Evaluation of SPARQL Property Paths via Recursive SQL

Nikolay Yakovets, Parke Godfrey, and Jarek Gryz

Department of Computer Science and Engineering, York University, Canada
{hush,godfrey,jarek}@cse.yorku.ca

Abstract. Property paths, a part of the proposed SPARQL 1.1 standard, allow for non-trivial navigation in RDF graphs. We investigate the evaluation of SPARQL queries with property paths in a relational RDF store. We propose a translation of SPARQL property paths into recursive SQL and discuss possible optimization strategies.

1 Introduction

The Semantic Web aims to augment the information stored on the Web today with structured, machine-readable, graph-like metadata via the Resource Description Framework (RDF). Under RDF, *resources* are uniquely identified by Internationalized Resource Identifiers (IRIs), and *statements* describe relationships between resources in the form of triples (*subject, property, object*). Properties name the relationships; e.g., "created by" and "born in". Thus, RDF triples define a directed, edge-labeled graph.

SPARQL Protocol and RDF Query Language (SPARQL) [1] is a query language for RDF. A SPARQL query consists of a set of *variables* and a *graph pattern* used for matching within the RDF graph. In the SPARQL 1.0 standard, graph patterns only allow simple navigation in the RDF graph, by matching nodes over fixed-length paths. Under the proposed SPARQL 1.1 standard, the W3C working group has greatly extended the navigational capabilities of SPARQL queries by allowing graph patterns that include regular expressions in the form of *property paths*, which enable matching of nodes over arbitrary-length paths, and which allow a limited form of negation.

We study the evaluation of property paths over a relational RDF store. Since RDF data is stored in a relational database for us, a property path needs to be translated into an SQL query. We define a translation strategy which, given a SPARQL property path, generates an equivalent recursive SQL query. We outline our vision of how the resulting SQL can be optimized. We believe this is the first study of the evaluation of SPARQL property paths in the relational setting.

2 Preliminaries

SPARQL Property Paths. We use the terminology from [2]. Consider the following pairwise disjoint, infinite sets: I (IRIs), B (blank nodes), L (literals)

and V (variables). The proposed SPARQL 1.1 standard [1] defines a property path recursively as follows: (1) any $a \in I$; (2) given property paths p_1 and p_2, p_1/p_2, $p_1|p_2$, $\hat{}p_1$, p_1*, p_1+ and $p_1?$; and (3) given $a_1, \ldots, a_n \in I$, then $!a_1$, $!\hat{}a_1$, $!(a_1|\ldots|a_n)$, $!(\hat{}a_1|\ldots|\hat{}a_n)$ and $!(a_1|\ldots|a_j|\hat{}a_{j+1}|\ldots|\hat{}a_n)$. Hence, property paths are regular expressions over vocabulary I of all IRIs, for which "/" is concatenation, "|" disjunction, "ˆ" inversion, "$*$" Kleene star (zero or more occurrences), "+" one or more occurrences, and "?" zero or one occurrence. Negated property paths are not supported, but negation on IRIs, inverted IRIs and disjunctions of combinations of IRIs and inverted IRIs is allowed. A property-path triple is a tuple t of the form (u, p, v), where $u, v \in (I \cup V)$ and p is a property path. Such a triple is a graph pattern that matches all pairs of nodes $\langle u, v \rangle$ in an RDF graph that are connected by paths that conform to p.

Relational RDF Stores. A wide variety of RDF stores that rely on a relational back-end have been proposed. (See [3] for an overview.) Some implementations require intricate mechanisms that link the RDF and relational models via schema, data, and query mappings. It is challenging to design such mappings so they provide a correct SPARQL-to-SQL translation which results in SQL that is also efficient to evaluate. We consider the simplest relational representation of an RDF graph: the triples are stored in a single table triples(s,p,o). For each triple in the RDF dataset, a single tuple is stored. SPARQL queries are translated into SQL that is evaluated by the relational system. Figure 1 shows such a translation.

```
SELECT ?who
{
    ?who :wrote :Hamlet .
}
```

(a) SPARQL

```
SELECT T.s
FROM triples T
WHERE T.p=":wrote" AND
      T.o=":Hamlet"
```

(b) SQL

Fig. 1: Sample SPARQL-to-SQL translation.

SQL Recursion. Recursion was introduced to SQL in the SQL99 standard [4]. This allows for *linear* recursion; i.e., the recursive definition may call itself at most once in its definition. Despite this limitation, linear recursion is quite useful in many practical settings. It allows for queries over hierarchical relationships, to compute bill of materials, and to find paths in graphs. A recursive query is written in SQL via common table expressions (CTEs). A recursive CTE has a *base* and a *recursive* `SELECT` statement. This recursion is then achieved by a join in the recursive `SELECT` with the CTE itself. This can appear only once in the `FROM` clause, which is what limits the recursion to being linear. Figure 2 illustrates a query that computes the transitive closure of a graph stored as an adjacency list.

```
WITH closure(i,j) AS (
     SELECT G.i, G.j FROM graph G               | base step
     UNION ALL
     SELECT C.i, G.j FROM closure C, graph G    | recursive
     WHERE C.j = G.i                            | step
    )
SELECT DISTINCT i, j FROM closure
```

Fig. 2: A recursion SQL query to compute the transitive closure of a graph.

One can limit the recursive depth in the `WHERE` statement in the recursive `SELECT` to avoid infinite recursion over cyclic data. Some database systems implement a `CYCLE` clause that will suppress duplicate matches caused by cycles.

3 Translation Strategy

We are solving the following evaluation problem. Given an RDF graph $G = (V, E)$ stored in the single relational table triples and a property-path triple pattern (u, p, v), we want to generate an SQL query that returns all pairs of nodes $(u, v) : u, v \in V$ such that there is a path between u and v matching p. To produce a correct SQL mapping, we must first establish the semantics of property paths; i.e., when does a path in a graph conform to a property path.

The problem of finding a constrained path between two nodes in a graph has been well studied. Regular paths are a subclass of such. A path matches if the concatenation of its edges matches the regular expression. Nodes along the path may be visited more than once (*regular* paths), or at most once (*simple* paths). The former notion is preferred over the latter, often for complexity reasons. In [5] the authors showed that regular paths are computable in polynomial combined (data and query) complexity, while simple paths are intractable, even for basic regular expressions.

Initially, W3C had adopted simple path semantics for arbitrary-length property paths in SPARQL 1.1 for the "*" and "+" operators. W3C had also required paths to be *counted*; i.e., to report the number of duplicate pairs (a, b) that correspond to a number of paths between a and b that conform to p. However, [2,6] have shown that both of these requirements are computationally infeasible in many cases. These observations led W3C to drop both simple path and path counting requirements in favor of regular paths and existential semantics.

In [6], dynamic programming was shown to be a good approach to evaluate regular path queries with existential semantics on graphs. We provide an SQL implementation of the algorithm in [6] to translate a SPARQL property path p into an equivalent SQL expression.

Definition 1 ***(SPARQL property path to SQL CTE translation)*** *Let G be an RDF graph, DB_G be its database representation as a single relation triples(s,p,o), and p be a property path. Then,* `trans(p)` *is a procedure that pro-*

duces an SQL CTE `P` *such that its database evaluation* `eval(P,` DB_G`)` *produces a relation* R_p *that contains all pairs of nodes* (s, o) *that conform to* p.

Translation procedure `trans(·)` works as follows. Let $T(p)$ be the parse tree of p, the nodes of which represent subexpressions of p. The hierarchy in $T(p)$ corresponds to the operator precedence specified by the SPARQL standard. Figure 3 presents an example. We build up the SQL query by traversing $T(p)$ bottom-up. For each node in $T(p)$ that corresponds to subexpression s, we generate an SQL CTE `S` that computes all pairs of nodes R_s in the RDF graph that conform to s. During the traversal of the parse tree, CTEs from its children nodes are combined to generate a CTE for the parent node.

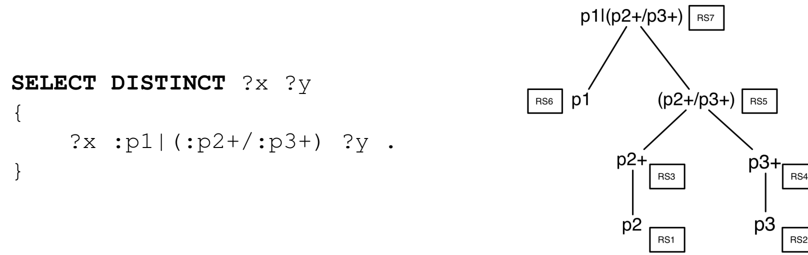

Fig. 3: A property path and its parse tree.

Let $a \in I$ and p, p_1 and p_2 be property paths. We assume set semantics for property paths as advocated in [2, 6, 7]. Below, rules **R1-R8** recursively define the SQL translation for each of the operators that appear in the property path parse tree:

R1: If $p = a$, then p is translated to `P` as in Figure 4a.
R2: If $p = !(a_1 \mid \ldots \mid a_n)$, then p is translated to `P` as in Figure 4f.
R3: If $p = \hat{\ } p_1$, then p is translated to `P` given `P1` as in Figure 4b.
R4: If $p = p_1?$, then p is translated to `P` given `P1` and `P2` as in Figure 4e.
R5: If $p = p_1/p_2$, then p is translated to `P` given `P1` and `P2` as in Figure 4c.
R6: If $p = p_1 \mid p_2$, then p is translated to `P` given `P1` and `P2` as in Figure 4d.
R7: If $p = p_1*$, then p is translated to `P` given `P1` as in Figure 4g.
R8: If $p = p_1+$, then p is translated to `P` given `P1` similarly to **R7**.

Definition 2 ***(Property-path triple to SQL translation)*** *Let* G *be the RDF graph and* DB_G *its database representation as a single relational table* triples(s,p,o). *Let* T *be a property path triple. Then, procedure* `trans(·)` *is overloaded so that given* T*, it produces a semantically equivalent SQL expression* $sql_T =$`trans(`T`)`.

Overloaded `trans(·)` works as follows. Depending on the form of the property path triple T, we generate a different final SQL statement. Rules **R9-R11** (presented in Figure 5) cover the possible cases, assuming that SQL CTE `P` was generated by `trans`(p).

```
WITH P(s,o) AS (
  SELECT DISTINCT T.s, T.o
  FROM triples T
  WHERE T.p = 'a'
)
```

(a) $p = \text{a}$.

```
WITH P(s,o) AS (
  SELECT DISTINCT
    P1.o as s, P1.s as o
  FROM P1
)
```

(b) $p = \hat{}p_1$.

```
WITH P(s,o) AS (
  SELECT DISTINCT P1.s, P2.o
  FROM P1, P2
  WHERE P1.o = P2.s
)
```

(c) $p = p_1/p_2$.

```
WITH P(s,o) AS (
  SELECT * FROM P1
  UNION
  SELECT * FROM P2
)
```

(d) $p = p_1|p_2$.

```
WITH P(s,o) AS (
  SELECT T.s as s, T.s as o
  FROM triples T
  UNION
  SELECT P1.s as s, P1.o as o
  FROM P1
)
```

(e) $p = p_1?$.

```
WITH P(s,o) AS (
  SELECT DISTINCT T.s, T.o
  FROM triples T
  WHERE
    T.p NOT IN ('a1',..,'an')
)
```

(f) $p = !(a_1|\ldots|a_n)$.

```
WITH closure(s,o) AS (
     SELECT P1.s, P1.o FROM P1
     UNION ALL
     SELECT C.s, P1.o FROM closure C, P1 WHERE C.o = P1.s
    ) CYCLE s SET cyclemark TO 'Y' DEFAULT 'N' USING cyclepath,
    P(s,o) AS (
      SELECT DISTINCT s, o FROM closure
      UNION
      SELECT T.s as s, T.s as o FROM triples T
    )
```

(g) $p = p_1*$.

Fig. 4: Translation of SPARQL property paths into SQL CTEs.

```
SELECT
    ?s as ?X,
    ?o as ?Y
FROM P
```

R9: $T = (?X, p, ?Y)$

```
SELECT
    ?o as ?Y
FROM P
WHERE s = 's'
```

R10: $T = (s, p, ?Y)$

```
SELECT
    ?s as ?X
FROM P
WHERE o = 'o'
```

R11: $T = (?X, p, o)$

Fig. 5: Translation of SPARQL property-path triples into SQL.

4 Discussion

Our translation of SPARQL property paths into equivalent SQL queries follows SPARQL 1.1 semantics closely. Showing correctness of this translation can be reasonably straightforward. However, the resulting SQL might be not efficient to evaluate. We envision optimization strategies to generate SQL queries that are simpler and more efficient.

Even modest SPARQL property paths may result in a complex, highly nested SQL queries. These are a challenge for present-day relational optimizers. Our translations may be rewritten to use augmentation-based algorithms [8] to generate a "flat" SQL query for each of the regular path operators. Such flattened SQL statements are typically processed more efficiently by relational query engines than their nested counterparts.

We are studying the optimization of the recursive components in the generated SQL. First, we are investigating the optimizations that speed up the recursion by filtering its base table. These include classical early selection rewrites [9], as well as novel algorithms specific to regular path translation such as pushing of the "/" operator into the "$*$" and "+" operators. Our preliminary evaluation results indicate orders of magnitude improvement in query processing times after incorporation of some of these techniques into the translation algorithm. Second, we shall study more advanced recursion optimization techniques. We are investigating the applicability of magic set transformations [10] that aim to reduce the recursive step deltas.

Next, we shall study the problem of evaluating the resulting SQL on cyclic RDF data. The adoption of regular path semantics by W3C for property paths can lead to infinite computation due to cycles in the graph. Our SQL translation and evaluation need a mechanism to handle cycles gracefully. In general, SQL99 includes a specification of a `CYCLE` clause which performs a memoization to "remember" matches considered so to suppress duplicates, avoiding unbounded recomputation. While, in our translation, we use `CYCLE` clause to handle cycles, in practice this approach has two important shortcomings. First, many open-source and commercial databases do not implement support for this clause, thus limiting the applicability of our approach. Second, our preliminary evaluation results suggest that path memoization introduces a considerable amount of computational overhead. Hence, we need to investigate how to deal with these problems efficiently. We consider alternative translations that deal with cycles such as placing an upper limit on the recursion depth or performing manual path memoization by using auxiliary data structures. Moreover, it has been shown in [6] that it is possible to identify the cyclic data in polynomial time. We shall study if we can incorporate this observation into SQL to decide when to use a cycle-proof, but less efficient evaluation strategy, or a faster approach that assumes acyclicity.

Finally, we shall compare the performance of our approach to other proposed methods of property path evaluation. In particular, we consider the recently proposed FEM framework [11]. FEM was originally developed to answer shortest paths queries on graphs stored in a relational database by *iteratively* applying

Frontier-Expand-Merge operations. In our work, we plan to adapt FEM to answer property path queries and to compare this adaptation to the recursion-based approach presented in this paper. Additionally, we compare both of these approaches to native triplestores such as those included in Jena [12] and Sesame [13] frameworks.

References

1. S. Harris and A. Seaborne. SPARQL 1.1 query language. W3C Working Draft (8 Nov 2012). Available at http://www.w3.org/TR/sparql11-query/.
2. M. Arenas, S. Conca, and J. Pérez. Counting beyond a Yottabyte, or how SPARQL 1.1 property paths will prevent adoption of the standard. In *Proceedings of the 21st international conference on World Wide Web*, pages 629–638. ACM, 2012.
3. Katja Hose, Ralf Schenkel, Martin Theobald, and Gerhard Weikum. Database foundations for scalable RDF processing. *Reasoning Web. Semantic Technologies for the Web of Data*, pages 202–249, 2011.
4. Jim Melton and Alan R Simon. *SQL: 1999-Understanding Relational Language Components*. Morgan Kaufmann, 2001.
5. A.O. Mendelzon and P.T. Wood. Finding regular simple paths in graph databases. *SIAM Journal on Computing*, 24(6):1235–1258, 1995.
6. K. Losemann and W. Martens. The complexity of evaluating path expressions in SPARQL. In *Proceedings of the 31st symposium on Principles of Database Systems*, pages 101–112. ACM, 2012.
7. M. Arenas, Gutierrez C., Miranker D., , J. Pérez, and Sequeda J. Querying Semantic Web Data on the Web. *Sigmod Record 41(4)*, pages 6–17, 2012.
8. B. Elliott, E. Cheng, C. Thomas-Ogbuji, and Z.M. Ozsoyoglu. A complete translation from SPARQL into efficient SQL. In *Proceedings of the 2009 International Database Engineering & Applications Symposium*, pages 31–42. ACM, 2009.
9. C. Ordonez. Optimization of linear recursive queries in SQL. *Knowledge and Data Engineering, IEEE Transactions on*, 22(2):264–277, 2010.
10. F. Bancilhon, D. Maier, Y. Sagiv, and J.D. Ullman. Magic sets and other strange ways to implement logic programs. In *Proceedings of the fifth ACM SIGACT-SIGMOD symposium on Principles of database systems*, pages 1–15. ACM, 1985.
11. Jun Gao, Ruoming Jin, Jiashuai Zhou, Jeffrey Xu Yu, Xiao Jiang, and Tengjiao Wang. Relational approach for shortest path discovery over large graphs. *Proceedings of the VLDB Endowment*, 5(4):358–369, 2011.
12. K. Wilkinson, C. Sayers, H. Kuno, D. Reynolds, et al. Efficient RDF storage and retrieval in Jena2. In *Proceedings of SWDB*, volume 3, pages 131–150, 2003.
13. J. Broekstra, A. Kampman, and F. Van Harmelen. Sesame: A generic architecture for storing and querying RDF and RDF Schema. *The Semantic WebISWC 2002*, pages 54–68, 2002.