Modern Techniques for Querying Graph-Structured Relations: Foundations, System Implementations, and Open Challenges

Amine Mhedhbi
University of Waterloo
amine.mhedhbi@uwaterloo.ca

Semih Salihoğlu
University of Waterloo
semih.salihoglu@uwaterloo.ca

ABSTRACT

The last decade has seen an emergence of numerous specialized graph DBMSs (GDBMSs) as well as graph-optimized extensions of RDBMSs. In addition, several query processing techniques, such as worst-case optimal join algorithms and factorized query processing, have been introduced in the context of RDBMSs, which find their best applications on graph workloads. In this tutorial, we review the recent advances in query processing techniques for graph workloads. For each technique, we first overview the theoretical foundations. Then, we overview how DBMSs implement these techniques. Finally, we discuss the open challenges for existing implementation approaches.

PVLDB Reference Format:
Amine Mhedhbi and Semih Salihoğlu. Modern Techniques for Querying Graph-Structured Relations. PVLDB, 15(12): 3762 - 3765, 2022.
doi:10.14778/3554821.3554894

1 INTRODUCTION

Querying graph-structured data is integral to a wide range of analytical applications such as recommendations in social networks, fraud detection in financial transaction networks, and inference over knowledge bases [21]. There are two primary defining features of graph workloads: (i) prevalence of many-to-many (m-n) relations across entities; and (ii) prevalence of complex join-heavy queries over these relations. The joins in these queries can have several different structures: (i) cyclic, such as when finding cliques of phone calls; (ii) acyclic, such as when finding long chains of financial transactions; or (iii) recursive, such as when finding shortest connections between users in social networks. This contrasts with traditional relational workloads, such as those found in the popular TPC benchmarks, that contain many primary-foreign (PK-FK) key joins. The combination of complex join structures in these queries and the m-n cardinality of relations in these datasets pose serious challenges for traditional query processors. For example, queries can generate large intermediate relations (IRs), which often cannot be handled by traditional techniques.

The last decade has seen an emergence of numerous prototype and commercial DBMSs that are optimized for graph workloads. These include specialized systems such as GDBMSs that adopt the property graph data model e.g., Neo4j, TigerGraph, Avantgraph, GraphflowDB [11], earlier RDF systems e.g., RDF-3x [18], and graph-optimized extensions of RDBMSs, e.g., GR-Fusion [8], GRainDB [10], and GQ-Fast [13]. The query processors of these systems contain specialized techniques, such as *pointer-based joins* that rely on dense system-level integer IDs, *worst-case optimal join (WCOJ) algorithms* or *factorized query processing* that limit IR sizes.

This tutorial covers 4 topics: (i) pointer-based joins and core binary joins; (ii) WCOJ algorithms; (iii) factorization; and (iv) recursive join query evaluation. We overview: (i) the foundations of these techniques when appropriate; (ii) the current design choices different DBMSs have made to integrate these techniques; and (iii) the challenges for existing implementation approaches. Our goal is to bring a structure to this vast theory and system-oriented literature. For reference, Table 1 shows the query processors of the systems that we cover and the implementation design choices.

2 ORGANIZATIONAL INFORMATION

- **Duration:** This 3 hour tutorial covers: (1) Graph workloads overview: 15 minutes; (2) Pointer-based joins (Section 3): 45 minutes; (3) WCOJ algorithms (Section 4): 45 minutes; (4) Factorized query processing (Section 5): 45 minutes; and (5) Techniques for recursive queries (Section 6): 30 minutes.
- **Intended Audience and Prerequisites:** The tutorial is intended for general database researchers and Ph.D. students, especially systems-oriented ones, and DBMS developers. We do not require any prior knowledge and will provide any necessary background.
- **Prior Tutorials:** We have not given a prior tutorial on the topics we are proposing here.

We begin the tutorial by providing a working definition of graph workloads and examples queries. Remaining sections cover (2)-(4).

3 POINTER- VS VALUE-BASED JOINS

We next give a brief historical overview of DBMSs that adopt graph-based data models, discussing the earliest IDS system, RDF systems, and modern GDBMSs which adopt the property graph model. We then cover our first main topic of *pointer-based joins* in GDBMSs which perform joins between node records along predefined edge records.GDBMSs use system-level dense integer IDs of nodes, which serve as pointers to look up neighbours. This contrasts with and can be more efficient than *value-based joins* on arbitrary attributes in RDBMSs.

3.1 System Implementations

We will next discuss pointer-based join implementations.
Native GDBMSs: There are three components to implementing pointer-based joins in GDBMSs:

- System-level dense integer node IDs: node records are given consecutive IDs starting from 0 to $|V|$ for an input graph $G(V, E)$.

This work is licensed under the Creative Commons BY-NC-ND 4.0 International License. Visit https://creativecommons.org/licenses/by-nc-nd/4.0/ to view a copy of this license. For any use beyond those covered by this license, obtain permission by emailing info@vldb.org. Copyright is held by the owner/author(s). Publication rights licensed to the VLDB Endowment.
Proceedings of the VLDB Endowment, Vol. 15, No. 12 ISSN 2150-8097.
doi:10.14778/3554821.3554894

Table 1: DBMSs and technique implementations. X indicates absence of feature. N/A indicates: "feature is not specified". INLJ, HJ, and MJ refer to Index Nested Loop Join, Hash Join, and Merge Join, respectively.

DBMS	Join Type	Core Join Alg	WCOJ Algo	Data Representation Scheme	Recursive Joins
Umbra [17]	Value-based	HJ	Hash-based	Flat	N/A
GrainDB [10]	Value- and Pointer-based	HJ	Hash-based	Flat	α-RA
EmptyHeaded [1]	Value-based	INLJ	Sorted indexes	Flat	X
GQ-Fast [13]	Value- and Pointer-based	INLJ	X	Flat	X
GR-Fusion [8]	Value- and Pointer-based	INLJ	X	Flat	α-RA
GraphflowDB [11]	Pointer-based	HJ & INLJ	Sorted indexes	F-representations (restricted)	X
AvantGraph	Pointer-based	N/A	Sorted indexes	N/A	WaveGuide
FDB [5]	Value-based	INLJ	Sorted indexes	F-representations	X
Neo4j	Pointer-based	HJ & INLJ	X	Flat	X
RDF-3X [18]	Value-based	MJ	X	Flat	X

- Adjacency list indexes: the edge records are indexed using the system-level node IDs providing constant-time access to all incoming/outgoing "neighbours" of a node.
- Index Nested Loop Joins (`INLJ`): The operator would use adjacency list indexes e.g., `Expand` operator in Neo4j.

Another adopted approach is that of the RDF-3x system which indexes triples and hence IDs in B+ trees and uses merge join (`MJ`).

GR-Fusion [8] and GQ-Fast [13]: GR-Fusion and GQ-Fast have proposed extending RDBMSs with pointer-based joins following the GDBMS approach by defining *graph views* which indicate the relations that correspond to nodes and edges. These views are indexed in adjacency list indexes using row identifiers (RIDs).

GRainDB [10]: An extension of DuckDB with pointer-based joins that proposes a more general approach to speed up `PK-FK` joins. The components in this implementation are:

- RID Materialization: Similar to prior approaches, GRainDB uses system-level RIDs. If users specify an equality join from a table F to P such that the join columns of F are foreign keys to P, then the system adds a `RID` column to F that contains for each row $r_f \in F$, the RID of the row $r_p \in P$ to which r_f has the foreign key.
- Sideways Information Passing (SIP): GRainDB uses the hash join (`HJ`) operator of DuckDB but modifies it to perform SIP to evaluate the join of F and P using RIDs.
- RID Indexes: Indexing the RID values in F in adjacency list indexes allows the system to pass information from P to F.

In general, `INLJ` and `HJ` provides different benefits: `HJ` is useful when predicates on the relationship table are selective, while `INLJ` is useful when predicates on source entities are selective.

3.2 Open Challenges

We will discuss two open challenges: (i) `INLJ`, `HJ`, and `MJ` are the default operators of different systems and a comprehensive understanding of which operator to use under which settings is needed. (ii) *Cost-based optimization:* In every RDBMS integration, a portion of the query is identified and planned separately to use pointer-based joins in a rule-based manner. A holistic cost-based approach has so-far not been described in the literature.

4 WORST-CASE OPTIMAL JOINS

Irrespective of the core join operator systems use and whether joins are pointer or value based, the predominant join plans of existing GDBMSs and RDBMSs are *binary join (BJ) plans.* These are plans whose join operators join two base or intermediate relations at a time until all relations are joined. The next part of our tutorial discusses an important shortcoming of BJ plans for cyclic queries when joins are over m-n relations.

4.1 Foundations

Let $Q_\triangle$ be the triangle self-join query, where `F` is the m-n `Follows(from, to)` relation in a social network: $Q_\triangle$:= `F(`u_1`,`u_2`)` $\bowtie$ `F(`u_2`,`u_3`)` $\bowtie$ `F(`u_3`,`u_4`)`. In graph terms, BJ plans correspond to evaluating the joins one query edge at a time. Such a plan first finds open triangles on `Follows` edges and then closes the triangle. When joins are over m-n relations, intermediate relations can be very large. This was made formal in the seminal paper of Atserias et al. [4], that put a tight bound, called the AGM bound, on the worst-case size of join queries when only the sizes of the input relations are known. The AGM bound of $Q_\triangle$ over a relation with N tuples is $N^{1.5}$. However there are input datasets where any binary join plan generates $\Omega(N^2)$ intermediate results for $Q_\triangle$.

This asymptotic gap was fixed in the new WCOJ algorithms [19]. Instead of joining Q table(s) at a time using binary joins, WCOJ algorithms join Q *an attribute at a time* using multiway join operators. The attributes $\mathcal{A} = \{a_1, ..., a_n\}$ of Q are given an ordering e.g., $(a_1, a_2, ..., a_n)$. Then, at step i, $(a_1, ..., a_i)$ prefixes are extended to the a_{i+1} attribute by obtaining a_{i+1} sets from each relation R_j that contains a_{i+1} and any of the $\{a_1, ..., a_i\}$ attributes and intersecting these sets. On $Q_\triangle$, a WCOJ algorithm would extend all edges to triangles without ever computing open triangles.

We will cover the shortcomings of BJs, the AGM bound, and the WCOJ algorithms.

4.2 System Implementations

We will next cover the two broad approaches to integrate WCOJ algorithms into DBMSs.

4.2.1 Sorted Index Approach. Most implementations use precomputed sorted indexes each of which sorts relations on different attribute permutations. The first documented WCOJ algorithm is of the Leapfrog Trie-Join (LFTJ) [22] in LogicBlox [3]. EmptyHeaded [1] (EH) is a prototype RDBMS and adopts a similar approach. In contrast to LFTJ, EH also uses BJs in addition to WCOJ algorithms. Specifically, EH optimizes queries using *generalized hypertree decomposition (GHD).* Two important limitations of EH's approach is: (i) it does not optimize the ordering of the attributes using Generic Join (GJ); and (ii) the computation is broken down into two phases where multiway joins happen before BJs.

Addressing these shortcomings of EH was the main objective of recent research [15, 16]. This work described a GDBMS integration of WCOJ algorithms in GraphflowDB. The approach of GraphflowDB, which also uses sorted indexes, has two primary differences compared to EH: (i) GraphflowDB plans seamlessly mixes binary join and WCOJ-style intersection based operators; and (ii) the system optimizes the choice of picks attribute orderings using a new cost metric called *intersection cost*, and can adaptively pick the attribute orderings during query evaluation.

4.2.2 Hash-trie Index Approach of Umbra [17]: This approach stems from the fact that maintaining sorted indices is very expensive and is not update friendly. Umbra computes the trie indexes required by its WCOJ algorithms *on the fly* when a query is issued. These indices are built as nested hash tables, where each level corresponds to exactly one join key attribute and the leaf nodes are sorted using a linked list structure. The system has a new `HashTrieJoin` operator that takes in hash-trie indexes of $k \geq 2$ relations $R_1, ..., R_k$, and joins them using GJ-style algorithm. The comparisons during GJ evaluation are based on hash values and not actual key values.

4.3 Open Challenges

We will discuss two open challenges: (i) the current two integration approaches suggest potential benefit from a hybrid *database cracking style* [9] of continuous physical reorganization or building of sorted indexes on demand as queries arrive; and (ii) the theory community has advanced the theory of WCOJ algorithms to *beyond WCOJs* [12] that have stronger optimality guarantees. These algorithms intersect *gaps in the trie indices* instead of actual tuples. How to integrate them into DBMSs is a promising research direction.

5 FACTORIZED QUERY PROCESSING

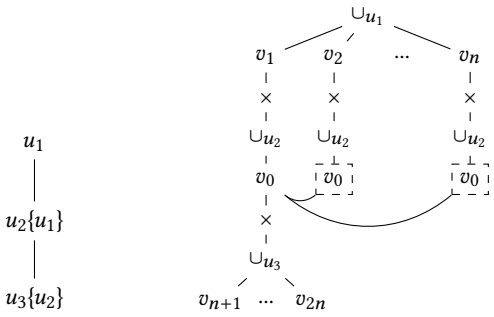
We next discuss factorized representations of relations [20]. While WCOJ algorithms reduce intermediate relation sizes for cyclic queries, factorization reduces IR sizes primarily for acyclic queries.

5.1 Foundations

Traditional query processors use *flat representations* of relations both for storage and query processing. When evaluating join queries over m-n relations, this may lead to redundancy.

We present factorization through examples. Let Q_{2F} be a 2-hop query: $\texttt{F(u}_1\texttt{,u}_2\texttt{)} \bowtie \texttt{F(u}_2\texttt{,u}_3\texttt{)}$ where `F` is the m-n `Follows` relation in a social network. We further evaluate the query on an input graph where $\{v_1, ..., v_n\}$ have outgoing edges to v_0, which has outgoing edges to $\{v_{n+1}, ..., v_{2n}\}$. The output relation of Q_{2F} as a flat representation $\texttt{OUT}_{2F} = \cup\texttt{t(u}_1\texttt{:v}_i\texttt{,u}_2\texttt{:v}_0\texttt{,u}_3\texttt{:v}_j\texttt{)}$ contains n^2 tuples and $3n^2$ atomic values, albeit with a lot of repetition. A more succinct representation is: $\texttt{OUT}_{2F} = \{v_1, ..., v_n\} \times \{v_0\} \times \{v_{n+1}, ..., v_{2n}\}$, which contains $O(n)$ atomic values.

Over the last decade, the *theory of factorized databases* [20] has laid the foundation for avoiding such repetition. Under this theory, each flat tuple is represented by a Cartesian product of singleton relations, which are unary relations with one tuple. This theory introduced further two factorized representation schemes: i) f-representations, which are unions of Cartesian products of sets that contain singleton values; ii) d-representations, which are generalized f-representations by defining and reusing repeated



(a) T_2. **(b) D-Rep.** Q_{2F} **per** D-T_2.

Figure 1: D-representations for Q_{2F} over D-tree $\mathcal{T}$.

subexpressions. F- and d-representations are denoted by *f-* and *d-trees*, respectively which describe the factorization structure of a relation at the attribute level. Figure 1 shows an example of a d-representation of $\texttt{OUT}_{2F}$ over d-tree $\mathcal{T}$.

The theory of factorized databases extends the theory of worst-case optimal join query sizes and has established that for any query Q the following holds: $s^{\uparrow}(Q) \leq s(Q) \leq AGM(Q)$. Here, $s^{\uparrow}(Q)$, $s(Q)$, and $AGM(Q)$ are the worst-case size of the d-, f- and flat representations of a query, respectively [20].

5.2 System Implementations

We will next discuss the existing approaches for factorization.

FDB [5]: was the first processor to introduce factorized query processing. This approach directly processes F-representations stored in tries i.e., operators take in and output F-representations. Each plan is linear and operators rely on full materialization of intermediate relations. An FDB-style processor requires novel restructuring operators such as `Swap` that manipulate f-representations.

LBQP [7]: is a pipelined query processor of GraphflowDB that extends block-based processors of columnar RDBMSs. LBQP is designed for in-memory GDBMSs and uses a restricted set of f-trees. LBQP operators exist in traditional processors with the goal of easy integration. We will give an overview of the design of LBQP and discuss the pros and cons of this approach with that of FDB.

5.3 Open Challenges

Compared to WCOJ algorithms, fewer systems work has been done in factorization and this field contains numerous open challenges. Two important ones are: (i) No prior work has proposed a complete pipelined query processor architecture; and (2) No prior work has integrated d-representations, i.e., named expressions and their reuse, in limited or general form into actual systems.

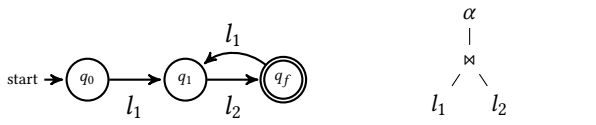
6 RECURSIVE JOINS

Queries in graph workloads can be augmented with fragments containing recursive joins such as *Shortest Path Queries* (SPQs) and *Regular Path Queries* (RPQs). Several systems natively support SPQs however these approaches are not conducive to optimizations. We will briefly mention these but instead focus on RPQs.

6.1 Foundations

An RPQ finds $(\texttt{v}_i, \texttt{v}_j)$ pairs connected through path(s) that adhere to a regular expression. RPQs can be defined semantically along

two axes: 1) counting the number of paths or just checking for existence; and 2) *simple* i.e., have a restriction such that no vertex in the path is visited twice, or *arbitrary* with no such restriction. Different semantics land themselves best for different optimizations. Nevertheless, existing approaches can be split into two broad categories: 1) α–RA which is an extension of relational algebra by adding the α operator to evaluate the *transitive closure* [2]; and 2) Finite Automaton (FA) plans [23] which started with the seminal work of Mendelzon and Wood [14].



(a) FA Plan. **(b) α plan.**

Figure 2: The parse tree and associated plan in α–RA.

We will demonstrate the difference between the two approaches with examples. Consider an RPQ that finds the pairs whose path matches the expression $(l_1 l_2)$+, where l_i is an edge label. Figure 2 shows the FA and an α-RA plans for this query. The FA plan evaluates the query one label at a time by matching the tuple values to the states as described by the FA shown in Figure 2a. Instead, an α-RA approach computes an intermediate result, which is given to an "α operator" that runs self-joins until a fixed point is reached.

In the tutorial, we will overview both approaches and optimizations and show the differences between the plan spaces.

6.2 System Implementations

We will next discuss approaches to evaluate RPQs: (i) General α–RA approach; (ii) WaveGuide which mixes both α–RA and FA plans [23].

α–RA: RPQs have generally been studied in the context of RDF systems. The α–RA approach [2] with an α operator enables RDF stores to evaluate the property paths in SPARQL. This approach is easy to integrate in RDBMSs that support recursive SQL queries as done in the prototype system on top of Apache Spark [6] for recursive components of SQL [6].

WaveGuide [23] is a hybrid approach that combines FA plans and the α–RA approach leading to a richer plan space for which a new cost optimization approach is needed [23]. The system can compute intermediate relations which are joined relations as indicated by the regular expression. The intermediate relations can later be used by an extended FA that treats them as a single label in the regular expression. For example for $(l_1 l_2)$+, WaveGuide can compute the join between l_1 and l_2, store the result and then use that as a new label over which an FA is defined. WaveGuide plans are also expressive enough to represent any α-RA based plan. A WaveGuide plan contains one or more wavefronts. These wavefronts are expanded repeatedly until no new answers are found.

6.3 Open Challenges

We will discuss two open challenges for evaluating recursive queries: (1) Waveguide-like plans can lead to a large plan space, which are challenging for existing DBMS query optimizers to explore and assign costs to; and (2) Existing approaches to evaluate recursive queries use flat tuple representations. Proposing recursive query processors that also factorize the intermediate results generated by their plans is a promising research venue.

7 AUTHOR INFORMATION

Amine Mhedhbi is a Ph.D. student at the University of Waterloo. His research focuses on scaling query processing over graph-structured relations. He received the VLDB Best Paper Award in 2018 and the Microsoft Ph.D. Research Fellowship 2020-2022.

Semih Salihoğlu is an Assistant Professor at the University of Waterloo. His work focuses on algorithms and systems for managing, processing, visualizing, and debugging large-scale graph-structured data. He has co-designed and co-implemented several graph processing systems, such as GraphflowDB [11] (along with Mhedhbi), GRainDB [10], GraphSurge, and GPS. He received the VLDB Best Paper Award in 2018 and Distinguished Reviewer or PC Member Award in PVLDB 2018 and 2020 and SIGMOD 2018 and 2021. He has received his PhD from Stanford University in 2015.

REFERENCES

[1] Christopher R. Aberger, Susan Tu, Kunle Olukotun, and Christopher Ré. 2016. EmptyHeaded: A Relational Engine for Graph Processing. In *SIGMOD*. 431–446.
[2] Rakesh Agrawal. 1987. Alpha: An Extension of Relational Algebra to Express a Class of Recursive Queries. In *ICDE*. 580–590.
[3] Molham Aref, Balder ten Cate, Todd J. Green, Benny Kimelfeld, Dan Olteanu, Emir Pasalic, Todd L. Veldhuizen, and Geoffrey Washburn. 2015. Design and Implementation of the LogicBlox System. In *SIGMOD*. 1371–1382.
[4] Albert Atserias, Martin Grohe, and Dániel Marx. 2017. Size bounds and query plans for relational joins. In *CoRR*, Vol. abs/1711.03860.
[5] Nurzhan Bakibayev, Dan Olteanu, and Jakub Závodný. 2012. FDB: A Query Engine for Factorised Relational Databases. In *PVLDB*. 1232–1243.
[6] Sarah Chlyah, Pierre Genevès, and Nabil Layaïda. 2021. Distributed Evaluation of Graph Queries using Recursive Relational Algebra. In *CoRR*, Vol. abs/2111.12487.
[7] Pranjal Gupta, Amine Mhedhbi, and Semih Salihoglu. 2021. Columnar Storage and List-based Processing for Graph Database Management Systems. In *PVLDB*. 2491–2504.
[8] Mohamed S. Hassan, Tatiana Kuznetsova, Hyun Chai Jeong, Walid G. Aref, and Mohammad Sadoghi. 2018. GRFusion: Graphs as First-Class Citizens in Main-Memory Relational Database Systems. In *SIGMOD*. 1789–1792.
[9] Stratos Idreos, Martin L. Kersten, and Stefan Manegold. 2007. Database Cracking. In *CIDR*. 68–78.
[10] Guodong Jin and Semih Salihoglu. 2022. Making RDBMSs Efficient on Graph Workloads Through Predefined Joins. In *PVLDB*. 1011–1023.
[11] Chathura Kankanamge, Siddhartha Sahu, Amine Mhedhbi, Jeremy Chen, and Semih Salihoglu. 2017. Graphflow: An Active Graph Database. In *SIGMOD*. 1695–1698.
[12] Mahmoud Abo Khamis, Hung Q. Ngo, Christopher Ré, and Atri Rudra. 2016. Joins via Geometric Resolutions: Worst Case and Beyond. In *TODS*.
[13] Chunbin Lin, Benjamin Mandel, Yannis Papakonstantinou, and Matthias Springer. 2016. Fast In-Memory SQL Analytics on Typed Graphs. In *PVLDB*. 265–276.
[14] Alberto O. Mendelzon and Peter T. Wood. 1995. Finding Regular Simple Paths in Graph Databases. In *SIAM J. Comput.* 1235–1258.
[15] Amine Mhedhbi, Chathura Kankanamge, and Semih Salihoglu. 2021. Optimizing One-time and Continuous Subgraph Queries using Worst-case Optimal Joins. In *TODS*. 6:1–6:45.
[16] Amine Mhedhbi and Semih Salihoglu. 2019. Optimizing Subgraph Queries by Combining Binary and Worst-Case Optimal Joins. In *PVLDB*. 1692–1704.
[17] Thomas Neumann and Michael J. Freitag. 2020. Umbra: A Disk-Based System with In-Memory Performance. In *CIDR*.
[18] Thomas Neumann and Gerhard Weikum. 2010. The RDF-3X engine for scalable management of RDF data. In *VLDBJ*. 91–113.
[19] Hung Q. Ngo, Christopher Ré, and Atri Rudra. 2013. Skew strikes back: new developments in the theory of join algorithms. In *SIGMOD Rec.* 5–16.
[20] Dan Olteanu and Jakub Závodný. 2015. Size Bounds for Factorised Representations of Query Results. In *TODS*. 2:1–2:44.
[21] Siddhartha Sahu, Amine Mhedhbi, Semih Salihoglu, Jimmy Lin, and M. Tamer Özsu. 2020. The ubiquity of large graphs and surprising challenges of graph processing: extended survey. In *VLDBJ*. 595–618.
[22] Todd L. Veldhuizen. 2014. Triejoin: A Simple, Worst-Case Optimal Join Algorithm. In *ICDT*. 96–106.
[23] Nikolay Yakovets, Parke Godfrey, and Jarek Gryz. 2016. Query Planning for Evaluating SPARQL Property Paths. In *SIGMOD*. 1875–1889.