WIREFRAME: Two-phase, Cost-based Optimization for Conjunctive Regular Path Queries

Parke Godfrey[†] Nikolay Yakovets[‡]
Zahid Abul-Basher[§] Mark Chignell[§]

[†]York University, Toronto, Canada
godfrey@yorku.ca
[‡]Eindhoven University of Technology, Eindhoven, The Netherlands
n.yakovets@tue.nl
[§]University of Toronto, Toronto, Canada
{zahid,chignell}@mie.utoronto.ca

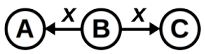
1 Introduction

Graph-like data has come to the forefront with the advent of social networks and the Semantic Web. As methods and tools have been deployed for handling graph data, many others have found the paradigm to benefit their applications. Large biological corpora fit the model well.

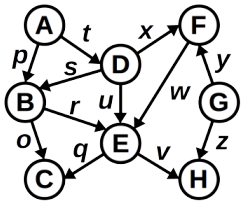
The W3C standards offer a graph data model via RDF, and a corresponding declarative query language for it via SPARQL. The queries are suited to *subgraph matching* into a large data graph. Each subgraph answer itself is small, but there can be *many* answers. The query language encompasses the earlier defined concept of *regular path queries* (RPQs) and *conjunctive regular path queries* (CRPQs). An RPQ queries for *pairs* of nodes from the data graph which have a requisite path between them which matches the RPQ's specification. CRPQ queries for *tuples* of nodes (subgraph matches) such that pairs within match by specified RPQs. This, of course, also entails that the pairs join up (conjoin) in the way the query asks. One can then call a CRPQ a "query graph": its "nodes" are the query's binding variables; and the "edges" between nodes are the constituent RPQs. Let these query edges be distinctly *labeled*.

While graph databases and their applications are coming into wide use, we are only at the very beginning of understanding how to scale these systems well. Recent work has brought a cost-based optimization approach to RPQs [6]. We set out a framework herein which we call WIREFRAME for a two-phase, cost-based optimization for CRPQs. In WIREFRAME, CRPQ *planning*—and, likewise, *evaluation*—is separated into two phases. In the first phase, the plan is for evaluating the "answer graph". In the second phase, a plan is posited for enumerating the subgraph-match answer tuples from this answer graph.

In this brief paper, we provide the motivation for WIREFRAME, present its general framework, and lay out our agenda for, and the challenges in, realizing the approach fully.



(a) Query graph *symmetry*.



(b) Many-many join *multiplicity*.

Fig. 1: Two query graphs: symmetry & join multiplicity.

2 WIREFRAME's Approach

Within SPARQL, a CRPQ query is with a fixed number of named node variables,[1] and a number of "edge" conditions defined via RPQs (property paths in SPARQL) between node variables. Let us define "closed SPARQL", cSPARQL, to encompass this CRPQ fragment with the proviso that each query edge (RPQ) is distinctly labeled.[2] A query in cSPARQL then can be considered as a labeled, directed multi-graph, the *query graph*, the same as the graph database itself (the *data graph*), albeit, significantly smaller, over which it is to be evaluated. Let the evaluation of a cSPARQL query *also* be a labeled, directed multi-graph, the *answer graph*. As with RDF, let the answer graph be represented as an *answer set* of *triples*: the "edges" are node pairs—with the nodes drawn from the data graph—labeled by the query graph's edge labels. Each answer edge means that the pair of nodes is an *answer* to the corresponding RPQ. And each answer edge must participate in *some* subgraph match that answers the CRPQ. This treats graph query evaluation *algebraically*: the result of the evaluation is a graph itself.

How to evaluate efficiently the answer-graph triples of a cSPARQL query with respect to a data graph is a worthwhile endeavor in its own right. This can also be used to evaluate the query with respect to SPARQL, to find the *subgraph match* (SGM) tuples. The set of answer triples *suffices* to enumerate the SGM tuples. This approach to CRPQ evaluation for SPARQL then is two-phase.

1. *optimization.* How to *evaluate* the answer graph most efficiently.
2. *enumeration.* How to *enumerate* the SGM tuples most efficiently.

The two phases have quite different optimization criteria; separating the steps allows for us to plan them "independently". Division between the answer graph and the subgraph-match answers also allows for *factorization*, which escapes from the combinatoric complexity that arises from subgraph match symmetries during evaluation against the data graph. At the core of planning in *each* phase is the choice of a *spanning tree* in the query graph. For the answer-graph phase, this directs the *graph walk* during evaluation. For the subgraph-match phase, this directs the enumeration of the SGM answer tuples for the CRPQ.

[1] Some may be *free* which are returned as part of an answer's bindings, and the others *existential* which are not. Query "edge" conditions may be between any of the node variables. Let us consider here only queries with *free* node variables.

[2] This has been introduced in SPARQL with the use of CONSTRUCT operator [3].

At one end of the spectrum, a CRPQ is simply sub-graph match template. Let each query edge be a data-graph label (the simplest RPQ). This type of SPARQL query has now been well studied [4]. The number of sub-graph matches can be extremely large, due to two causes: *symmetry* in the query graph; and *multiplicity* from "many-many joins". For example, if $\langle 1, 2, 3\rangle$ (on $\langle A, B, C\rangle$) is an answer tuple to query graph in Fig. 1a, then so is $\langle 3, 2, 1\rangle$. If *sub-tuple* $\langle 4, 5\rangle$ on $\langle D, E\rangle$ appears in m sub-tuple answers on $\langle A, B, C, D, E\rangle$ *and* appears in n sub-tuple answers on $\langle D, E, F, G, H\rangle$ to query graph in Fig. 1a, then $\langle 4, 5\rangle$ appears in $m \times n$ answer tuples (on $\langle A, B, C, D, E, F, G, H\rangle$).

Our two-phase approach will often result in better plans for pure sub-graph match queries. It improves extremely, however, for CRPQs where the query edges are complex RPQs. The *graph walks* for evaluating the RPQs are not effectively repeated due to multiplicities resulting from SGM matching. Given a CRPQ, let k be the number of node variables, n be the number of triples in its answer graph, and s be the number of its SGM tuples. Most always $3n \ll ks$. This is akin to *factorization* in relational queries. In certain application scenarios, it has been demonstrated that solving *projections* of the query can offer immense performance improvement, as the "final" answers are many-many joins of the projected answer tuples [2]. This is *always* the scenario for CRPQs. The ultimate factorization of a CRPQ is down to node pairs—the answers to the individual RPQs—which are our very answer-graph triples.

A way to evaluate the answer graph is to use a *rooted spanning tree* of the query graph. Evaluate the RPQs emanating from the root node. Only triples for each RPQ that *join* at the root may be expanded upon. Call query edges that are not in the spanning tree *filter* edges. On reaching a node with filter edges, we can only expand a node value (to find triples with respect to the RPQs in the spanning tree from this node) which matches via triples across the filters on the corresponding nodes. (This is *sideways information passing*.) This operation is quite similar to what is called a *semi-join*; optimizations for semi-join [5] can be employed. When a node value fails the filter test ultimately, those answer triples are removed, recursively up to the root. The evaluation proceeds recursively, expanding nodes down the spanning tree. WIREFRAME uses a cost-based planner to choose the most cost-effective spanning tree of the query graph for evaluating the answer graph.

A way to enumerate the sub-graph match tuples given the answer graph is to use a *rooted spanning tree* of the query graph. Join answer triples by the root node to create sub-tuples. A sub-tuple covers partially the spanning tree from the root. Again, call those query edges not participating in the spanning tree filter edges. Whenever a sub-tuple covers a filter edge, check that the pair-projection of the tuple is an answer triple labeled by that edge. Proceed recursively. WIREFRAME uses a cost-based planner to choose the most cost-effective spanning tree of the query graph for enumerating the SGM tuples from the answer graph.

Note that the spanning tree chosen for answer-graph evaluation is *not* likely to be the same spanning tree for SGM-tuple enumeration. The filter edges for the two are used in quite different ways. For answer-graph evaluation, an existence

check for a *node* value must be made over the answer triples for the edge (RPQ). For SGM-tuple enumeration, an existence check for a *node-pair* value is made.

In [6], WAVEGUIDE, a cost-based optimizer that enumerates through a plan space for RPQs, was presented. WIREFRAME employs WAVEGUIDE for planning the RPQs. As a CRPQ also encompasses joins between the RPQs, the results of one constrain the results of others. Choice of a spanning tree in the query graph specifies a topological sort of the RPQs. The cardinality reductions conferred by an earlier RPQ on latter ones can be passed to WAVEGUIDE for planning for those. (Choice of the spanning tree dictates that the remaining edges are filters.) Cost considerations can be additionally made on sharing parts of RPQ plans, as RPQs may overlap in their regular expressions, as is explored in [1]. The space of spanning trees can be enumerated to choose a best one.

A cost-based enumeration of spanning trees can be also done to choose the best plan for enumerating the SGM tuples from the answer graph. The cost criteria here are that the multiplicity from the many-many join at each branch in the tree is *minimized*, while also *maximizing* the filtering by the filter edges. Essentially, this is the best plan for "defactorizing" the SGM tuples.

3 Next Steps & Challenges

WIREFRAME offers a viable cost-based optimization methodology for CRPQs. Our next steps in the work are to implement cost metrics, to design a dynamic-programming enumeration for constructing the best spanning trees for both phases, and to devise efficient runtime execution. The challenges are to couple this more tightly with the WAVEGUIDE optimizer for the RPQ planning [6], while also taking advantage of commonalities across the RPQs [1].

References

1. Z. Abul-Basher, N. Yakovets, P. Godfrey, S. Ghajar-Khosravi, and M. H. Chignell. TASWEET: Optimizing Disjunctive Path Queries in Graph Databases. In EDBT, pp. 470–473, March 2017.
2. N. Bakibayev, D. Olteanu, and J. Závodnỳ. FDB: A Query Engine for Factorised Relational Databases. In VLDB, 5(11):1232–1243, 2012.
3. S. Harris, A. Seaborne, and E. Prudhommeaux. SPARQL 1.1 Query Language. W3C Recommendation, 21(10), 2013.
4. J. Pérez, M. Arenas, and C. Gutierrez. Semantics and Complexity of SPARQL. In ISWC, pp. 30–43. Springer, November 2006.
5. K. Stocker, D. Kossmann, R. Braumandi, and A. Kemper. Integrating Semi-join-reducers into State-of-the-art Query Processors. In ICDE, pp. 575–584. IEEE, April 2001.
6. N. Yakovets, P. Godfrey, and J. Gryz. Query Planning for Evaluating SPARQL Property Paths. In SIGMOD, pp. 1–15, ACM, June 2016.