs in graph databases. In: Proc. of EDBT, pp. 456–467. ACM (2012)
17. Zou, L., Chen, L., Lu, Y.: Top-k subgraph matching query in a large graph. In: Proc. of the ACM First Ph.D. Workshop in CIKM, pp. 139–146. ACM (2007)