Elena Vasilyeva, Maik Thiele, Christof Bornhövd, Wolfgang Lehner

Considering User Intention in Differential Graph Queries

Erstveröffentlichung in / First published in:

Journal of Database Management. 2015, 26(3), S. 21 – 40 [Zugriff am: 05.11.2020]. IGI Global. ISSN 1533-8010

DOI: https://doi.org/10.4018/JDM.2015070102

Considering User Intention in Differential Graph Queries

Elena Vasilyeva, SAP SE, Dresden, Germany

Maik Thiele, Database Technology Group, Technische Universität Dresden, Dresden, Germany

Christof Bornhövd, SAP Labs, LLC, Palo Alto, CA, USA

Wolfgang Lehner, Database Technology Group, Technische Universität Dresden, Dresden, Germany

ABSTRACT

Empty answers are a major problem by processing pattern matching queries in graph databases. Especially, there can be multiple reasons why a query failed. To support users in such situations, differential queries can be used that deliver missing parts of a graph query. Multiple heuristics are proposed for differential queries, which reduce the search space. Although they are successful in increasing the performance, they can discard query subgraphs relevant to a user. To address this issue, the authors extend the concept of differential queries and introduce top-k differential queries that calculate the ranking based on users' preferences and significantly support the users' understanding of query database management systems. A user assigns relevance weights to elements of a graph query that steer the search and are used for the ranking. In this paper the authors propose different strategies for selection of relevance weights and their propagation. As a result, the search is modelled along the most relevant paths. The authors evaluate their solution and both strategies on the DBpedia data graph.

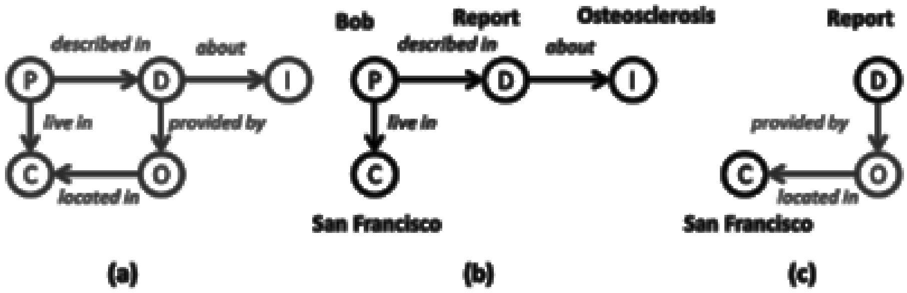
Keywords: Differential Queries, Flooding, Graph Databases, Result Ranking, Routing, Top-K Queries

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

DOI: 10.4018/JDM.2015070102

Figure 1. Differential query and its results (Vasilyeva, Thiele, Bornhövd, & Lehner, 2014): (a) Differential query, (b) Discovered subgraph, (c) Difference graph



empty answer, we already proposed the notion of a differential query (Vasilyeva, Thiele, Bornhövd, & Lehner, 2014), a graph query (for example see Figure 1(a)) which result consists 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),
2. A difference graph reflecting the remaining part of a query like in Figure 1(c).

Although our approach (Vasilyeva, Thiele, Bornhövd, & Lehner, 2014) 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 graph query with 10 edges. A user has two alternatives to deal with so many answers. The results can be ranked based on a scoring function, but the size of the result set stays the same; as a consequence, the response time can be very high. Alternatively, the results can be pruned by optimization strategies that reduce the number of traversals for a query based on cardinality and degree of a query's vertices. This approach reduces significantly the response time. As a negative side effect, optimization strategies may 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 (Vasilyeva, Thiele, Bornhövd, & Lehner, 2014). These queries allow the user to mark vertices, edges, or entire subgraphs of a graph query with relevance weights showing how important graph elements are in a query. To make the search of a top-k differential query with multiple relevance weights possible, we present two algorithms for the propagation of relevance weights: relevance flooding and broadcasting. 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 and to rank individual answers according to the user's interest.

RELATED WORK

In this section we present solutions for "Why Not?" and "Why Empty?" queries, ranking of query results, and flexible query answering.

"Why Not?" Queries

The problem of unexpected answers is generally addressed by "Why Not?" queries (Chapman & Jagadish, 2009) determining why items of interest are not in the result set. It is assumed that the size and complexity of data prevents 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. A "Why Not?" query then could be "Why are the items with predicate P not in the result set?" These queries are classified into two groups based on the explanations they provide: provenance-based and query rewriting.

The first group, provenance-based methods, generates the following explanations: query-based (Chapman & Jagadish, 2009; Bidoit, 2014), instance-based (Huang, Chen, Doan, & Naughton, 2008; Herschel & Hernández, 2010; Herschel, Hernández, & Tan, 2009), and hybrid explanations (Herschel, 2013). Query-based explanations (Chapman & Jagadish, 2009) detect operators of a query tree, rejecting the interesting tuples from the result set. For this purpose, the original query is manipulated. The extension of this work (Bidoit, 2014) allows to find a full solution for query-based answers. Instance-based explanations (Huang, Chen, Doan, & Naughton, 2008; Herschel & Hernández, 2010; Herschel, Hernández, & Tan, 2009) show how a data source has to be modified to deliver missing answers. These explanations fit the data integration tasks, because an extraction process can fail and result in wrong data or if data sources are of different levels of trust. In contrast to above presented approaches for relational databases, we do not deal with a query tree generated for a query execution plan and our source of data is fully trusted. Moreover, our solution investigates missing structural parts of a graph query that prevent a graph processing engine from delivering a non-empty result set.

The second group, query rewriting methods, modifies a query in such a way that it delivers items of interest (Tran & Chan, 2010; Islam, Liu, & Zhou, 2012). A response to a modified query includes original query results and items of interest (Tran & Chan, 2010) or unexpected results are removed from the result set (Islam, Liu, & Zhou, 2012). The second approach involves a user in a query refinement process who has to provide an exemplar subset of tuples to be delivered. A relational "Why Not?" query studies the problem only on the level of a vertex and not on the level of a subgraph. To apply the above presented solutions, the notion of graph and its refinement have to be introduced.

"Why Empty?" Queries

A solution for the problem of empty answers can be modeled by automatic or interactive "Why Empty?" queries based on query rewriting techniques.

In automatic solutions for the query refinement, additional information is required like context, preliminary information, or historical data. An original query is extended by constraints for a result set that have to be guaranteed (Chaudhuri, 1990). Each modification operator for a query is applied according to its application conditions and rules. Automatic techniques for query rewriting are commonly used in recommender systems (Jannach, 2007).

In interactive solutions, a user is integrated in a query refinement process. A user decides which query candidate has to be followed. For example, an optimization-based interactive query relaxation framework for conjunctive queries (Mottin, Marascu, Roy, Das, Palpanas, &

Velegrakis, 2013) 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. The relaxation process terminates (1) if a proposed query candidate has delivered a non-empty result set or (2) if such a query candidate was reached that results in an empty answer and cannot be further relaxed. This approach (Mottin, Marascu, Roy, Das, Palpanas, & Velegrakis, 2013) 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 (Mottin, Marascu, Roy, Das, Palpanas, & Velegrakis, 2013). Moreover, we do not modify queries, but we provide intermediate results and calculate difference graphs as explanations for a user.

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 matches (Zhu, Qin, Yu, & Cheng, 2012; Zou, Chen, & Lu, 2007). These ranking strategies differ in the way 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 (Zhu, Qin, Yu, & Cheng, 2012). If a database maintains a single large data graph, then the approach of top-k subgraph matches (Zou, Chen, & Lu, 2007) can be applied. In this context, a data graph has naturally a hierarchical structure that can be used for (1) index construction and (2) data subgraphs clustering for effective pruning. These solutions do not consider any relevance function for a graph query which is paramount in our setup.

To rank the results, an "interesting" function (Gupta, Gao, Yan, Cam, & Han, 2014), relevance and distance functions (Fan, Wang, & Wu, 2013), or estimation of confidence, informativeness, and compactness (Kasneci, Suchanek, Ifrim, Ramanath, & Weikum, 2008) can be used. An "interesting" function (Gupta, Gao, Yan, Cam, & Han, 2014) can be 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. The matching problem can be revised by the concept of "output node" (Fan, Wang, & Wu, 2013), 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. Additional semantic information can also facilitate the estimation of scoring functions (Kasneci, Suchanek, Ifrim, Ramanath, & Weikum, 2008). In contrast, we assume that the data graph has the maximal confidence; our user is interested in subgraph matching queries without considering any 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.

The top-k processing is also used in XML repositories (Amer-Yahia, Koudas, Marian, Srivastava, & Toman, 2005). 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

Solution for an empty-answer problem for graph data can be modeled by approximate queries or keyword-based queries. In the first group, approximate queries (Mandreoli, Martoglia, Villani, & Penzo, 2009) are supported by a flexible model for querying graph-modeled data that is not bounded to any specific graph data model. Approximation works for both: the vocabularies and the structure. A top-k list of approximated embedding are proposed to a user as a result. This work considers semantic relatedness between vertices to identify if the connection is meaningful and deals with incorrect annotations for edges and vertices. In contrast, we assume that descriptions are grammatically correct, and we discover missing structural patterns of a query. The second group, keyword-based queries, does not include topology information, which is of a main focus of our work. Therefore, these studies are not discussed here.

The empty-answer problem can also be solved by introducing uncertain predicates in queries (Zhou, Shaverdian, Jagadish, & Michailidis, 2010). Zhou et al. assume that the data graph itself is exact, and a user has limited knowledge about predicate values for vertices or edges. A user has to define which predicates are uncertain. Then the query processing engine automatically generates a set of exact predicates matching to uncertain predicates, which a user has to annotate with probability. After composition of predicates a list of exact queries is generated, which is ranked based on given probabilities. In contrast, in our solution a user is only slightly involved into the query answering process: a user has to choose only which vertices are relevant. Alternatively, our query processing engine automatically indicates relevance weights. Moreover, we focus on the structure of a graph query and not on the predicates of vertices and edges.

A different approach tackling the problem of overspecified queries can be modeled by the SPARQL language (Prud'hommeaux & Seaborne, 2008). SPARQL provides the OPTIONAL clause, which allows processing of a graph query if a statement or an entire subgraph is missing in a data graph. The UNION clause allows specification of 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 graph query to derive results, this requires good knowledge of SPARQL. Moreover, this language does not support relevance weights on a graph query directly, and a user cannot influence directly the search in the database. Furthermore, it does not support the calculation of difference graphs.

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 (Rodriguez & Neubauer, 2010; Vasilyeva, Thiele, Bornhövd, & Lehner, 2014), 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.

Differential Queries

If a user receives an empty result set from a graph database, a differential query can be triggered that investigates the reasons of an empty result (Vasilyeva, Thiele, Bornhövd, & Lehner, 2014). The differential query is the initial graph query delivering an empty answer, marked by a keyword. In our work we focus on such queries that have to be specified in the form of a graph, such as subgraph matching queries. In order to provide some insights into the "failure" of a query, a user receives intermediate results of query processing consisting of two parts: data subgraphs and missing parts of the original graph query. The first part consists of a maximum common subgraph between a data graph and the graph query that was discovered by any maximum common subgraph algorithm suitable for property graphs. This can be for example the McGregor maximum common subgraph algorithm (McGregor, 1982). The second part reflects a difference graph – a "difference" between a graph query 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 triggers 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 informaon 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 graph query. 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 6 (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 (Vasilyeva, Thiele, Bornhövd, & Lehner, 2014): 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 graph query and rank results according to user-defined relevance weights.

TOP-K DIFFERENTIAL QUERY PROCESSING

In this section we describe the extended version of our approach (Vasilyeva, Thiele, Bornhövd, & Lehner, 2014) – the relevance-based search and cost-minimizing routing with relevance flooding and broadcasting and the detection of an optimal traversal path through a differential query, and ranking of results.

Top-K Differential Queries

We define a *top-k differential query* as a directed graph $G_q^k = \left(V, E, u, f, g, k\right)$ over attribute space $= A_V \dot{\cup} A_E$, 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. $f\left(V\right)$ and $g\left(E\right)$ 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 $\omega\left(v_i\right)$ and edges $\omega\left(e_j\right)$ in a graph query, which annotate graph elements, vertices and/or edges, in a graph query with float numbers $\in \left[0;1\right]$. 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 not affect the definition of top-k differential queries, this is just an additional property for edges and vertices: $\omega\left(V\right) \subseteq f\left(V\right)$ and $\omega\left(E\right) \subseteq g\left(E\right)$.

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 or cost-minimizing routing 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.

Specification 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), if a doctor is more interested in the names of patients and their diseases, then he provides the highest relevance to corresponding vertices: $\omega\left(v_P\right)=1$ and $\omega\left(v_I\right)=1$

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.

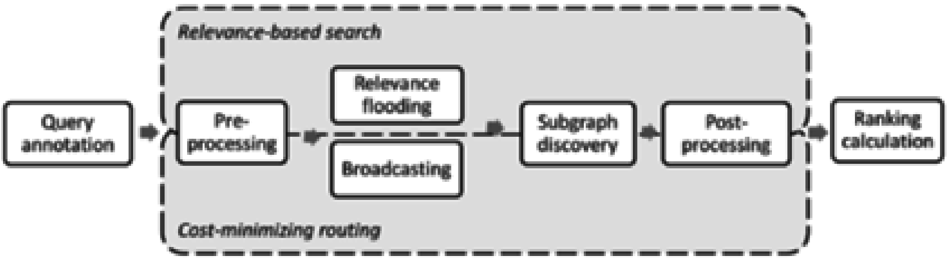
If a user needs support in determining, which elements have to be specified by a weight, a user can use one of the following strategies:

- **Minimum-Representative Element Strategy:** For this strategy, we define cardinality $C\left(v_i\right)$ of a query vertex v_i ($C\left(e_j\right)$ of a query edge e_j) as a number of data vertices (edges) matching to vertex v_i (to edge e_j). A graph element with the least number of representatives is preferred. This strategy is based on the following principle: if a graph element has fewer representatives in a data set, it can be of more interest to a user than more general described graph elements with more instances. This is valid for edges and vertices. This strategy facilitates the reduction of number of answers.
- **Maximum-Connected Vertex Strategy:** For this strategy, we analyze degree of vertices that describes the number of connections (outgoing and incoming) in a graph query. It is calculated for each vertex. Vertices with the maximum degree are preferred and they get the highest relevance weights. This strategy is based on the following principle: if a graph element has the most number of connections to other vertices then it represents more information about a graph query than other vertices. Such vertices represent the topological centers of a graph query. As a consequence, they can be potentially of more interest to a user than less connected vertices. This strategy helps to find larger maximum commons subgraphs.

Relevance-based Search

After a user has annotated the graph query, the relevance-based search is conducted, which is outlined in the dashed box in Figure 2. This is the first option for discovery of relevant subgraphs. At the stage of pre-processing, the relevance weights are transformed into the format required by the relevance flooding: edge relevance weights are converted into the relevance weights of incident vertices. Afterwards, the relevance flooding propagates the weights along the graph query, if at least one vertex does not have a user-defined relevance weight. Then, 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.

Figure 2. Top-k differential query processing



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):

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

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:

$$\omega^{min}(e_j) = \frac{1}{M}$$

$$\omega^{min}(v_i) = \frac{1}{N}$$

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:

$$\omega\left(e_j\right)=\omega\left(e_j^{source}\right)*\omega\left(e_j^{target}\right)$$

Relevance Flooding

The goal of relevance flooding is to annotate all vertices in a graph query by relevance weights. It takes place if not all vertices of a graph query 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 graph query, which are specified by relevance weights. The algorithm for relevance flooding is based on similarity flooding (Melnik, Garcia-Molina, & Rahm, 2002), where two schemas are matched by comparing the similarity of their vertices. We extend this algorithm to propagate the relevance weights to all vertices in a graph query and to keep the initial user-defined relevance weights.

The relevance flooding 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.

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 graph query.

Following our example in Figure 1 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 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 graph query as a cost-based optimization, where we maximize the relevance of a path.

Algorithm 1. Relevance flooding

```
For All vertex $v_i$ In graph query $G_q$
    If $v_i$ has user-defined weight Then
        $\omega$ = weight of $v_i$
        neighbors = neighbors of $v_i$
        $\Delta\omega$ = $\omega$ / number of neighbors (propagation weight)
        For All neighbor In neighbors
            store $\Delta\omega$ in neighbor
For All vertex $v_i$ In graph query $G_q$
    increase weight of $v_i$
initialize max($\omega$)
For All vertex $v_i$ In graph query $G_q$
    If max($\omega$) $\leq$ weight of $v_i$ Then
        max($\omega$) = weight of $v_i$
For All vertex $v_i$ In graph query $G_q$
    If $v_i$ has user-defined weight
        weight of $v_i$ = user-defined weight of $v_i$
    Else
        weight of $v_i$ = weight of $v_i$ / max($\omega$)
initialize sum
For All vertex $v_i$ In graph query $G_q$
    sum = sum + (previous weight of $v_i$ - current weight of $v_i$)$^2$
If sum $\leq \varepsilon$ Or $\kappa \geq$ longest path Then
    terminate flooding
```

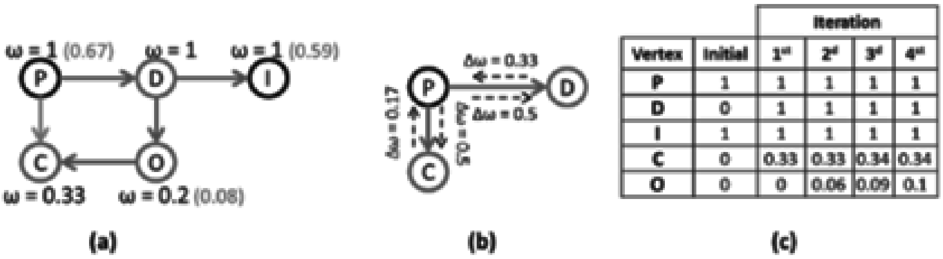
The search of subgraphs is modeled by the GraphMCS algorithm (Vasilyeva, Thiele, Bornhövd, & Lehner, 2014), a depth-first search for property graphs, discovering maximum common subgraphs between a query and a data graph. First, we choose the first vertex to process. The vertices with the highest relevance 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 graph query 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 a minimal cardinality or a 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.

Cost-Minimizing Routing

As an alternative, we can conduct cost-minimizing routing to discover high-relevant subgraphs (see Figure 2), where a graph query is modeled as a routing network. Cost-minimizing routing includes steps similar to the relevance-based search like pre- and post-processing as well as a routing-specific subgraph discovery.

Figure 3. Relevance flooding (Vasilyeva, Thiele, Bornhövd, & Lehner, 2014): gray relevance weights show the case, where the initial relevance weights are not set back: (a) The first iteration (b) Distribution (c) Flooding



		Iteration			
Vertex	Initial	1st	2d	3d	4th
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

Pre-Processing and Post-Processing of Relevance Weights

Similarly to the relevance-based search, we have to pre-process and post-process relevance weights. For routing we consider relevance weights on edges and vertices without any conversion between each other. To model the cost-minimization approach, the system transforms relevance weights into distances for vertices and edges like $dist(v_i) = 1 - \omega(v_i)$ and $dist(e_j) = 1 - \omega(e_j)$, correspondingly. If a graph element does not have any relevance weight, then it has the maximal distance: $dist = 1$. It means graph elements with lower relevance weights add longer distances into processing paths.

At the post-processing step, the system follows the post-processing step of the relevance-based flooding: if a graph element does not have any weight, a default weight is assigned to it.

Broadcasting

Each vertex keeps a routing table that describes the distance to each vertex with a user-defined relevance weight. The entry in a routing table includes three fields: a destination vertex, a distance to this vertex, and an adjacent edge as a first edge to follow. The distance is calculated based on the aggregated distances of all edges and vertices on the path between two vertices:

$$dist(v_m, v_n) = \sum_{i=1}^{K} dist(v_i) + \sum_{j=1}^{L} dist(e_j),$$

where K is a number of vertices $K \le N$ and L is a number of edges $L \le M$ on the path between v_m and v_n.

In the extreme case, when no relevance weights are provided the distance is summed up from the number of hops and vertices between source and destination vertices. The routing table can include several paths to the same vertex via different adjacent edges. If a destination vertex can be reached via all adjacent edges without the crossing the source vertex multiple times, then the number of entries for this destination vertex equals to the number of adjacent edges.

At the beginning for each user-defined vertex the system generates and broadcasts a token. A *token* is a unique identifier which is generated and sent from a user-defined vertex to all vertices in a graph query. The broadcasting of tokens is presented in Algorithm 2. After a vertex has received a token, it checks whether its distance is initialized at line 2 and sets it if required.

Algorithm 2. Broadcasting

```
1: Input: prevVertex, currVertex, token
2: IF token is not set Then
3: id of token = currVertex
4: neighbors = neighbors of currVertex
5: remove prevVertex from neighbors
6: For all neighbor In neighbors
7: newToken = token
8: newToken += distance to neighbor
9: If newToken Not In neighbor Then
10: store newToken in neighbor
11: broadcast newToken
```

Then the system retrieves the set of direct neighbors of the vertex at line 4 and removes from this set that vertex, from which the token came from (line 5). Finally, each neighbor receives the token and increments it. If the token was not earlier processed by the vertex, then it is stored (line 10) and the vertex broadcasts it to its direct neighbors at line 11. The broadcasting continues until all tokens are processed by all vertices. After the broadcast, all vertices have constructed their routing tables. Based on the routing tables, a distance of each vertex can be calculated. The path to process a graph query is based on these distances: vertices with shorter distances are processed first.

Following the example provided earlier, a user defines two relevant vertices: vertices P, I with $\omega = 1$ (see Figure 4(a)). After the broadcasting, the routing tables are constructed as shown in Figure 4(b). If we start at vertex P then we can reach the vertex I via edge e_4 with $dist = 3$ or via edge e_1 with $dist = 7$.

Maximum Common Subgraph Discovery with Relevance Distances

After tokens are broadcasted from user-specified vertices of a graph query, the total distances are calculated for all vertices based on a search distance to each relevant vertex:

$$totalDist\left(v_i\right) = \sum_{l=1}^{K} \min_{e_j} dist_{e_j}\left(v_i, v_l\right)$$

where K is a number of user-defined relevant vertices $K \leq N$

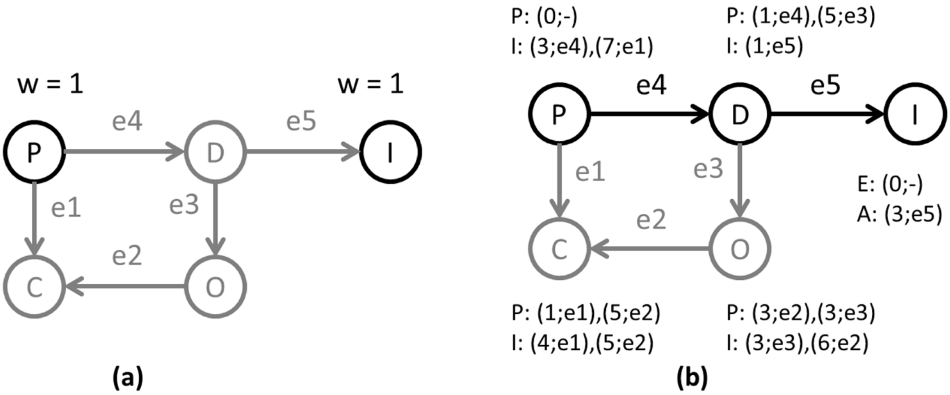
A vertex with the minimal distance is taken as a starting node. The system chooses the shortest distance from its routing table and follows the corresponding edge. According to our example in Figure 4(b),

$$\mathrm{dist}\left(v_P\right) = 3,\ \mathrm{dist}\left(v_C\right) = 5,\ \mathrm{dist}\left(v_O\right) = 6,\ \mathrm{dist}\left(v_D\right) = 2,\ \mathrm{dist}\left(v_I\right) = 3.$$

Vertex D has the minimal distance; it is taken as a starting vertex.

If a specific query edge does not have an instance in a data graph, then two scenarios are possible. We can use the earlier proposed solution with a restart strategy (Vasilyeva, Thiele,

Figure 4. Cost-minimizing routing: (a) Initial setup (b) Constructed routing tables



Bornhövd, & Lehner, 2014): restart search from a possible target of a missing data edge. Alternatively, we can use the advantages of a cost-minimizing routing: we can choose the next shortest path to a dedicated destination and follow it. Every time, when an edge is not presented in a data graph, we adjust search dynamically. If the distances to several destinations are the same, then we decide based on cardinalities of corresponding edges. Following the example in Figure 4(b), if edge e_4 is not presented in a data graph, then the search will continue along edge e_3.

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 graph query G_q, the rating of discovered subgraph G'_d is calculated as follows

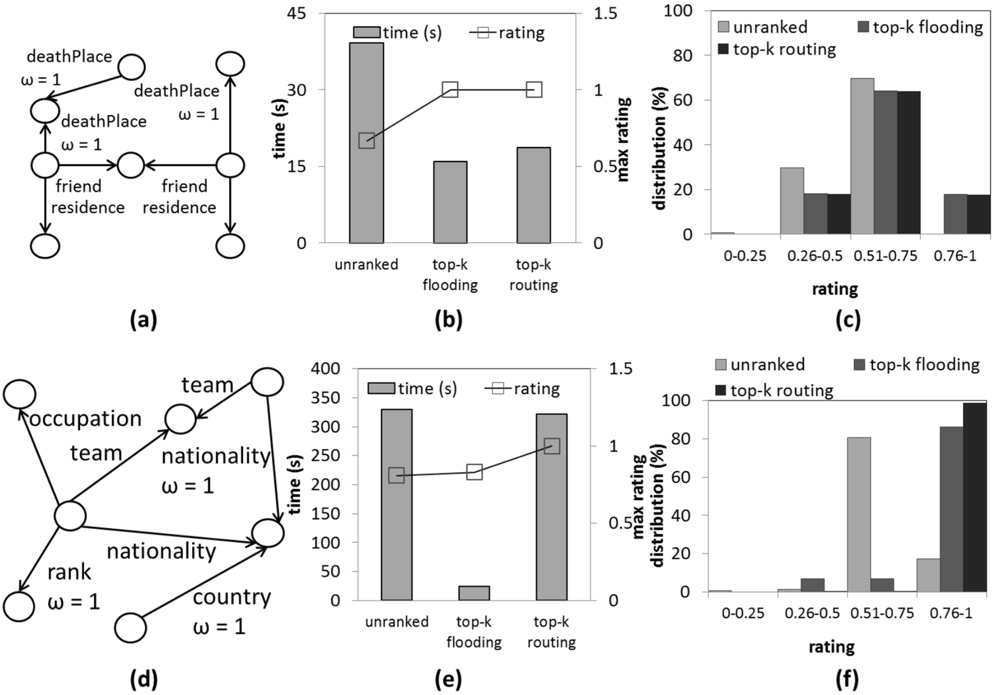
$$rating\left(G'_d\right)=\sum_{i=1}^{N}\begin{Bmatrix}\omega\left(v_i\right), & if\ v_i\in G'_d\\ 0, & otherwise\end{Bmatrix}+\sum_{j=1}^{M}\begin{Bmatrix}\omega\left(e_j\right), & if\ e_j\in G'_d\\ 0, & otherwise\end{Bmatrix}$$

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)).

EVALUATION

In this section, we compare top-k differential queries and unranked differential queries. First, we describe the evaluation setup and compare both approaches. Then, we present and interpret the scalability of the top-k differential queries.

Figure 5. Evaluation of unranked and top-k differential queries (a,d) Graph query (b,e) Performance (c,f) Ranking



(a) (b) (c)

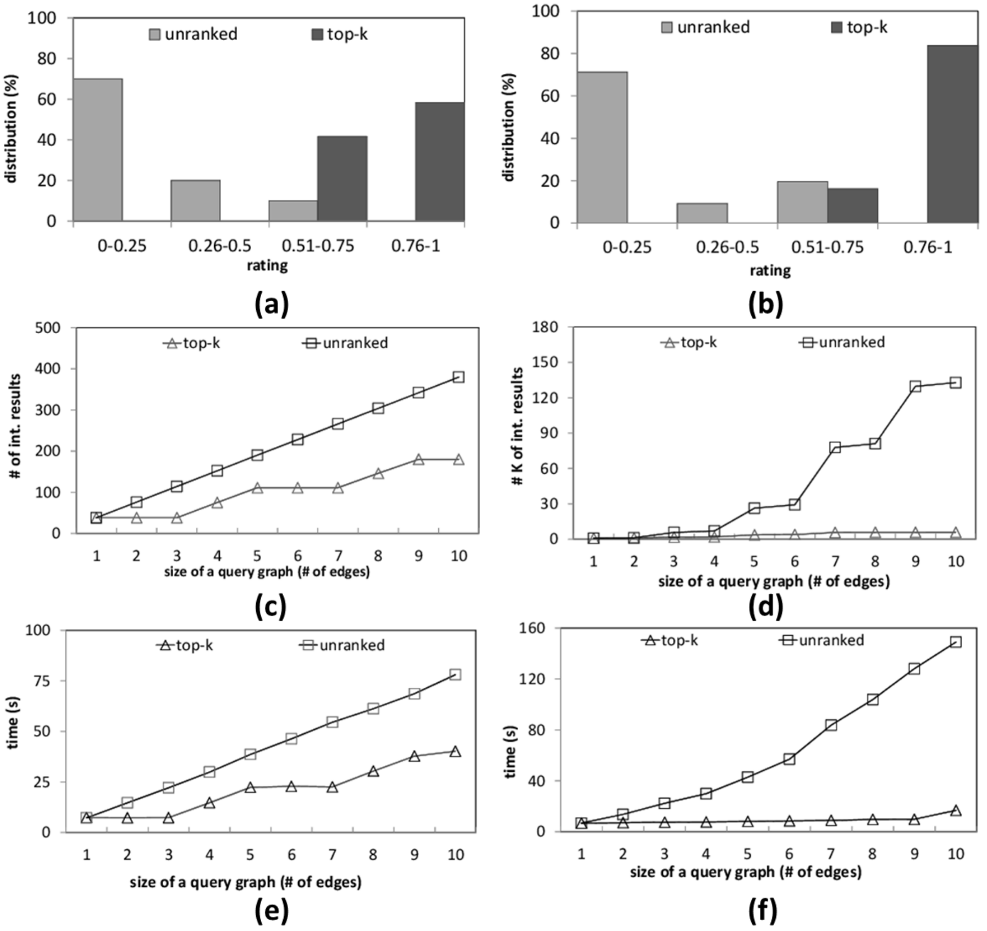
(d) (e) (f)

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 (Bornhövd, Kubis, Lehner, Voigt, & Werner, 2012; Rudolf, Paradies, Bornhövd, & Lehner, 2013). 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.

Figure 6. Performance evaluation for differential and top-k differential queries (Vasilyeva, Thiele, Bornhövd, & Lehner, 2014): (a) Ranking (path, 10 edges) (b) Ranking (zigzag, 10 edges) (c) Intermediate results (path) (d) Intermediate results (zigzag) (e) Response time (path) (f) Response time (zigzag)



General Comparison

We constructed exemplary queries shown in Figure 5(a,d) and marked three edges of the type “deathPlace” in the first query and edges of types “nationality”, “rank”, and “country” in the second query with relevance weights $\omega = 1$. We evaluated both versions of top-k differential queries: with relevance-based search and cost-minimizing routing. The unranked differential query delivers results with a lower maximal rating and exhibits similar or longer response times than the top-k differential queries (Figure 5(b,e)). The top-k differential queries discover more subgraphs of higher ratings than the unranked differential query (Figure 5(c,f)). The unranked query also discovers the graphs with low ratings. Flooding-based and routing-based queries deliver results of similar quality, but the routing procedure exhibits higher response time. Although in the second case routing-based top-k differential query delivers larger graphs, it exhibits longer response time. Based on this observation, we evaluate only flooding-based approach in the following.

Performance Evaluation

We evaluate two kinds of graph queries: path and zigzag topologies. 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 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 graph query) in Figure 6(a,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 graph query 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 graph query (Figure 6(c,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 (Figure 6 (e,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.

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 (Rodriguez & Neubauer, 2010). 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 (Vasilyeva, Thiele, Bornhövd, & Lehner, 2014) that provide intermediate results of a query processing and difference graphs as the reasons of an empty answer.

In (Vasilyeva, Thiele, Bornhövd, & Lehner, 2014) we showed that the result set 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 (Vasilyeva, Thiele, Bornhövd, & Lehner, 2014) 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 graph query.

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 graph query 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 show that more meaningful results are discovered according to a user’s preferences. Our proposed solution delivers results with high rating scores and omits graphs with low ratings. Our approach shows good scalability results with an increasing number of edges in a graph query. In the future, we would like to speed up 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

Amer-Yahia, S., Koudas, N., Marian, A., Srivastava, D., & Toman, D. (2005). Structure and Content Scoring for XML. *Proceedings of the VLDB Endowment* (pp. 361-372).

Bidoit, N. a. (2014, March). Query-Based Why-Not Provenance with NedExplain. *Proceedings of the 17th International Conference on Extending Database Technology (EDBT)* (pp. 145-156).

Bornhövd, C., Kubis, R., Lehner, W., Voigt, H., & Werner, H. (2012). *Flexible Information Management, Exploration and Analysis in SAP HANA* (pp. 15–28). DATA.

Chapman, A., & Jagadish, H. V. (2009). Why Not? *Proc. of ACM SIGMOD* (pp. 523-534). New York, NY, USA: ACM.

Chaudhuri, S. (1990). Generalization and a Framework for Query Modification. *Proceedings of the Sixth International Conference on Data Engineering* (pp. 138-145). Washington, DC, USA: IEEE Computer Society. doi:10.1109/ICDE.1990.113463

Fan, W., Wang, X., & Wu, Y. (2013). Diversified Top-k Graph Pattern Matching. *Proceedings of the VLDB Endowment*, *6*(13), 1510–1521. doi:10.14778/2536258.2536263

Gupta, M., Gao, J., Yan, X., Cam, H., & Han, J. (2014). Top-K Interesting Subgraph Discovery in Information Networks. *Proc. of ICDE* (pp. 820-831). IEEE. doi:10.1109/ICDE.2014.6816703

Herschel, M. (2013). Wondering Why Data Are Missing from Query Results?: Ask Conseil Why-not. *Proceedings of the 22nd ACM International Conference on Conference on Information and Knowledge Management* (pp. 2213-2218). ACM. doi:10.1145/2505515.2505725

Herschel, M., & Hernández, M. A. (2010). Explaining missing answers to SPJUA queries. *Proceedings of the VLDB Endowment*, *3*(1-2), 185–196. doi:10.14778/1920841.1920869

Herschel, M., Hernández, M. A., & Tan, W.-C. (2009, August). Artemis: A System for Analyzing Missing Answers. *Proceedings of the VLDB Endowment*, *2*(2), 1550–1553. doi:10.14778/1687553.1687588

Huang, J., Chen, T., Doan, A., & Naughton, J. F. (2008). On the Provenance of Non-answers to Queries over Extracted Data. *Proceedings of the VLDB Endowment*, *1*(1), 736–747. doi:10.14778/1453856.1453936

Islam, M. S., Liu, C., & Zhou, R. (2012). On Modeling Query Refinement by Capturing User Intent Through Feedback. *Proceedings of the Twenty-Third Australasian Database Conference* (Vol. 124, pp. 11-20). Darlinghurst, Australia, Australia: Australian Computer Society, Inc.

Jannach, D. (2007). Techniques for Fast Query Relaxation in Content-based Recommender Systems. *Proceedings of the 29th Annual German Conference on Artificial Intelligence* (pp. 49-63). Berlin, Heidelberg: Springer-Verlag. doi:10.1007/978-3-540-69912-5_5

Kasneci, G., Suchanek, F., Ifrim, G., Ramanath, M., & Weikum, G. (2008). NAGA: Searching and Ranking Knowledge. *Proceedings of the IEEE 24th International Conference on Data Engineering* (pp. 953-962). IEEE.

Mandreoli, F., Martoglia, R., Villani, G., & Penzo, W. (2009). Flexible Query Answering on Graph-Modeled Data. *Proceedings of the 12th International Conference on Extending Database Technology: Advances in Database Technology* (pp. 216-227). New York, NY, USA: ACM. doi:10.1145/1516360.1516386

McGregor, J. J. (1982). Backtrack search algorithms and the maximal common subgraph problem. *Software, Practice & Experience*, *12*(1), 23–34. doi:10.1002/spe.4380120103

Melnik, S., Garcia-Molina, H., & Rahm, E. (2002). Similarity flooding: A versatile graph matching algorithm and its application to schema matching. *Proc. of ICDE* (pp. 117-128). IEEE. doi:10.1109/ICDE.2002.994702

Mottin, D., Marascu, A., Roy, S. B., Das, G., Palpanas, T., & Velegrakis, Y. (2013). A Probabilistic Optimization Framework for the Empty-answer Problem. *Proceedings of the VLDB Endowment.*, *6*(14), 1762–1773. doi:10.14778/2556549.2556560

Prud'hommeaux, E., & Seaborne, A. (2008). *SPARQL Query Language for RDF*. SPARQL Query Language for RDF.

Rodriguez, M. A., & Neubauer, P. (2010). Constructions from dots and lines. *Bulletin of the American Society for Information Science and Technology*, *36*(6), 35–41. doi:10.1002/bult.2010.1720360610

Rudolf, M., Paradies, M., Bornhövd, C., & Lehner, W. (2013). *The Graph Story of the SAP HANA Database* (pp. 403–420). BTW.

Tran, Q. T., & Chan, C.-Y. (2010). How to ConQueR Why-not Questions. *Proceedings of the 2010 ACM SIGMOD International Conference on Management of Data* (pp. 15-26). New York, NY, USA: ACM. doi:10.1145/1807167.1807172

Vasilyeva, E., Thiele, M., Bornhövd, C., & Lehner, W. (2014). *GraphMCS: Discover the Unknown in Large Data Graphs* (pp. 200–207). EDBT/ICDT Workshops.

Vasilyeva, E., Thiele, M., Bornhövd, C., & Lehner, W. (2014). Top-k Differential Queries in Graph Databases. In Y. Manolopoulos, G. Trajcevski, & M. Kon-Popovska (Eds.), Advances in Databases and Information Systems (Vol. 8716, pp. 112-125). Springer International Publishing. doi:10.1007/978-3-319-10933-6_9

Zhou, H., Shaverdian, A. A., Jagadish, H., & Michailidis, G. (2010). Querying graphs with uncertain predicates. *Proceedings of the Eighth Workshop on Mining and Learning with Graphs* (pp. 163-170). doi:10.1145/1830252.1830273

Zhu, Y., Qin, L., Yu, J. X., & Cheng, H. (2012). Finding Top-k Similar Graphs in Graph Databases. *Proc. of EDBT* (pp. 456-467). ACM. doi:10.1145/2247596.2247650

Zou, L., Chen, L., & Lu, Y. (2007). Top-k Subgraph Matching Query in a Large Graph. *Proc. of the ACM First Ph.D. Workshop in CIKM* (pp. 139-146). ACM. doi:10.1145/1316874.1316897

Elena Vasilyeva is a PhD student at Technische Universität Dresden, Germany. She obtained her master degree at the Petrozavodsk State University (Russia) and Technische Universität Dresden (Germany). She writes her PhD thesis about "Debugging of pattern matching queries delivering unexpected results" in the cooperation with SAP SE Germany. Her research interests include efficient graph processing, pattern matching in graph databases, and time series analysis.

Maik Thiele finished his dissertation on "Quality-Driven Data Production Controlling in Real-Time DW Systems" in May 2010 and received his doctorate with distinction. He was a visiting scientist at UBS Zurich, GfK Nuremberg, and HP Labs Palo Alto. His research interests include large-scale data processing, information extraction, and data integration.

Christof Bornhövd is currently working as a Lead Architect for RMS on a new cloud-based platform for large-scale model execution and big data analytics, where he focuses on scalability and schema extensibility. Prior to joining RMS in 2014, he was a Chief Architect with the SAP HANA development team, where he has been the technical project lead for the SAP HANA Graph project which leverages HANA in-memory technology for flexible graph data management and fast analysis. From 2002 to 2004, Bornhövd was with the database research group at the IBM Almaden Research Center where he was working on database caching and replication for e-business applications and the integration of DB2 and WebSphere. From 2000 to 2002 he worked at HP Labs on CRM and Data Warehousing projects. Christof Bornhövd received his PhD in computer science from TU Darmstadt in 2000.

Wolfgang Lehner is head of the database technology group at Technische Universität Dresden. He obtained his PhD at the University of Erlangen-Nuremberg. Subsequently, he joined the Business Intelligence (BI) group at the IBM Almaden Research Center in San Jose (CA). Since 2002, Prof. Lehner has been conducting research and teaching at TU Dresden and is involved in multiple industrial projects. He was a temporarily visiting scientist at the University of Waterloo (Canada), SAP Palo Alto, GfK Nuremberg, SAP Walldorf, UBS WMBB Zurich, and Microsoft Research in Redmond (WA). Wolfgang Lehner is a member of the Review Board on Computer Science of DFG (German Research Foundation). Since 2014, he is also an appointed member of Academia Europaea (The Academy of Europe).