Pushing Complexity Down the Stack

Gregory Todd Williams[1] and Kjetil Kjernsmo[2]

[1] greg@evilfunhouse.com
[2] University of Oslo, Department of Informatics, Postboks 1080 Blindern, N-0316 Oslo, Norway kjetil@kjernsmo.net

Abstract. We believe that there is great potential for performance improvements in making high-level query features visible to low-level data sources. In our work in the PerlRDF and SPARQLKit systems, we have tried to apply this thinking while maintaining flexibility and extensibility. We discuss two major approaches we have taken – growing low-level APIs to handle more complex query operations, and allowing low-level frameworks to participate in query planning – and briefly examine some benefits and challenges to their use.

1 Introduction

Commonly, APIs to triple stores have made it necessary to break down more complex (e.g. SPARQL) queries into individual triple patterns, which are then evaluated against the triple store. This causes many problems: It is hard to use any optimizations on the triple store level, to use any statistics available for index scan efficiency or join order optimization, hard to implement experimental extensions to query languages, and hard to exploit special features of triple stores. Some systems provide an alternative to using these APIs by allowing the triple store to evaluate the entire query. Between these two extremes, we propose to make high-level query features visible to low-level data sources, allowing planning- and evaluation-time customization of the query answering process. We discuss approaches we have implemented in the SPARQLKit[1] project and the new PerlRDF[2] Attean[3] framework, and the challenges we see in using them in the future.

2 Increasing API Capabilities

In the PerlRDF system, we define data stores (either triple or quad stores) at the most fundamental level as objects implementing a method to match triple/quad patterns (along with some other utility methods). However, many of the stores we support are capable of more complex data matching operations, and are very often able to perform these operations more efficiently than the generic system

[1] https://github.com/kasei/SPARQLKit
[2] http://www.perlrdf.org/, also packaged and available in Debian and Ubuntu
[3] https://github.com/kasei/attean

routines. For this reason, our API has been extended over time to support more and more complex matching operations. These operations include:

1. Match triple/quad pattern
   The fundamental matching operation supported by all data store implementations.
2. Match BGP pattern
   This allows stores to implement the matching of multiple triple/quad patterns at once, returning a multiset of solutions (mapping variable names to RDF terms).
3. Match BGP pattern with simple filter
   This operation extends BGP matching with support for filtering intermediate results, leveraging existing data indexing and/or knowledge of the store-internal data layout and representation.
4. Match full SPARQL query
   This allows stores to produce results to entire queries at once, capturing the full range of operations including pattern matching, graph operations, and solution modifiers.

Data store classes may indicate their conformance to one or more of these operations, allowing the query planner to choose the best available. This has allowed store implementations flexibility in choosing a balance between complexity and performance. For example, our relational database store implements APIs 1 and 2 (allowing triple pattern joins to be handled entirely within the database), while the SPARQL Protocol store providing access to a remote SPARQL endpoint implements APIs 1–4, simply forwarding full query string across the network.

This sort of pushing down of query operators can be seen in a limited form as far back as System R[1] where filtering expression "search arguments" could be attached to data access operations, allowing only data satisfying the expression to be returned. In Semantic Web systems, the OpenRDF Sesame[2] SAIL API provides a general mechanism for handling this sort of optimization with the SailConnection `evaluate` method. SAIL implementations that override this method are passed a query representation and must return corresponding query results. In this way, implementations may optimize queries in any way they see fit, but they will also be responsible for the evaluating the *entire* query, which is a situation we have set out to improve upon.

3 Delegating Query Planning

In implementing the SPARQLKit project, and in designing the next generation PerlRDF API, we take a more flexible approach by simply delegating planning decisions to the underlying data stores. To do this, we rely on the trait systems[3] (also known as roles or protocols) of the Perl/Moose and Objective-C languages to define a *QueryPlanner* trait as requiring the single method:

$$plan : Algebra \mapsto Option[Set[Plan]]$$

Any data source conforming to this trait may participate in the query planning process, as described in Algorithm 1. (Here we show the planning process for a graph store composed of discrete triple store implementations, but the same approach can be used for quad stores.)

Algorithm 1: Delegating Query Planner, *dqPlanner*

Input: *graphStore*, a collection mapping graph names to triple store objects
Input: *graph*, the active graph
Input: *algebra*, a SPARQL algebra expression
Output: *plans*, a set of query plans for executing *algebra*

```
1 if graphStore[graph] conforms to the QueryPlanner trait then
2 |   p ← store[graph].plan(algebra) ;
3 |   if p = Some(plans) then
4 |   |   return prunePlans(plans)
5 |
6 return BuiltInPlanner(store, graph, algebra)
```

It is worth noting that due to the composability of traits, triple stores conforming to the *QueryPlanner* trait do not need to rely on inheritance to provide a default behavior. In traditional object-oriented systems (such as the Sesame SAIL API discussed in section 2), optimizing triple store planners are often modeled as inheriting from a system-provided base class which provide the default planning routines. In a trait system, the query planner is able to test each triple store for trait conformance and conditionally call the store's planner, defaulting to the system planning routines. Freeing the planning classes from this unnecessary inheritance allows more flexibility in design and implementation both triple stores and optimizing planners, and shows one of the major benefits to implementors of leveraging a trait system.

A triple store that conforms to the *QueryPlanner* trait may authoritatively return custom query plans for any part of the query algebra it wishes. Alternatively, the triple store may decline the request for planning by returning *None*. For example, the photo library triple store[4] optimizes matching triple patterns of the form { `?s a foaf:Image` } by directly returning the set of known images (as opposed to using the more general triple pattern matching mechanism).

However, the structure of the query algebra may not be in a form the backing store can directly use. Separately, the photo library store can optimize matching of both a geographic metadata BGP such as
{ `?image dcterms:spatial [ geo:lat ?lat ; geo:long ?long ]` } and a depiction BGP such as { `?image foaf:depicts ?person` }. The store cannot directly optimize a BGP query combining all of these triple patterns:

[4] https://github.com/kasei/GTWApertureTripleStore

```
?image foaf:depicts ?person ;
       dcterms:spatial [ geo:lat ?lat ; geo:long ?long ]
```

To produce an optimized plan for this BGP the store must synthesize one by joining an optimized sub-plan with a system-generated plan for the remaining triple pattern(s). In this case, the store might choose to first produce an optimized geographic sub-plan, and request the system generate a plan for the remaining depiction triple pattern. The system planner will immediately try to determine if the store can also optimize the depiction triple pattern (which it can in this case). A representation of the final resulting plan would be:

```
Join(
  PhotoStore_GeographicQuery(?image, ?lat, ?long),
  PhotoStore_DepictionQuery(?image, ?person)
)
```

Furthermore, our approach would simplify extensions to SPARQL greatly. For example, all optimizations done in e.g. the stSPARQL extension [4] could be implemented and encapsulated in a trait, and the rest of the query evaluation could be left to the default implementation, which would not have to be modified.

Information integration systems such as Garlic[5, 6] and HERMES[7] have previously explored this sort of pushing down complexity of query planning (and cost estimation) to heterogeneous data providers. We draw heavily on this work, and find that it pairs naturally and to great effect with a trait system. The generality of the RDF data model, extensibility of SPARQL, and flexibility of traits allow a wide range of data sources and query features to be captured by a trait-based planning system. As a result, our systems can be widely applicable without requiring data source schema modeling and other bookkeeping tasks often required by information integration systems.

4 Challenges

We see several challenges to using the approaches described above in a well-designed SPARQL system. The accretion of more and more complex API methods (as described in section 2) over nearly a decade is clearly not sustainable. While there are historical architectural reasons why we did not implement a fully general system like Sesame SAIL's `evaluate`, we believe such generality is the correct approach. However, we see value in performing this sort of store-specific optimization at planning time (as with our delegating planner) rather than during query execution. This provides flexibility to the planning system, allowing further planning, rewriting, and optimization to occur after the custom plan is produced.

The delegating planner method is still somewhat brittle with respect to the exact structure of the query algebra. Specific queries may not be as easy to decompose in a planning store as the synthesized join example in section 3. For

example, even for a store that can produce optimized plans for certain filters and triple patterns, planning may not succeed if unrelated operations appear between them:

`Filter(?o > 10, Extend(`$?z \leftarrow ?o + 1$`, BGP(?s ?p ?o)))`

If the triple store can only optimize BGPs with an optional enclosing filter, this query would not be optimized even though it is semantically equivalent to one in which the filter and BGP operations are adjacent (and therefore available for optimization):

`Extend(`$?z \leftarrow ?o + 1$`, Filter(?o > 10, BGP(?s ?p ?o)))`

Without relying on the query planner to exhaustively test equivalent query plans, a more flexible system is needed to allow recognizing query structures that are well-suited for store-specific optimization.

Finally, custom query plans produced by stores should ideally work with the query planner's cost model to allow the system to compare the relative costs of otherwise opaque custom plans. We don't currently have a good system for allowing this, and instead trust that store-produced plans will always beat system-produced ones (and that all equivalent store-produced plans are equally efficient). In the long run, we believe having an extensible cost model (likely by requiring that query plan implementations conform to an *Auditable* trait, allowing cost information to be accessed) is important for a system that allows custom query plans.

Acknowledgements We thank Toby Inkster, Chris Prather, and Shawn M. Moore for their assistance in applying trait-based programming techniques to the design of the next generation PerlRDF system (and indirectly the SPARQLKit project).

References

1. Astrahan, M.M., Blasgen, M.W., Chamberlin, D.D., Eswaran, K.P., Gray, J.N., Griffiths, P.P., King, W.F., Lorie, R.A., McJones, P.R., Mehl, J.W., Putzolu, G.R., Traiger, I.L., Wade, B.W., Watson, V.: System R: Relational approach to database management. ACM Transactions on Database Systems (TODS) **1**(2) (1976) 97–137
2. Broekstra, J., Kampman, A., Harmelen, F.: Sesame: A Generic Architecture for Storing and Querying RDF and RDF Schema. Proceedings of the 1st International Semantic Web Conference (ISWC) (2002) 54–68
3. Schärli, N., Ducasse, S., Nierstrasz, O., Black, A.P.: Traits: Composable units of behaviour. In Cardelli, L., ed.: ECOOP 2003 – Object-Oriented Programming. Volume 2743 of Lecture Notes in Computer Science. Springer Berlin Heidelberg (2003) 248–274
4. Kyzirakos, K., Karpathiotakis, M., Koubarakis, M.: Strabon: A semantic geospatial dbms. In Cudr-Mauroux, P., Heflin, J., Sirin, E., Tudorache, T., Euzenat, J., Hauswirth, M., Parreira, J., Hendler, J., Schreiber, G., Bernstein, A., Blomqvist, E., eds.: The Semantic Web ISWC 2012. Volume 7649 of Lecture Notes in Computer Science. Springer Berlin Heidelberg (2012) 295–311

5. Roth, M.T., Schwarz, P.M.: Don't scrap it, wrap it! a wrapper architecture for legacy data sources. In: VLDB. (1997) 25–29
6. Roth, M.T., Ozcan, F., Haas, L.M.: Cost models do matter: Providing cost information for diverse data sources in a federated system. In: VLDB. (1999) 599–610
7. Adali, S., Candan, K., Papakonstantinou, Y., Subrahmanian, V.: Query caching and optimization in distributed mediator systems. In: Proceedings of the ACM Sigmod Conference. (1996)