WAVEGUIDE: Evaluating SPARQL Property Path Queries

Nikolay Yakovets Parke Godfrey Jarek Gryz
York University
Toronto, Canada
{hush, godfrey, jarek}@cse.yorku.ca

ABSTRACT

The extension of SPARQL 1.1 of *property paths* now offers a type of *regular path query* for RDF graph databases. While eminently useful, these queries are difficult to optimize to evaluate efficiently. We have embarked on a project we call WAVEGUIDE to build a cost-based optimizer for SPARQL queries with property paths. WAVEGUIDE maps the property path to a *waveguide plan* (WGP) composed of *wavefront automata* (WFAs) modeled by (non-deterministic) finite automata. The waveguide plan *guides* the graph search during evaluation. Our WAVEGUIDE prototype illustrates the types of optimizations this approach affords and the performance gains that can be obtained.

1. INTRODUCTION

Graph data has quickly become prevalent with the rise of the Semantic Web, social networks, and data-driven exploration in life sciences. Natural and efficient ways are needed to query over the structure of the graph. *Regular path queries* (RPQs) offer a means to query for nodes connected via matching paths. Support for RPQs has been recently added in the SPARQL query language for RDF data in its latest version, 1.1, via *property paths*.[1]

While eminently useful, property-path queries are challenging to evaluate efficiently and to optimize well. We have embarked on a project that we call WAVEGUIDE to build a highly effective, full-fledged cost-based optimizer for SPARQL queries with property paths. Our approach uses guided search through the graph using *finite state automata* based upon the regular expression of the property path to guide. We are able to gain orders of magnitude performance improvement for many property-path queries, while maintaining comparable performance for others, as the leading SPARQL query engines.

Regular path queries have been considered ever since *semi-structured* data models were first introduced [1, 13]. The complexity of RPQs for graph databases particularly has been well studied [2, 3]. In [11], the idea of employing NFAs to guide search for RPQ evaluation is introduced. (The introduction of *product automata* in [13] can well be considered a precursor to this idea.) In [7], they investigate fixpoint evaluation for property paths. In [18], we considered a mapping of property paths to SQL queries with common table expressions (with SQL recursion). In [19], we present a precursor of WAVEGUIDE that explores fixpoint evaluation for property paths using SQL recursion.

WAVEGUIDE's strategy is based on an *iterative search algorithm* guided by a *query plan*, which we call a *waveguide plan* (WGP), composed of *wavefront automata* (WFAs) modeled by *non-deterministic finite state automata* (NFAs).[2] Within this framework, we are able to express complex query evaluation plans which involve multiple search *wavefronts* that iteratively explore the graph. The *states* (of the automata) of the WGP represent path queries in their own right. States of the plan are materialized selectively during evaluation which allows for re-use of intermediate results. We call such materialized states *path views*.

A SPARQL path query can be potentially evaluated by any number of WGPs. A good plan is the one that achieves a balance of *minimizing*

1. the *search space* that needs to be explored,
2. the *recomputation* of answers as much as possible (through re-use with path views), and
3. the degree of *caching* needed by the plan.

These objectives cannot be optimized independently of one another. To address this, we propose a cost-based method that selects the best WGP based on estimated total cost. Our ultimate goal is a cost-based optimizer for SPARQL queries for RDF databases of the same caliber as cost-based optimizers for SQL queries for relational databases.

The graph exploration for the query's evaluation is driven by an iterative search procedure that is effectively a fixpoint evaluation (semi-naïve and bottom-up [8, 12]). Three steps are performed each iteration: crank, reduce and union.

1. Crank expands the search wavefronts in the graph to produces a set of tuples (a *delta*).
2. Reduce eliminates the duplicates from a delta to counter unbounded computation on cyclic graphs.
3. Union selectively materializes delta into cache.

The iteration stops when no new tuples are produced (i.e., we reach the fixpoint).

Each search wavefront is guided by a *wavefront automa-*

[1] We consider SPARQL queries with *distinct*, so a pair of nodes is considered an answer if there *exists* a path between the pair in the graph that matches the regular expression.

[2] We name these *wavefronts* following the convention in [8].

10.5441/002/edbt.2015.49

ton (WFA), a finite state machine modeled on a nondeterministic finite automaton (NFA). Unlike NFAs, which are used as recognizers of regular expressions on strings, WFAs afford us a number of features related to evaluation of regular expressions on graphs such as use of *seeds*, *append/prepend* transitions, and *path views*.

We demonstrate our WAVEGUIDE prototype via a *query plan designer* for designing and viewing the plans and a *runtime visualizer and profiler* for tracing the guided search evaluation. The interactive demonstration over social-network and life-science datasets highlights the benefits—and the interesting challenges—with our methodology.

In §2, we posit a cost model, discuss costs that arise in property-path evaluation with respect to graph and query characteristics, and present optimization strategies. In §3, we overview the implementation of the WAVEGUIDE prototype. In §4, we present the demonstration scenario.

2. PLAN PERFORMANCE

For a given query, of course, there may be many ways to guide the search. Our *cost model*, in abstract, is essentially to predict the size of the *graph walk*—the number of triples from the RDF store (that is, labeled edges from the graph) that will be joined—during the search evaluation as guided by the plan. We summarize *search cost factors* that can affect the cost (properties of the graph and of resulting pre-paths computed during evaluation) and *optimization methods* that are enabled by waveguide plans which address the search factors, in turn.

2.1 Search Cost Factors

Properties of the graph and of the WGP chosen—thus, the guided search during evaluation in terms of the pre-paths that are computed—will determine the evaluation cost.

Search Cardinalities. The *wavefronts*, that we chose for the WGP determine during the search the intermediate results (pairs of nodes labeled by state, thus connected by valid pre-paths) that are collected each iteration. Just as with different join orders in relational query evaluation, different wavefronts will result in different intermediate delta sizes. These intermediate cardinalities can vary greatly over plans.

To reduce overall search size, we need to choose wavefronts that result in fewer edge walks. WGPs can be costed to estimate their search sizes based on statistics about the graph, such as 1-gram and 2-gram label frequencies. (Such graph statistics can be computed offline for this purpose.)

Solution Redundancy. Each node pair appears at most once in the answer, even if there are multiple paths between the node pair satisfying the query's regular expression. As such, answer-path redundancy arises from two sources. First, in dense graphs, solutions are re-discovered by following conforming, yet different paths. Second, nodes are revisited by following cycles in the graph. Thus, the same answer pair may be discovered repeatedly during evaluation. It is critical to detect such duplicate solutions early in order to keep the search size and search cache small.

Sub-path Redundancy. The paths justifying multiple answer pairs may *share* significant segments (*sub-paths*) in common. This arises, for instance, in dense graphs and with hierarchical structures (e.g., *isa* and *locatedIn* edge labels). Consider the query "?p :locatedIn+ Canada". Every person located in the Annex in Toronto qualifies, since the Annex is located in Toronto located in Ontario located in Canada. The sub-path "Annex :locatedIn+ Canada" is shared by the answer path for each Annex resident.

Because we keep only node-pairs (plus state) in the search deltas, and not explicitly the paths themselves,[3] we may walk these sub-paths many times, recomputing "Annex :locatedIn+ Canada" for each Annex resident.

2.2 Optimization Methods

We consider WGP optimization methods in relation to the search cost factors above.

Choice of Wavefronts. The direction in which we follow edges, and where we start in the graph, with respect to the regular expression will result in different *search cardinalities*. Our choice of WFAs in the plan dictates the wavefront(s).

Reduce. WAVEGUIDE's evaluation strategy is designed to counter solution redundancy. Redundancy of *candidate* solutions is addressed by removal of duplicates against both cache (cache) and delta (delta) by the reduce operation. As a further optimization, once a solution seed-target pair has been discovered, *first-path pruning* (fpp) removes the *seed* from further expansion by the search wavefronts.

Threading / Sub-queries. To counter *sub-path redundancy* requires us to decompose a query into sub-queries. We call this decomposition *threading*, and our waveguide plans accommodate this. The portion of the regular expression that will result in sub-paths that will be shared by many answer paths can be computed independently by a separate wavefront. Sub-path sharing can be predicted by graph statistics to indicate when sub-queries should be considered.

Partial Caching. Delta results are cached during evaluation as we need to check against the cache for redundantly computed pairs. For large intermediate cardinalities, this can be a significant cost. However, some of this cost can be negated. In particular, not every state in the plan's WFAs needs to have its node-pairs cached. Caching is only needed when redundancy is possible, due to cycles in a WFA or in the graph. States without cycles need not be cached.

2.3 An Illustration

We illustrate the impact of different plans (WGPs) on query evaluation over an example query

$Q =$?p :marriedTo/:diedIn/:locatedIn+/:dealsWith+ USA

over the real-world dataset YAGO2s [17] with 229M triples.

P_1: single prepending wavefront USA → ?p.

P_2: single appending wavefront ?p → USA.

P_3: two wavefronts and a join:

?p → :locatedIn+ ?x

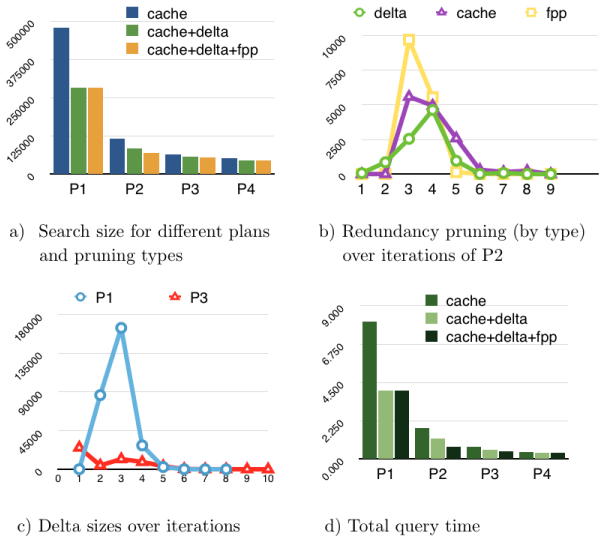
?x :dealsWith+ ← USA.

P_4: P_2 but with a threaded sub-path

:locatedIn+/:dealsWith+ USA.

Fig. 1a shows the effects of wavefront choice on search cardinality. Note the order of magnitude difference between the best, P_4, versus the worst, P_1. The three types of redundancy pruning—cache, delta, and fpp—are illustrated for each plan. Fig. 1b plots search size across iterations for P_2 with pruning; over 40% of tuples are pruned. Fig. 1c plots delta sizes over iterations for P_1 and P_3. Note how the selective search of P_3 is better behaved than the rapid expansion of P_1. In Fig. 1d, the total execution time for each plan is

[3]Note this design choice in our evaluation strategy is critical for good performance, due to solution redundancy.



a) Search size for different plans and pruning types

b) Redundancy pruning (by type) over iterations of P2

c) Delta sizes over iterations

d) Total query time

Figure 1: Effect of plans on query evaluation.

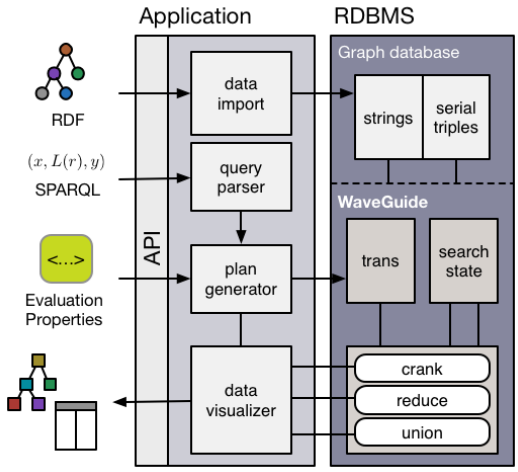


Figure 2: Overview of a prototype system

presented.[4] This demonstrates the significant improvement in performance achievable by careful design of the WGP.

3. THE WAVEGUIDE PROTOTYPE

To demonstrate the efficacy of the WAVEGUIDE evaluation strategy, we focus on evaluation of SPARQL 1.1 property paths over large RDF graphs. We illustrate how WAVEGUIDE can be implemented effectively on a modern relational database system. We use POSTGRESQL due to that it is open-source and has a high-performance procedural SQL implementation. However, any RDBMS with good procedural SQL support could be used.

WAVEGUIDE's resource-intensive tasks can be delegated to POSTGRESQL via SQL and *procedural* SQL routines. This implementation of our methodology gains us high performance, scalability, and rapid deployment.

The architecture of our prototype is shown in Fig. 2. It consists of two layers: *application* and *RDBMS*. The application layer provides a user front-end, preprocessing the graph data, parsing user queries, generating WGPs, and visualizing key steps during the search. The RDBMS layer provides postprocessing of the graph data and performing the iterative WAVEGUIDE graph search for the given WGP.

We implement the application layer of the WAVEGUIDE prototype in Java. The layer consists of four main modules: a data importer, a query parser, a plan generator, and a data visualizer.

Data importer. This validates RDF data encoded in common formats (e.g., N-triples and Turtle.). It converts these to a tab-separated value format for bulk loading in the RDBMS.

Query parser. We use the Apache ANTLR open-source framework to parse SPARQL 1.1 property path query strings into an internal tree representation.

Plan generator. Given the query parse tree, we produce a base WGP from an NFA that recognizes the regular expression of the query. We then employ a simple greedy WGP generation algorithm using the label cardinality estimates from the graph database. The produced WGP can be manually tuned by the end user via a *graphical evaluation plan designer* (shown on the left in Fig. 3).

Data visualizer. We employ the GRAPHSTREAM open-source library [5] to perform the graph visualization in our system. This allows us to visualize dynamically the key steps involved in the WAVEGUIDE search process. We interface with the RDBMS to visualize the search cache at each iteration of the crank, reduce, and union steps. To provide technical insight to the WAVEGUIDE process, we display a number of relevant evaluation parameters and statistics (shown on the right in Fig. 3).

The RDBMS acts both as the graph store and as the execution platform for WAVEGUIDE's iterative algorithms.

Graph database. We represent a graph database in a single logical triples table, which is decomposed into two physical tables—strings and a surrogate serialTriples—to reduce storage space and improve performance. The surrogate table is indexed in all six ways—spo, sop, pos, pso, osp, and ops—to accommodate the guided search.

Guided search. We implement the guided search process via a procedural SQL program. The WGP that guides the search process is encoded in the trans table. To improve the performance, the cache is stored in an unlogged, ephemeral searchCache table. This is indexed to cover the access paths used by the iterative search. We implement profiling functions here to feed evaluation statistics to the data visualizer.

4. DEMONSTRATION

We design scenarios for three demonstration *objectives* for demonstrating our prototype system: 1. *familiarization*, 2. *challenges*, and 3. *efficacy*. Due to the sizes of the datasets, we deploy the database layer of the prototype in the "cloud" in *Amazon EC*^{2}.

To *familiarize* researchers with our methodology, we demonstrate WAVEGUIDE evaluation of a number of simple property path queries over well-known RDF datasets such as FOAF [6] and DBPedia [4]. We construct the queries to

[4]The queries were run on a 2xXeon E5-2640v2 CPU server with 7200RPM HDD running Ubuntu Server 12.04 x64 and PostgreSQL 9.3.

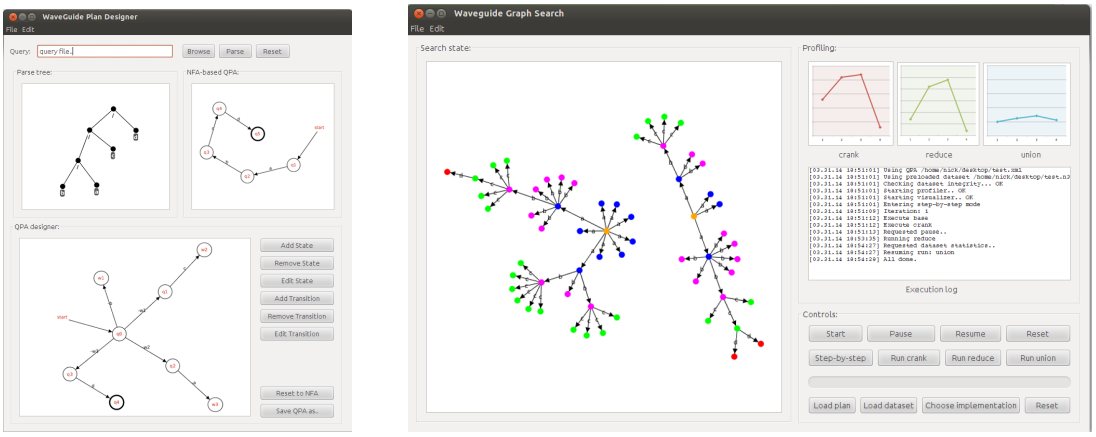


Figure 3: *Query plan designer* and *runtime visualizer and profiler.*

be fairly selective such that the whole evaluation process is comprehensible when visualized step-by-step.

To highlight the *challenges* of the proposed evaluation process, we design a number of non-trivial queries for various domains such as social networks (e.g., the LDBC Social Network Intelligence Benchmark [14]), life sciences (e.g., UNIPROT [16]), and encyclopedic (e.g., YAGO2s [17]). We present the audience with the query at hand, various statistics about the dataset, and show how to design an efficient evaluation plan using the capabilities offered by WAVEGUIDE's WGP mechanism. We focus on the interesting challenges for WGP optimization: efficient join order, cardinality estimation of simple and transitive paths, simplification of the guiding automaton, and intermediate data re-use.

To demonstrate the *efficacy* of WAVEGUIDE, we perform an online, interactive benchmark on a number of datasets and query loads against the native RDF-store Apache JENA [10]. We show that in many situations WAVEGUIDE outperforms JENA by several orders of magnitude.

5. REFERENCES

[1] S. Abiteboul, D. Quass, J. McHugh, J. Widom, and J. L. Wiener. The Lorel query language for semistructured data. *International Journal on Digital Libraries*, 1(1):68–88, 1997.

[2] P. Barcelo, L. Libkin, A. W. Lin, and P. T. Wood. Expressive languages for path queries over graph-structured data. *Transactions on Database Systems*, 37(4):31, 2012.

[3] D. Calvanese, G. De Giacomo, M. Lenzerini, and M. Y. Vardi. Rewriting of regular expressions and regular path queries. In *Proceedings of the Symposium on Principles of Database Systems*, pages 194–204. ACM, 1999.

[4] The DBpedia knowledge base. http://dbpedia.org/.

[5] A. Dutot, F. Guinand, D. Olivier, Y. Pigné, et al. GraphStream: A tool for bridging the gap between complex systems and dynamic graphs. In *Emergent Properties in Natural and Artificial Complex Systems (Satellite Conference within ECCS)*, 2007.

[6] FOAF: The friend of a friend project. http://www.foaf-project.org/.

[7] A. Gubichev, S. J. Bedathur, and S. Seufert. Sparqling Kleene: Fast property paths in rdf-3x. In *Workshop on Graph Data Management Experiences and Systems*, pages 14–20. ACM, 2013.

[8] J. Han, G. Qadah, and C. Chaou. The processing and evaluation of transitive closure queries. In *Advances in Database TechnologyâĂŤEDBT'88*, pages 49–75. Springer, 1988.

[9] S. Harris and A. Seaborne. SPARQL 1.1 query language. W3C working draft. http://www.w3.org/TR/sparql11-query/, November 2012.

[10] Apache Jena. https://jena.apache.org/, 2013.

[11] A. Koschmieder and U. Leser. Regular path queries on large graphs. In *Scientific and Statistical Database Management*, pages 177–194. Springer Berlin Heidelberg, 2012.

[12] M. J. Mäher and R. Ramakrishnan. Déjà vu in fixpoints of logic programs. In *Proc. North American Conf. on Logic Programming*, pages 963–980, 1989.

[13] A. Mendelzon and P. Wood. Finding regular simple paths in graph databases. *SIAM Journal on Computing*, 24(6):1235–1258, 1995.

[14] M.-D. Pham, P. Boncz, and O. Erling. S3G2: A scalable structure-correlated social graph generator. In *Selected Topics in Performance Evaluation and Benchmarking*, pages 156–172. Springer, 2013.

[15] W3C: Resource Description Framework (RDF). http://www.w3.org/TR/rdf-concepts/, 2004.

[16] UniProt: Protein knowledgebase. http://www.uniprot.org/.

[17] YAGO2s: A high-quality knowledge base. http://yago-knowledge.org/resource/. Max Planck Institut Informatik.

[18] N. Yakovets, P. Godfrey, and J. Gryz. Evaluation of SPARQL property paths via recursive SQL. In L. Bravo and M. Lenzerini, editors, *AMW*, volume 1087 of *CEUR Workshop Proceedings*. CEUR-WS.org, May 2013.

[19] N. Yakovets, P. Godfrey, and J. Gryz. Waveguide: Toward cost-based evaluation of sparql property path queries. In *Proceedings of the 4th International Workshop on Semantic Search Over the Web VLDB*. ACM, 2014.