How and Why is An Answer (Still) Correct? Maintaining Provenance in Dynamic Knowledge Graphs

Garima Gaur  
Computer Science and Engineering,  
Indian Institute of Technology, Kanpur  
India  
garimag@cse.iitk.ac.in

Arnab Bhattacharya  
Computer Science and Engineering,  
Indian Institute of Technology, Kanpur  
India  
arubb@cse.iitk.ac.in

Srikanta Bedathur  
Computer Science and Engineering,  
Indian Institute of Technology, Delhi  
India  
srikanta@cse.iitd.ac.in

ABSTRACT

Knowledge graphs (KGs) have increasingly become the backbone of many critical knowledge-centric applications. Most large-scale KGs used in practice are automatically constructed based on an ensemble of extraction techniques applied over diverse data sources. Therefore, it is important to establish the provenance of results for a query to determine how these were computed. Provenance is shown to be useful for assigning confidence scores to the results, for debugging the KG generation itself, and for providing answer explanations. In many such applications, certain queries are registered as standing queries since their answers are needed often. However, KGs keep continuously changing due to reasons such as changes in the source data, improvements to the extraction techniques, refinement/enrichment of information, and so on. This brings us to the issue of efficiently maintaining the provenance polynomials of complex graph pattern queries for dynamic and large KGs instead of having to recompute them from scratch every time the KG is updated. Addressing these issues, we present HUKA which uses provenance polynomials for tracking the derivation of query results over knowledge graphs by encoding the edges involved in generating the answer. More importantly, HUKA also maintains these provenance polynomials in the face of updates—insertions as well as deletions of facts—to the underlying KG. Experimental results over large real-world KGs such as YAGO and DBpedia with various benchmark SPARQL query workloads reveals that HUKA can be almost 50 times faster than existing systems for provenance computation on dynamic KGs.

1 MOTIVATION

Use of large-scale knowledge graphs (KGs), which model the inter-relationships between various entities that occur in real-life, has become common in many knowledge-centric applications. Apart from the critical role they play in web-search systems (e.g., Google Knowledge Graph, Microsoft Bing Satori, etc.) they are used in applications such as e-governance [22], technical support [45], drug administration [41], scholarly search [38], and many more. While some of the knowledge graphs are carefully hand-crafted, most KGs are automatically constructed using one or more information extraction pipelines over a variety of underlying data sources. As a result, the KG could contain facts obtained through different mechanism and, thus, a query result could be generated through a combination of facts with vastly varying provenance. For instance, it is possible that a fact derived from research studies is joined with a fact from web-based user-generated sources [40]. Tracking fine-grained provenance of individual facts in a knowledge graph alone (as done in [29]) is not sufficient, since we also need to track the provenance of individual query results as well, i.e., which specific facts were involved in generating an individual result.

In this paper, we use provenance polynomials [19] to track the provenance of answers of a graph pattern query over knowledge graphs stored in property-graph engines such as Neo4j. While provenance polynomials have been successfully used by relational databases [3, 11, 19, 35, 36] and RDF triple stores [44] in the past, they are not available as over a property graph store such as Neo4j.

In addition to providing efficient derivation of provenance information for query answers, we focus on making the query answers as well as their provenance derivations easily updated as the underlying KG is updated. The “knowledge” contained in most real-world KGs continually evolves over time. Such evolution could be due to addition of new facts obtained from novel data sources, via knowledge-base completion (KBC) techniques [39], continuous fact refinement approaches [31], etc. Similarly, facts could be culled from the KG when evidence that they are incorrect/invalid is obtained. For example, consider a (fragment of) medical knowledge graph extracted from recent medical news regarding the novel SaRS-CoV-2 virus, one may observe that new symptoms such as “SaRS-CoV-2 causes loss of smell” and “SaRS-CoV-2 causes skin rash” which need to be included in the medical KG. Addition of such new symptoms can result in (re-)classifying many patients, whose symptoms were considered benign earlier, as potential COVID-19 patients now. Note that it is also possible to de-classify some symptoms, resulting in the deletion of the corresponding fact(s) from the KG.

As a result, queries such as “return all COVID-19 positive cases and their contacts in a hospital database” that are very important and are registered as standing queries, may generate significantly different answers with potentially significant effect on the real-world decisions based on the query answers. Therefore, an important technical challenge that needs to be addressed is to determine whether answers to a query is still valid or up-to-date in the face of changes to the KG on which it is evaluated, and if so, demonstrate how so by presenting its evidence. A typical evidence one seeks is the provenance of how a query result was generated by identifying database entries which were responsible for computing each result instance.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.

© 2020 Association for Computing Machinery.
ACM ISBN 978-1-4503-7836-8/YY/MM...
https://doi.org/10.1145/mmnmnm.mmnmnm
These are the why and how provenance of queries. Provenance is also important in demonstrating why the result of a query is no longer valid if it is not so.

A simplistic way to check if a query result is still valid or not is to re-execute all the registered queries after each update to the knowledge graph, and generate an alert if a result of any query differs from the previous one. Clearly, this is impractical due to potentially large number of registered queries, the large scale of knowledge graphs, the need to materialize potentially large volume of query results for comparison and the high frequency of updates to the KG.

One may also wonder if any incremental graph pattern matching algorithms such as TurboFlux [27], IncIsoMat [16], SJ-Tree [12], Graphflow [26], etc. can be used to maintain the basic graph pattern queries. However, none of these algorithms support computing and maintaining the provenance (either why or how) of queries. Further, the presence of projections in the queries we consider (see Sec. 3) make simple pattern finding harder [9]. Thus, incremental graph pattern matching algorithms are not applicable in the settings we consider in this paper.

1.1 Contributions

Our contributions in this paper are:

- We present an algorithmic framework called HUKA (maintaining How provenance under Updates to Knowledge graphs) that achieves the above two objectives by incrementally maintaining the how-provenance of query results in the face of updates to the underlying knowledge graph (Sec. 4). Our solution is based on an adaptation of provenance polynomials [19] for KGs queried using SPARQL. It addresses both the why- and how-provenance of conjunctive queries under dynamic KGs where both insertion and deletion of facts can take place (updates to a fact are modeled using a deletion and a subsequent insertion).
- HUKA compactly maintains, using provenance polynomials, the results of all standing queries registered for maintenance, and the facts that are relevant for the query (Sec. 5). It also maintains information about the subqueries obtained by removing one edge at a time from the basic graph pattern of the query. In order to exploit the shared plans across all queries/subqueries in the workload, it merges individual subquery execution plans generated using AND-OR graphs [30] to form a single global execution tree. When the KG is updated, the updated query results is computed using a filter-and-refine paradigm that helps to quickly recompute the subquery results as well (Sec. 6).
- We use two large real-world knowledge graphs for empirical evaluation, namely, YAGO2 [24] and DBpedia [8]. Our results show that we can update the answer sets along with their provenance polynomials in about 1 second, and can be faster than the baselines by almost 50 times (Sec. 7).

2 RELATED WORK

The need of query result provenance to better understand the role of underlying data in generating a materialized view, called lineage earlier, was felt and fulfilled [13] in data warehouse environments. Query provenance is utilized for various tasks like schema debugging [2], reverse engineering queries [15], trust assessment [37], query computation in probabilistic databases [34], etc.

The first attempt to formalize the concept of provenance is by [10]. They proposed a model to capture the why and where provenance. Later more models capturing different aspects of query provenance, including when [33], how [19], and why not [7], were proposed. These variations provide different dimensions to understand the query results. The seminal work by Green et al. [19] introduced the how provenance and modeled it using provenance semirings. They used the annotated relations to encode the provenance and presented a framework to propagate these annotations along with the query computation. This model proposed a symbolical representation of the derivation process of query answers. Each derivation is represented by a polynomial with each monomial representing a single derivation. The how provenance constitutes of, along with the derivation process, the data items involved in the derivation process. Since how-provenance subsumes why-provenance, we have used the how-provenance model to capture the provenance.
language SPARQL is used to formally express a query as a collection of triple patterns. Similar to a triple, a triple pattern also consists of a subject, a predicate and an object. The subject and object can be variables or can be bound to a particular vertex of the KG. A predicate can also be a variable or can be one of the labels of the edges of the KG. A graph query is formulated as a conjunction of edges/nodes. In this section, we delineate the provenance model, as the underlying KG undergoes updates, i.e., insertion/deletion of edges/nodes. In this section, we delineate the provenance model used and the key idea of maintaining potential matches. In Sec. 5, we discuss the offline phase involving registration of a standing query. When an update occurs, HUKA operates in the online phase as discussed in Sec. 6.

4 THE HUKA FRAMEWORK

Our goal is to maintain the results and their provenance for a set of standing queries that have been registered for maintenance in the system, as the underlying KG undergoes updates, i.e., insertion/deletion of edges/nodes. In this section, we delineate the provenance model used and the key idea of maintaining potential matches. In Sec. 5, we discuss the offline phase involving registration of a standing query. When an update occurs, HUKA operates in the online phase as discussed in Sec. 6.

4.1 Why and How Provenance

The two important classes of query provenance, namely, the why provenance and the how provenance [11] in graphs are next illustrated using our running example KG and the graph pattern query from Fig. 1. Consider Table 1 that lists the results and their provenance of running the example query. Throughout the paper, we will use a running example of a toy knowledge graph depicting relationships between academic researchers and a query pattern illustrated in Fig. 1. Facts such as Sarawagi had Stonebraker as her advisor are represented using a directed edge labeled hadAdvisor connecting the corresponding vertices and is stored as a triple \(( \text{Sarawagi}, \text{hadAdvisor}, \text{Stonebraker})\) in RDF. There are relationships such as coAuthor represented using a bi-directional edge, which correspond to two triples, e.g., \(( \text{Sarawagi}, \text{coAuthor}, \text{Ramakrishnan})\) and \(( \text{Ramakrishnan}, \text{coAuthor}, \text{Sarawagi})\).

We define an example graph pattern query on the above KG which seeks to return professors as well as all collaborators of their students. Note that it only accepts professors and collaborators currently working in an organization. It can be written in SPARQL as follows:

```
SELECT ?prof ?collab WHERE {
  ?stud hasAdvisor ?prof.
  ?prof worksIn ?org2.
  ?collab coAuthor ?stud.
  ?collab hasDegree PhD.
  ?collab worksIn ?org1. }
```

The query graph corresponding to the above query pattern is given in Fig. 1b.

Throughout the paper, we will use a running example of a toy knowledge graph depicting relationships between academic researchers and a query pattern illustrated in Fig. 1. Facts such as Sarawagi had Stonebraker as her advisor are represented using a directed edge labeled hadAdvisor connecting the corresponding vertices and is stored as a triple \(( \text{Sarawagi}, \text{hadAdvisor}, \text{Stonebraker})\) in RDF. There are relationships such as coAuthor represented using a bi-directional edge, which correspond to two triples, e.g., \(( \text{Sarawagi}, \text{coAuthor}, \text{Ramakrishnan})\) and \(( \text{Ramakrishnan}, \text{coAuthor}, \text{Sarawagi})\).

We define an example graph pattern query on the above KG which seeks to return professors as well as all collaborators of their students. Note that it only accepts professors and collaborators currently working in an organization. It can be written in SPARQL as follows:

```
SELECT ?prof ?collab WHERE {
  ?stud hasAdvisor ?prof.
  ?prof worksIn ?org2.
  ?collab coAuthor ?stud.
  ?collab hasDegree PhD.
  ?collab worksIn ?org1. }
```

The query graph corresponding to the above query pattern is given in Fig. 1b.

### Table 1: Query result and provenance of the running example

| ?prof | ?collab | why provenance | how provenance |
|-------|---------|---------------|---------------|
| Stonebraker | Ramakrishnan | \{(e2, e3, e6, e17)\}, \{(e2, e3, e5, e14, e17)\} | \(e2\cdot e3\cdot e6\cdot e17 + e2\cdot e3\cdot e5\cdot e14\cdot e17\) |

### Figure 1: Running example of a toy Knowledge Graph of academic collaborations and a corresponding graph pattern query.

(a) Knowledge Graph

(b) Query pattern
each monomial representing bindings of edges that lead to the answer (Stonebraker, Ramakrishnan). Here, two different sets of edge combinations \{e_2, e_3, e_6, e_8, e_17\} and \{e_2, e_3, e_5, e_14, e_17\} lead to the same result item. In Fig. 1a, subgraphs corresponding to these two different edge bindings are shaded in red and blue respectively. Since how-provenance subsumes and captures more information about generated results than why-provenance, HUKA uses the how-provenance model represented through provenance polynomials.

4.2 Potential Matches

Fundamental to our approach is the concept of potential matches of a query \(Q\). Intuitively, these potential matches of a query correspond to subgraphs of the KG that partially match the given query graph pattern. We maintain the potential matches of a query \(Q\) that require only a single edge insertion to transform them to full matches, i.e., actual results of \(Q\). We now formally define potential matches, and prove how they can be maintained.

**Definition 1 (Potential Match).** Any subgraph \(S\) of the knowledge graph \(G\) which can become an actual match of a query \(Q\) after a single edge insertion is called a potential match. \(\Box\)

Consider a query \(Q\) consisting of \(n\) triple patterns. Suppose \(S \in G\) is a subgraph such that \(n - 1\) triple patterns of the query \(Q\) match and only 1 triple pattern does not match. After a new edge \(e\) that matches with the unmatched triple pattern of \(Q\) is added to the \(S\), the new subgraph, \(S' = S \cup \{e\}\) now becomes an actual match for \(Q\). In other words, \(S' \in A(Q)\), where \(A(Q)\) is the answer-set of the query \(Q\). \(S\) is, therefore, a potential match.

Next, we categorize the potential matches based on the number of matches resulting from the newly inserted edge. Either the newly added edge matches a single triple pattern of the \(Q\) or it matches many triple patterns in the query.

**Definition 2 (1:1 Potential Match).** If a newly added edge \(e\) to a potential match \(S\) matches only one triple pattern of the query \(Q\) such that \(S' = S \cup \{e\}\) becomes an actual match of \(Q\), then the subgraph \(S\) is called a 1:1 potential match, denoted by \(\mathcal{PM}_{1:1}\). \(\Box\)

**Definition 3 (1:m Potential Match).** If a newly added edge \(e\) to a potential match \(S\) matches multiple triple patterns of the query \(Q\) such that \(S' = S \cup \{e\}\) becomes an actual match of \(Q\), then the subgraph \(S\) is called a 1:m potential match, denoted by \(\mathcal{PM}_{1:m}\). \(\Box\)

We illustrate these important concepts using Fig. 1b. The subgraphs \(G_1\) and \(G_2\), shown in Fig. 2a and Fig. 2b respectively, are subgraphs of the example toy KG, and are potential matches of the parent query. Although both are potential matches, \(G_1\) is anticipating the insertion of only one fact \(qp_1^1\) : \(\{\text{collab}, \text{worksIn}, \text{?org1}\}\) to match the query. On the other hand, \(G_2\) needs to get match for two triple patterns \(qp_1^2\) : \(\{\text{prof}, \text{worksIn}, \text{?org2}\}\) and \(qp_2^2\) : \(\{\text{collab}, \text{worksIn}, \text{?org1}\}\) in the query.

Interestingly, a single fact insertion can complete both kinds of potential matches. For instance, if a fact \(\langle \text{Sarawagi}, \text{worksIn}, \text{MIT}\rangle\) is added to the KG, it will match to the query pattern \(qp_1^1\), resulting \(G_1\) to fully satisfy the query. At the same time, the same fact also matches both \(qp_1^2\) and \(qp_2^2\) making even \(G_2\) to satisfy the query.

4.3 Query Categorization

We categorize registered queries into two types, regular and MultiMap, based on the kind of potential matches it may have. Both of these are conjunctive queries and they differ only in the distribution of predicates among triple patterns. We formally define them next.

**Definition 4 (Regular Query).** A query that has only 1:1 potential matches (and no 1:m potential match) is called a regular query, denoted by \(Q_{reg}\).

For example, the query in Fig. 1b is not a regular query as its two triple patterns \(\{\text{prof}, \text{worksIn}, \text{?org2}\}\) and \(\{\text{collab}, \text{worksIn}, \text{?org1}\}\) have the same predicate \(\text{worksIn}\).

**Definition 5 (MultiMap Query).** A query that has 1:m potential matches (possibly in addition to 1:1 potential matches) is called a MultiMap query, denoted by \(Q_{mm}\).

The example query, shown in Fig. 1b, is a MultiMap query with \(G_2\), in Fig. 2b, as a \(\mathcal{PM}_{1:m}\). A necessary condition for a conjunctive query to be a MultiMap query is to have at least two triple patterns with common predicate. However, this condition is not sufficient. We discuss those characteristics in Sec. 5.4 while discussing the registration process of MultiMap queries.

The separation of potential matches and the queries into different classes is important since they need to be handled differently. Our strategy of query registration process is based on the following lemmas.

**Lemma 1.** A \(\mathcal{PM}_{1:1}\) can satisfy one and only one subquery of size \(n - 1\) of a given parent query \(Q\) of size \(n\).

**Proof.** Consider a parent query \(Q = \{t_1, \ldots, t_n\}\) of size \(n\), i.e., having \(n\) triple patterns. A subquery \(Q_i\) of size \(n - 1\) is generated by removing the triple pattern \(t_i\) of \(Q\), i.e., \(Q_i = \{t_1, \ldots, t_{i-1}, t_{i+1}, \ldots, t_n\}\).

Assume that a subgraph \(S\) is a \(\mathcal{PM}_{1:1}\) such that completing the triple \(t_i\) will complete its match to the parent query \(Q\). By definition, \(S\) satisfies the subquery \(Q_i\).

Assume that \(S\) also satisfies another subquery \(Q_j\) (\(i \neq j\)) of \(Q\). This implies that \(S\) has all the matching edges of \(Q_j\). Since \(Q_j\) misses only the triple pattern \(t_i\), it must contain \(t_i\). This is a contradiction and, therefore, \(S\) cannot match any other subquery \(Q_j\).

Hence, \(S\) matches one and only one subquery of \(Q\). \(\Box\)

**Lemma 2.** A \(\mathcal{PM}_{1:m}\) cannot satisfy any subquery of size \(n - 1\) of a given parent query \(Q\) of size \(n\).

**Proof.** Consider a parent query \(Q = \{t_1, \ldots, t_n\}\) of size \(n\), i.e., having \(n\) triple patterns. A subquery \(Q_i\) of size \(n - 1\) is generated by removing the triple pattern \(t_i\) of \(Q\), i.e., \(Q_i = \{t_1, \ldots, t_{i-1}, t_{i+1}, \ldots, t_n\}\).
Assume that a subgraph \( S \) is a \( \mathcal{P} \mathcal{M}_{l,m} \) such that completing the set of triples \( \{ t_1, \ldots , t_k \} \) will complete its match to the parent query \( Q \).

Assume that \( S \) satisfies a subquery \( Q_j \) of \( Q \). Since the subquery \( Q_j \) misses only one triple pattern, \( t_1 \), this implies that \( S \) matches the rest of the \( n-1 \) triple patterns. By definition, this is a contradiction.

Hence, \( S \) cannot match any subquery of \( Q \).

**Lemma 3.** After an edge insertion, a \( \mathcal{P} \mathcal{M}_{l,m} \) satisfies a parent query \( Q \) if and only if the \( \mathcal{P} \mathcal{M}_{l,m} \) satisfies all the subqueries of \( Q \).

**Proof.** (Only if) Suppose a \( \mathcal{P} \mathcal{M}_{l,m} \) satisfies a query \( Q \) consisting of \( n \) triple patterns. This implies that \( S \) has a match for every triple pattern \( t_i \in Q_i \), \( i = 1, \ldots , n \). Consider any subquery \( Q_j \) of \( Q \) that satisfies \( S \). Since it consists of the rest \( n-1 \) triple patterns except \( t_i \), \( S \) necessarily satisfies \( Q_j \).

(If) Since \( S \) satisfies all subqueries, it satisfies, in particular, two subqueries \( Q_1 \) and \( Q_2 \) \( (i \neq j) \) of \( Q \). The triple patterns \( t_i, l = 1, \ldots , n, l \neq i \) match \( S \) due to \( Q_1 \). Since the triple pattern \( t_i \) match due to \( Q_j \), together all the \( n \) triple patterns match. Hence, \( S \) matches the entire query \( Q \).

Next, we show how these observations can be used to preprocess queries during their registration for maintenance.

## 5 Registering Standing Queries

In this section, we discuss in detail the process of registering a query. These queries are called standing queries or continuous queries [6] since their results are always relevant and need to be updated when the KG undergoes changes. When a standing query is registered for maintenance, apart from evaluating its results, HUKA performs the following additional steps: (a) generation of all subqueries obtained by removing one triple pattern at a time from the query and identification of all the potential matches, (b) annotation of the connection points in the knowledge graph, and (c) precomputation of a query processing plan for the subqueries so that they can be recomputed efficiently. In the rest of the section, we describe these steps in detail, first for regular queries and then for MultiMap queries since they need special handling.

### 5.1 Subquery Generation

Since we are interested in only those subqueries that are just one triple pattern less than the registered query being processed, they are all of size \( n-1 \). From the running example query in Fig. 1b, we can generate a number of subqueries by systematically removing each triple pattern specified in the query. This results in \( S \) subqueries as listed in Fig. 3. Notice that in some cases the removal of a triple pattern can lead to generation of two disconnected subqueries.

If a given query is of size \( n \), i.e., has \( n \) triple patterns, removal of a triple pattern can generate at most two subgraphs in the resulting subquery. Denoting these two subgraphs as \( SQ_1 \) and \( SQ_2 \), without any loss of generality, we assume that \( |SQ_1| = k \) and \( |SQ_2| = n-k-1 \) where \( 0 \leq k \leq n-1 \). The subqueries can be categorized into 4 types depending on the size of their subgraphs and the resulting connection points. These types are illustrated in Fig. 4. In each of these illustrations, the gray-shaded node is part of the subgraph it is shown connected to; they have been shown separately only to highlight the subquery generation process.

- **Type I:** These subqueries have a singleton subgraph \( k = 0 \), and are generated when the triple pattern corresponding to the edge of a leaf node of the query graph is removed. For instance, the subquery shown in Fig. 3a, generated by removing \( workIn \) edge. This leads to \( |A(SQ_1)| \) connection points in the knowledge graph. The general subquery structure for this type is given in Fig. 4a.

- **Type II:** In this case, one of the subgraphs contains only one triple pattern, i.e., \( k = 1 \), as shown in Fig. 4b. In this case, we may end up identifying all edges with the predicate of the triple pattern in \( SQ_2 \) as connection points which could be a potentially large number of instances in the knowledge graph. We avoid this, and instead only identify the connection points derived from \( A(SQ_1) \), leading to only \( |A(SQ_1)| \) connection points. Fig. 3c shows an example.

- **Type III:** As shown in Fig. 4c, both \( SQ_1 \) and \( SQ_2 \) contain at least two triple patterns, i.e., \( 2 \leq k < n-2 \). For instance, removal of coAuthor generated two subqueries, shown in Fig. 3b, each of size 2. This results in at most \( |A(SQ_1)| + |A(SQ_2)| \) connection points.

- **Type IV:** In this last type, the removal of a triple pattern from the query does not disconnect the query graph, i.e., \( k = n-1 \). As shown in Fig. 4d, each match of \( SQ_1 \) can result in two potential matches leading to a total of \( 2 \cdot |A(SQ_1)| \) connection points in the graph since they may be annotated from both sides.

It is important to note that in each case, both \( SQ_1 \) and \( SQ_2 \) subgraphs are effectively treated as distinct query patterns and are maintained by HUKA.

Potential matches are identified as subgraphs of the KG that match the subqueries of a parent query.

### 5.2 Connection Point Annotations

After identifying all the potential matches of a query, HUKA annotates the entities corresponding to their connection points. Each connection point result in its entity to be annotated with a tuple \((\text{expRel}, \text{dir}, \text{sqId}, \text{result}, \text{provPoly})\), where \(\text{expRel} \) is the relation predicate the connection point expects, \(\text{dir} \) is the direction of the edge (either outgoing or incoming), \(\text{sqId} \) is the identifier of the subquery that the connection point belongs to, \(\text{result} \) is the value of the query variable that needs to be projected, and \(\text{provPoly} \) is the provenance polynomial corresponding to the potential match.

For instance, consider the subqueries generated by removing the coAuthor edge in Fig. 3b. For ease of exposition only two connections are considered, one for each subquery. The node Ooi in the KG in Fig. 1 matches one of the connection points, and is annotated with the tuple \((\text{coAuthor}, \text{out}, q_1, \text{Ooi}, e_{15}, e_{16})\) as it is expecting an outgoing edge with label \(\text{coAuthor} \). Similarly, node Gehrke has the annotation \((\text{hadAdvisor}, \text{in}, q_3, \text{Ramakrishnan}, e_1, e_3)\). Note that for Gehrke the \(\text{result} \) attribute has value \(\text{Ramakrishnan} \) as it would match the projected variable \(\text{prof} \) of the parent query.

### 5.3 Subquery Execution Plan

As new edges are added to the KG, it is crucial that the results of all the subqueries of registered queries are continually and efficiently maintained. Instead of generating an execution plan each time subquery results have to be updated, HUKA generates an execution plan for each subquery at the time of query registration. These plans are then merged together to form a global plan that tries to exploit shared execution segments.
Local Plan Generation. In order to generate the execution plan for each subquery, HUKA adapts AND-OR trees [30]. These trees are comprised of two types of nodes, AND and OR nodes. They are present at alternate levels in the tree, i.e., each AND node has an OR node as its parent and also as children. An AND node represents a way of deriving the intermediate expression corresponding to its parent OR node, and vice versa. These trees effectively capture all possible derivations of the subquery expressions, with OR nodes corresponding to an intermediate expression in the evaluation of the final subquery expression.

After the AND-OR tree is constructed, it is required to choose the best query plan of the subquery. A greedy heuristic is employed to choose the best plan. Each intermediate expression is assigned a score based on its estimated cardinality. The execution tree is then traversed in a bottom-up manner. At each level \( i \), the node corresponding to the intermediate expression with the lowest cardinality is greedily chosen. No other node at level \( i \) is considered. At the next level \( i + 1 \), its parent nodes are explored. The traversal continues till the root of the tree is reached.

Consider the parent query shown in Fig. 1b. Removal of the triple pattern \((?\text{stud}, \text{hadAdvisor}, ?\text{prof})\) generates the subquery with triple patterns containing \text{hasDegree}, \text{worksIn} and \text{coAuthor} as predicates. A snippet of partial AND-OR tree for this subquery is shown in Fig. 5. In interests of space, relations \text{hasDegree}, \text{worksIn} and \text{coAuthor} are denoted by \(P_1\), \(P_2\) and \(P_3\) respectively. As shown, AND nodes on the second level correspond to intermediate expressions \((P_1 \iff \text{subject} P_2)\) and \((P_2 \iff \text{subject} P_3)\). Similarly, the root node corresponds to the final expression of the subquery. The children of the root node represent the two possible derivations, viz., \(((P_2 \iff \text{subject} P_3) \iff \text{subject} P_1)\) and \(((P_1 \iff \text{subject} P_2) \iff \text{subject} P_3)\).

Global Plan Computation. Once the best plan is chosen for a subquery, it is merged into the global query plan. The global plan is built to reuse the computation of shared intermediate expressions across all subqueries. In a local plan, the subtree rooted at a given node gives its derivation. We start traversing the best local plan in a top-down manner. We do a breadth-first traversal down the best local plan. At each node \( k \), we look for an equivalent expression in the global plan. If an equivalent expression already exists, it implies that the derivation of the expression corresponding to node \( k \) exists. So, we do not explore another derivation of the same expression. If no equivalent expression is found, we add the node \( k \) along with the corresponding equivalent expression to the global tree and move to the next node. The global plan can have multiple roots. Each root represents the final expression of one of the subqueries.

Cardinality Estimation. In order to obtain high-quality execution plans for queries, an accurate cardinality estimation is crucial. The cardinality estimation used in HUKA is specifically designed for RDF knowledge graphs, and captures the neighborhood of entities in the graph. Specifically, HUKA models queries as star-chain shapes, i.e., a sequence of star shaped queries [28]. Based on this, the statistics are maintained as counts of \((s, o)\) pairs that have the same characteristic pair [20] \(P_c\), given by: \(P_c(S_c(s), S_c(o))\) = \{(\text{\(S_c(s), S_c(o), p\))} | S_c(o) \neq \phi \land \exists p : (s, p, o) \in E\} \) where \(S_c\) is the characteristic set. The cardinality estimate of a query, represented as a sequence of \(k\) star-shaped fragments \(C_1, \ldots, C_k\), is the sum of the estimates of the pairs of fragments:

\[
\text{card} = \sum_{i=1}^{k-1} \text{count}(P_c(C_i, C_{i+1}))
\]
5.4 Processing MultiMap Queries

The above discussed steps involving subquery generation, connection point annotation and global execution plan generation work for maintaining regular queries. MultiMap queries, however, require special handling. The presence of more than one triple with the same predicate is a necessary but not a sufficient condition to identify a conjunctive query as a MultiMap query. We next use an example of a non-MultiMap query \( Q \) in Fig. 6 to highlight the characteristics of MultiMap queries.

Characteristics of MultiMap Queries. The query \( Q \) has two pairs of triple patterns with the same predicate: \((\langle ?x, P, q \rangle, \langle ?x, P, q \rangle)\) and \((\langle ?x, P, C \rangle, \langle ?x, P, C \rangle)\). There are two important observations:

- If \( C_1 \neq C_2 \) and relation \( P \) is a one-to-one relation then triples \( \langle ?x, P, C_1 \rangle \) and \( \langle ?x, P, C_2 \rangle \) cannot be mapped to the same edge of a KG. In other words, \( ?x2 \) and \( ?x6 \) cannot be bound to a same node. Propagating this constraint backwards, triples \( \langle ?x, P, ?x2 \rangle \) and \( \langle ?x, P, ?x6 \rangle \) cannot be mapped to the same edge as \( ?x2 \) and \( ?x6 \) cannot be mapped to same node of KG. So, triple patterns participating in two different paths, comprising of one-to-one relations, leading to different bound variables cannot be mapped to same edges of KG. Such a query cannot be MultiMap. For instance, paths from \( ?x1 \) to \( ?x5 \) does not have relation sequence \( P \), but none of the involved triple patterns can be mapped to same KG edge. Thus, \( Q \) is not a MultiMap query.

- Suppose relations \( P_1, P_2 \) and \( P_3 \) are same, say isLocatedIn. Then node \( ?x1 \) and \( ?x5 \) cannot be mapped to same node of KG as a location cannot be located in itself. Thus, triples \( \langle ?x1, P, ?x2 \rangle \) and \( \langle ?x5, P, ?x6 \rangle \) cannot be mapped to same edge. Therefore, a query cannot have a \( PM_{1,m} \) involving variables \( ?x \) and \( ?z \), if a path \( P_1, \ldots, P_j \) from \( ?x \) to \( ?z \) is comprised of the same asymmetric predicate, i.e., \( P_1 = \cdots = P_j = P \) where \( (aPb) \Rightarrow \neg bPa \).

Identification of MultiMap Queries. Based on the above two observations, we identify MultiMap queries and the corresponding predicate \( P \) shared by the triple patterns, \( t_1, \ldots, t_j \), that can match to the same edge. A MultiMap query can have both \( PM_{1,1} \) and \( PM_{1,1} \). A \( PM_{1,i} \) would satisfy one of the subqueries (Lemma 1) but a \( PM_{1,m} \) would not satisfy any subquery before an appropriate edge \( e \) is inserted (Lemma 2). However, when an appropriate edge is inserted, the \( PM_{1,m} \) would satisfy all the subqueries and the parent MultiMap query simultaneously (Lemma 3). Thus, to identify new matches to the parent MultiMap query, we examine the new results of a subquery. Instead of examining all subqueries, only those that are generated by removing a triple pattern that shares the same predicate, are enough to identify a \( PM_{1,m} \).

5.5 Processing to Support Deletions

An edge deletion requires not only the updates to the parent query results, but also to the existing potential matches and the connection point annotations in the KG. In the running example (Fig. 1), if edge \( e_{15} \) is deleted, then the node \( Ooi \) is no longer a connection point. HUKA constructs two inverted index structures: edgeToResult and edgeToCP to efficiently support this. The edgeToResult index maintains a mapping from an edge to the answer set items containing the edge in their provenance (i.e., the presence of the edge that contributed to the item being in the answer set). The edgeToCP index simply maintains the correct state of all the connection points corresponding to the edge.

To build the edgeToResult, the following two sets of information for each query \( Q_i \) are maintained: (a) \( R_i \): the set of result items of query \( Q_i \) along with their corresponding provenance polynomials, and (b) \( E_i \): the set of all edges involved in the provenance polynomials of all the result items \( R_i \).

Based on these, an index is built on edges \( E_i \) that appear in the result set of at least one query. The index entry corresponding to an edge \( e \) consists of the set \( \{(Q_i, L_i)\} \) where \( L_i \) is the list of result items such that \( e \) appears in its associated provenance polynomial.

Similarly, the edgeToCP index is built by considering the provenance polynomials of the potential matches. Each participating edge is mapped to a set of connection points which are annotated with those provenance polynomials.

6 UPDATE PROCESSING

6.1 Insertion

When a new fact is added to the KG, we are first required to find the affected queries and then update the indexes and annotations for handling subsequent updates. Suppose that edge \( e(u, P, v) \) is added to a KG. We start by fetching relevant annotations of \( u \) that have \( P \) as expRel and “out” as dir. Suppose, the relevant annotation has query id sqId as \( q_i \). Recall that each annotation corresponds to a \( PM_{1,i} \) of one of the subqueries. Based on the type of subquery \( q_i \), we proceed as follows:

- **Type I:** If \( q_i \) is a Type I subquery, then we do not need to perform any check further as the insertion of \( e \) with expected relation \( P \) has completed the potential match. For example, in Fig. 1a, if edge \( \text{Sarawagi, worksIn, IITB} \) is inserted, then based on the fact that Sarawagi is expecting an outgoing edge label worksIn is enough to confirm the transition of the corresponding potential matches.

- **Type II:** Recall that this class has only one subquery \( SQ_1 \) of size \( n - 2 \) and an single edge with predicate \( E \). We need a simple lookup to check if node \( u \) has an correct incident edge with label \( E \). The presence of such edge leads to a parent query answer.

- **Type III:** We examine the annotation of \( v \) as well. If \( v \) has an annotation with \( P \) as expRel and “in” as dir, and query id of complimentary subquery of subquery \( q_i \) as sqId. If the required sqId is present, then \( e \) changes the potential match to an actual match. For instance, the connection points Gehrke and Ooi in Fig. 1a correspond to a pair of complimentary subqueries. Therefore, if the edge \( \langle Ooi, coAuthor, Gehrke \rangle \) is added to the KG, it would result into a new answer \( \langle \text{Ramakrishnan, Ooi} \rangle \).

- **Type IV:** We check the annotation of \( v \) and if we find the same subquery id as that in \( u \) and with expRel as \( P \) and “in” as dir, we can conclude that \( e \) has completed the potential match.

The above checks are for \( PM_{1,i} \) only. If the predicate \( P \) of the newly added edge is involved in any of the MultiMap queries, then it
needs special handling. Before reporting new matches to a MultiMap query, the subquery results are updated. The global query plan is browsed in a bottom-up manner and the incremental results of all the intermediate expressions involving the predicate \( P \) are computed. Consequently, we find the incremental results of all the concerned subqueries. Recall that each MultiMap query is associated with one of its subqueries \( S \). The subquery was chosen in a manner that if the newly added edge \( e(u, P, v) \) matches the missing triple pattern of the subquery, then all the new answers of the subquery would match the parent query, and, thus, would be reported as parent query answers.

The new results also correspond to new connection points which are then annotated. Both the indexes \( \text{edgeToResult} \) and \( \text{edgeToCP} \) are also updated.

6.2 Deletion

The deletion algorithm is based on the standard filter-and-refine paradigm where candidate queries are first generated and later confirmed. Assume that an edge \( e_d \) is deleted. First, we identify the provenance polynomials in which this edge is involved and the corresponding results. This is done by fetching the entry corresponding to \( e_d \) in the \( \text{edgeToResult} \) index. The next task is to confirm if these results actually get affected by the edge deletion. For this, the polynomial is evaluated with \( e_d \) set to 0 and the other edge symbols remaining at 1. An answer vanishes if and only if all the monomials corresponding to it get evaluated to 0. In other words, the answer no longer remains valid. Since the connection points are annotated with the provenance polynomials, they get updated as well. Finally, the entry of \( e_d \) is dropped from both the indexes \( \text{edgeToResult} \) and \( \text{edgeToCP} \).

Suppose edge \( e_{14} : (\text{Ooi, hadAdvisor, Stonebraker}) \) is deleted from the example KG in Fig. 1. We evaluate the provenance polynomial of the query answer, given in Table 1, by assigning 0 to \( e_{14} \) and 1 to the rest. The resultant value would be 1 which indicates a valid derivation \( \{e_2, e_3, e_6, e_8, e_{17}\} \) of the answer exists and, hence, \( (\text{Stonebraker, Ramakrishnan}) \) is still an answer. However, if the edge \( e_2 \) is deleted then the polynomial would collapse to 0, denoting no valid derivation exists, i.e., the pair is no longer a valid answer.

6.3 Proof of Correctness

Insertion. We assume a single edge insertion and argue the correctness for the two types of queries separately. Since maintenance is done after every insertion operation, the correctness extends to multiple insertions.

Lemma 1 shows that a \( P \cdot M_{1:1} \) of a regular query will be of size exactly \( n - 1 \) and will satisfy a subquery. Hence, once a \( P \cdot M_{1:1} \) is identified correctly, and an edge insertion completes it, it must match the complete parent query. The corresponding subgraph match and its provenance polynomial is updated to reflect the current state.

A MultiMap query may have a \( P \cdot M_{1:m} \). If it has a \( P \cdot M_{1:m} \), using Lemma 3, we see that if an edge insertion satisfies the subqueries, it must also satisfy the parent query. Thus, once a \( P \cdot M_{1:m} \) is identified correctly in a MultiMap query, the correctness is ensured since the subquery results are also updated.

Deletion. If after the deletion of an edge \( e \), a provenance polynomial evaluates to 0, it implies that \( e \) was present in each corresponding derivation (monomial). Conversely, as long as a derivation exists that does not involve \( e \), its corresponding monomial will continue to evaluate to 1, and the whole polynomial will be non-zero. If \( e \) is not involved in any derivation of any answer, then it will not have any entry in \( \text{edgeToResult} \) index and there will be no impact on the query results.

7 EXPERIMENTAL EVALUATION

7.1 Setup

7.1.1 Datasets. We consider two widely used benchmark datasets. YAGO2 [23, 24]: It is an automatically built ontology gathering facts from sources like Wikipedia, GeoNames, etc. DBpedia3.9 [8]: It is the structured counterpart of Wikipedia.

Table 2 summarizes the statistics of these two datasets.

7.1.2 Query Sets. For YAGO2, we used a set of queries on which the RDF-3X was originally validated [32]. We chose 4 queries (ids A1, A2, B1, B2) that are fairly large and complex. For DBpedia, we built the query workload from the real world queries available from 2014 USEWOD (https://eprints.soton.ac.uk/379401/) query logs. This query log is a collection of over 253K actual http requests to the DBpedia SPARQL endpoint. Out of these, 12K are SPARQL SELECT queries. After filtering out nested, ordering and aggregate queries, we obtained a set of 5,490 queries. We further removed queries of size 1, those which were not parsed by a SPARQL parser, and duplicates to obtain the final workload of 215 queries.

7.1.3 KG Update Workload. For each KG, we generated insertion workloads in the following manner. Starting from the original graph, we randomly selected a pair of vertices not yet connected, and connected them with a randomly selected predicate. Note that we did not consider any predicate for insertion which is not part of at least one query, since they do not affect the results of any query and can be trivially answered by a simple lookup. For deletion workloads, we randomly deleted edges of the predicates that are involved in at least one of the queries. Using this procedure, we generated workloads of size 10000 updates each.

7.1.4 Implementation. We conducted all the experiments on a machine with 32-core 2.1 GHz CPU, 512 GB RAM, and 1 TB hard drive. Our implementation was single-threaded and in Java. We chose Neo4j 3.3.9 to store the knowledge graphs due to its capability to model property graphs. HUKA codebase is publicly available.\(^1\)

7.1.5 Baseline Systems. Wylot et al. [44] highlighted the shortcomings of named graphs to support the provenance polynomials and devised TripleProv to support provenance computation over a customized RDF-engine dipLODocus [42]. We contacted the authors for their code but unfortunately, they responded that TripleProv is no longer available. We, therefore, did a best effort implementation of TripleProv on top of property graphs supported by a leading graph database engine Neo4j and used it as a baseline. We are calling \(^1\)https://github.com/gaurgarima/HUKA

---

Table 2: Statistics of datasets

| Dataset       | Vertices | Edges | Predicates | Queries | Avg. Query Size | Subqueries |
|---------------|---------|-------|------------|---------|----------------|------------|
| YAGO2         | 8.8M    | 23M   | 78         | 4       | 6.25           | 26         |
| DBpedia       | 32M     | 117M  | 53K        | 215     | 3.90           | 879        |

---

1. https://github.com/gaurgarima/HUKA
this baseline as Neo4j. We also used two existing frameworks that enable provenance-aware query processing on PostgreSQL, a versatile middleware GProM [3] and a recently introduced PostgreSQL extension ProvSQL [36]. For the PostgreSQL based baseline systems, we modeled the KG as relational database by grouping all the triples of a particular predicate in a table, i.e., for each predicate \( P \) of the KG we construct a table \( R_P \) and populate it with the triples having predicate \( P \). We also translated SPARQL queries into SQL queries by interpreting each pair of triple sharing either subject or object with each other as a join operation on the corresponding predicate relations.

### 7.2 Analysis of Query Processing

#### 7.2.1 Subquery Generator

The subquery generator produced 26 and 879 subqueries out of 4 and 215 parent queries present in YAGO2 and DBpedia query sets respectively. Based on the results of these subqueries, connection points were identified. For YAGO2, subqueries resulted into 164K connection points whereas DBpedia queries generated 6.5M connection points.

#### 7.2.2 Global Execution Plan

The global execution tree for YAGO2 subqueries has 80 nodes while for DBpedia, it has 866 nodes. Each node in the global execution tree corresponds to an intermediate expression. To analyze the queries better, we partitioned the query set of DBpedia based on the query size. In Table 3, the query set \( Q_S_i \) is the set of DBpedia queries of size \( i \). The entire query set corresponds to the full query set.

To confirm our claim that subqueries are likely to overlap and, thus, the global plan promotes re-usability, we compare the number of intermediate expressions in the global plan to the total number of intermediate expressions required to compute the final expression of all the subqueries. To compute the latter, we sum up the sizes of all the subqueries as the size, i.e., number of triple patterns, gives the number of joins required to compute it which, in turn, correspond to the number of intermediate expressions. We have reported the total number of intermediate expressions in the absence of a global plan in Table 3. The global plan has cut down the intermediate expression computation considerably. The total number of intermediate expressions is reduced by close to 70%. (The column “coverage” will be discussed in Sec. 7.3.3.)

#### 7.2.3 Space Usage

The edgeToResult inverted index sizes for YAGO2 and DBpedia are 43MB and 322MB respectively while the edgeToCP index sizes are 41MB and 995MB respectively.

#### 7.2.4 Time Consumption

We took an average time of 6.9 minutes and 10 minutes to register DBpedia and YAGO2 queries respectively. Majority of the time while registering a query is spent on execution plan generation as we exhaustively generate all possible derivations of a subquery. Larger the query size, more is the number of derivations. Since YAGO2 queries are fairly large with an average size of 6.25 triples per query, whereas DBpedia queries size has an average of 3.90 triples per query, it requires more time to register YAGO2 queries than DBpedia.

It needs to be noted that queries are registered only once, though. The average time spent per update operation subsequently is much less as reported next.

### 7.3 Analysis of Update Processing

#### 7.3.1 Baseline Comparison

The running times (averaged over 5 runs) per update operation taken by our system HUKA and the various baselines on the two datasets are reported in Table 4. For HUKA, we recorded the average total time taken to update the parent queries and maintain the supporting structure. As none of the baseline systems support incremental update of query provenance, we computed the average total time taken to execute all the relevant queries for a particular update request. The relevant queries are those in which the updated edge predicate is present.

On YAGO2, HUKA takes less than a second to update the query results and their provenance polynomials, whereas for DBpedia, the time taken is slightly more. Importantly, our approach is around 4 to 48 times faster than the nearest baselines.

Since the query size directly translates to number of joins, GProM and ProvSQL require lesser time for DBpedia. However, for HUKA, the number of affected queries for DBpedia are generally higher as the total number of queries (4 in YAGO2 versus 215 in DBpedia) are higher. This overshadows the impact of lesser number of joins per query and, consequently, HUKA requires more time for DBpedia than YAGO2. Unlike relational data, Neo4j uses graph data organization and stores logically related data nearby. This results in efficient graph traversal and, therefore, Neo4j requires considerably lesser time for YAGO2 since it is mostly impacted by the number of queries.

#### 7.3.2 Varying Workloads

We next study in detail the effect of different types of updates: deletion versus insertion. We generated multiple workloads (as described in Sec. 7.1) of size 10000 updates each with controlled ratio of operations. We used 5 configurations of deletion to insertion ratios: Insertion-Heavy (1:9), Insertion-Moderate (3:7), Balanced (5:5), Deletion-Moderate (7:3), and Deletion-Heavy (9:1). The average times are reported in Table 5. For both the datasets, the average total time gradually increases as we...
move from *deletion-heavy* to *insertion-heavy* workload. The trend is expected as insertion is computationally heavier than deletion.

7.3.3 Closer Look. Next, to get a better insight of our system, we decided to have a closer look at its different steps. We chose the larger query set DBpedia and the query set partitions outlined in Table 3. We treat each query set $QS_i$ individually, i.e., built separate query execution plans, annotated separate KGs, and constructed separate indexes for each query set.

We divide the total running time into 2 components: (a) *response time*, defined as the time taken to update the parent query result(s), and (b) *maintenance time*, defined as the time taken to maintain the data structures so as to ensure correctness of subsequent updates. We can update result of regular parent queries just by examining the coverage of a global plan as the average number of intermediate expressions to compute, higher the number of intermediate expressions to compute, higher is the subquery result updation time. To quantify this, we defined Why-Not Provenance with NedExplain. In EDBT.

In this paper, we have proposed a system to efficiently maintain provenance of results of standing queries in knowledge graphs that are dynamic in nature. We used decomposition of queries into subqueries and inverted indexes to quickly update the answers and their provenance polynomials. Experiments show the effectiveness in large real-life knowledge graphs.

In future, we would like to work with more generic SPARQL queries that include aggregates, nested queries, ordering, etc.

REFERENCES

[1] Parag Agrawal et al. 2006. Trie: A system for data, uncertainty, and lineage. *VLDB* (2006).
[2] Bogdan Alexe, Laura Chaticaria, and Wang-Chiew Tan. 2006. Spider: a schema mapping debugger. In *VLDB*.
[3] Bahareh Salat Arab et al. 2018. GProM-a swiss army knife for your provenance needs. *Qte. Rel. Comp. Soc. IEEE Tech. Comm. Data Engg.* 41, 1 (2018).
[4] Marcelo Arenas, Sebastián Conca, and Jorge Pérez. 2012. Counting beyond a Yottabyte, or How SPARQL 1.1 Property Paths Will Prevent Adoption of the Standard. In *WWW*.
[5] Arygyo Avgoustaki et al. 2016. Provenance Management for Evolving RDF Datasets. In *ESWC*.
[6] Shivnath Babu. 2009. Continuous Query. *Springer US*, 492–493.
[7] Nicole Bidot, Melanie Herschel, and Katerina Tsompanaki. 2014. Query-Based Why-Not Provenance with NedExplain. In *EDBT*.
[8] Christian Bizer et al. 2009. DBpedia - A crystallization point for the Web of Data. *J. Web Semant.* 7, 3 (2009), 154–165.
[9] Angela Bomfim, Wim Martens, and Thomas Timm. 2019. Navigating the Maze of Wikidata Query Logs. In *WWW*.
[10] Peter Buneman, Sanjeev Khanna, and Tan Wang-Chiew. 2001. Why and where: A characterization of data provenance. In *ICDT*.
[11] James Cheney, Laura Chaticaria, and Wang Chiew Tan. 2009. Provenance in Databases: Why, How, and Where. *Foundations and Trends in Databases* 1 (2009), 379–474.
[12] Sutanay Choudhury et al. 2015. A Selectivity based approach to Continuous Pattern Detection in Streaming Graphs. In *EDBT*.
[13] Yingwei Cui and Jennifer Widom. 2000. Practical lineage tracing in data warehouses. In *ICDE*.
[14] Richard Cyganiak. 2005. A relational algebra for SPARQL. *Technical Report* HPL-2005-170. HP Laboratories.
[15] Daniel Deutsch and Amir Gilad. 2019. Reverse-Engineering Conjunctive Queries from Provenance Examples. In *EDBT*.
[16] Wenlei Fan et al. 2011. Incremental Graph Pattern Matching. In *SIGMOD*.
[17] Garima Gaur, Srikanta J. Bedathur, and Arnab Bhattacharya. 2017. Tracking the Impact of Fact Deletions on Knowledge Graph Queries Using Provenance Polynomials. In *CIKM*.
[18] Boris Glavic and Gustavo Alonso. 2009. The Perm Provenance Management System in Action. In *SIGMOD*.
[19] Todd J. Green, Grigoris Karvounarakis, and Val Tannen. 2007. Provenance Semirings. In *PODS*.
[20] Andrey Gubichev and Thomas Neumann. 2014. Exploiting the query structure for efficient join ordering in spارql queries. In *EDBT* 439–450.
[21] Harry Halpin and James Cheney. 2014. Dynamic Provenance for SPARQL Updates. In *ESWC*.
[22] James Hendler et al. 2012. US government linked open data: semantic data gov. *IEEE Intelligent Systems* 27, 3 (2012), 25–31.
[23] Johannes Hoffart et al. 2011. YAGO2: exploring and querying world knowledge in time, space, context, and many languages. In *WWW*.
[24] Johannes Hoffart et al. 2013. YAGO2: A spatially and temporally enhanced knowledge base from Wikipedia. *Artificial Intelligence* (2013).
[25] Zachary G Ives et al. 2008. The ORCHESTRA collaborative data sharing system. *ACM SIGMOD Record* 37, 3 (2008).
[26] Chathura Kankanamge et al. 2017. Graphiflow: An Active Graph Database. In *SIGMOD*.
[27] Kyoungmin Kim et al. 2018. TurboFlux: A Fast Continuous Subgraph Matching System in Action. In *SIGMOD*.
[28] Marinos Meimaris et al. 2017. Extended characteristic sets: graph indexing for sparql query optimization. In *ICDE*.
[29] Paolo Missier, Khalid Belhajjame, and James Cheney. 2013. The W3C PROV Family of Specifications for Modelling Provenance Metadata. In *ICDT*.
[30] Hoshi Mistry et al. 2001. Materialized View Selection and Maintenance Using Multi-query Optimization. In *SIGMOD*. 12.
[31] Tom M. Mitchell et al. 2015. Never-Ending Learning. In *AAAI*.
[32] Thomas Neumann and Gerhard Weikum. 2010. The RDF-3X engine for scalable management of RDF data. *The VLDB Journal* (2010).
[33] Mary Roth and Wang Chiew Tan. 2013. Data Integration and Data Exchange: It’s Really About Time. In *CIDR*.
[34] A. D. Sarma, M. Theobald, and J. Widom. 2008. Exploiting Lineage for Confidence Computation in Uncertain and Probabilistic Databases. In *ICDE*.
[35] Pierre Senellart. 2018. Provenance and probabilities in relational databases. ACM SIGMOD Record 46, 4 (2018), 5–15.
[36] Pierre Senellart et al. 2018. ProSQL: provenance and probability management in PostgreSQL. PVLDB 11, 12 (2018), 2034–2037.
[37] Nicholas E. Taylor and Zachary G. Ives. 2006. Reconciling While Tolerating Disagreement in Collaborative Data Sharing. In SIGMOD, 12.
[38] Prinja Upadhyay et al. 2020. Aspect-Based Academic Search Using Domain-Specific KB. In ECIR.
[39] Quan Wang et al. 2017. Knowledge graph embedding: A survey of approaches and applications. TKDE 29, 12 (2017), 2724–2743.
[40] R. B. White et al. 2014. Toward Enhanced Pharmacovigilance Using Patient-Generated Data on the Internet. Clinical Pharmacology and Therapeutics 96 (2014), 239 – 246.
[41] David S. Wishart et al. 2008. DrugBank: a knowledgebase for drugs, drug actions and drug targets. In Nucleic Acids Research, Vol. 36.
[42] Marcin Wytot et al. 2011: diplODocu[RDF] - Short and Long-Tail RDF Analytics for Massive Webs of Data. In ISWC.
[43] M. Wytot et al. 2017. Storing, Tracking, and Querying Provenance in Linked Data. IEEE TKDE 29, 8 (2017).
[44] Marcin Wylot, Philippe Cudre-Mauroux, and Paul Groth. 2014. TripleProv: Efficient Processing of Lineage Queries in a Native RDF Store. In WWW. 455–466.
[45] Shuo Yang et al. 2017. Efficiently Answering Technical Questions - A Knowledge Graph Approach. In AAAI.