Pushing Complexity Down the Stack

Gregory Todd Williams
greg@evilfunhouse.com

Kjetil Kjernsmo
kjetil@kjernsmo.net

Goal

- Allow low-level components to handle as much complexity as possible
  - Triple store knows more about the data: indexes, structure, layout on disk
  - Can likely execute partial queries (e.g. BGPs) more efficiently than query engine
  - How can we allow this with API design?

PerlRDF Project

- Old approach: More and more complex methods
  - `get_statements( s, p, o )`
  - `get_pattern( triples_and_filters )`
  - `get_sparql( query_string )`
- Query engine would probe triple stores for each of these methods, delegate biggest sub-query possible

Challenges

- Continually growing API
- Arbitrarily chosen granularity of each method (e.g. `get_pattern` handled BGPs and *some* filters)
- Both query planner and triple stores needed updating to leverage each new method

New Approach

- Leverage Traits* to design flexible query planning and triple store APIs
- Cleaner, more concise code with less copy-pasting of functionality across the class hierarchy
- No requirement for shared superclasses

* http://scg.unibe.ch/research/traits/

Trait-based Design

- Two new trait-based systems:
  - Attean
    - In-progress PerlRDF rewrite
    - Entirely new trait-based API
  - SPARQLKit
    - SPARQL 1.1 implementation in Objective-C*

Traits Example

- Simplest *TripleStore*:
  - `get_triples(s,p,o)`
- Trait provides default implementation:
  - `count_triples(s,p,o)`
  - `size(s,p,o)`
- Store may conform to other traits:
  - *MutableTripleStore*, *CacheableTripleStore*, *BulkUpdatableStore*, ***QueryPlanner***

QueryPlanner Trait

- Participation in query planning:
  - `$plans = $store->plan(algebra);`
  - If store can efficiently execute an algebra, returns a custom QueryPlan object which is preferred to other plans
  - Simplifies and generalizes old API
  - Similar to existing run-time approaches in SAIL API, and information integration/wrapper systems, but providing benefits of proper query plans

Challenges

- Plans should be comparable with an *Auditable* trait and `cost()` method
- Structure of query algebra matters; may need more flexible system for stores to choose only parts of an algebra expression
  - `Extend(Filter(BGP()))` vs. `Filter(Extend(BGP()))`

Conclusions

- Trait-based design has yielded us many benefits:
  - Simpler yet more powerful triple store implementations
  - More flexible query planning (leading to more efficient plans)
  - Smaller, more concise codebase
- Thank you
  - `Perl`
    `> cpan Attean`
  - `Debian (unstable and testing*)`
    `> apt-get install libattean-perl`
  - irc.perl.org/#perlrdf
  - http://www.perlrdf.org
  - https://github.com/kasei/ {attean, sparqlkit}