Faceted Views over Large-Scale Linked Data

Orri Erling
OpenLink Software, Inc.
10 Burlington Mall Road
Suite 265
Burlington,MA 01803
U.S.A.
oerling@openlinksw.com

ABSTRACT

Faceted views over structured and semi structured data have been popular in user interfaces for some years. Deploying such views of arbitrary linked data at arbitrary scale has been hampered by lack of suitable back end technology. Many ontologies are also quite large, with hundreds of thousands of classes. We show how we deal with the 221,000 class structure of Yago in an interactive browsing context.

Also, the linked data community has been concerned with the processing cost and potential for denial of service presented by public SPARQL end points. This paper shows how we use Virtuoso for providing timely response on large data sets through a combined text/facet interface and how we can enforce a limit on query processing time while still providing results for arbitrary queries.

Categories and Subject Descriptors

H.5.4 [**Information Systems**]: Hypertext/Hypermedia; H.2.8 [**Information Systems**]: Database Applications

Keywords

Faceted Views, Linked Data, SPARQL, OpenLink Virtuoso

1. INTRODUCTION

The transition of the web from a distributed document repository into a universal, ubiquitous database requires a new dimension of scalability for supporting rich user interaction. If the web is the database, then it also needs a query and report writing tool to match. A faceted user interaction paradigm has been found useful for aiding discovery and query of variously structured data. Numerous implementations exist but they are chiefly client side and are limited in the data volumes they can handle.

At the present time, linked data is well beyond prototypes and proofs of concept. This means that what was done in limited specialty domains before must now be done at real world scale, in terms of both data volume and ontology size. On the schema, or T box side, there exist many comprehensive general purpose ontologies such as Yago[1], Open CYC[2], Umbel[3] and the Dbpedia[4] ontology and many domain specific ones, such as [5]. For these to enter into the user experience, the platform must be able to support the user's choice of terminology or terminologies as needed, preferrably without blow up of data and concomitant slowdown.

WWW2009, April 20-24, 2009, Madrid, Spain.

Likewise, in the LOD world, many link sets have been created for bridging between data sets.Whether such linkage is relevant will depend on the use case. Therefore we provide fine grained control over which `owl:sameAs` assertions will be followed, if any.

Against this background, we discuss how we tackle incremental interactive query composition on arbitrary data with Virtuoso Cluster[**?**][6]. At each step, the user is given a choice of actually existing properties, classes and search hits for refining a search. The user can form combinations of text search and structured criteria, including joins to an arbitrary depth. If queries are precise and select a limited number of results, the results are complete. If queries would select tens of millions of results, partial results are shown instead of counting all properties and class memberships of the results.

The paper is divided into the following parts:

- SPARQL support for large graphs of subclasses and subproperties, into the hundreds of thousands, with no materialization
- SPARQL partial query evaluation for displaying partial results in fixed time
- a facets web service providing an XML interface for submitting queries, so that the user interface is not required to parse SPARQL
- a sample web interface for interacting with this
- sample queries and their evaluation times against combinations of large LOD data sets

2. PROCESSING LARGE HIERARCHIES IN SPARQL

Virtuoso has for a long time had built-in superclass and superproperty inference. This is enabled by specifying the `define input:inference "`*context*`"` option, where context is previously declared to be all subclass, subproperty, equivalence, inverse functional property and same as relations defined in a a given graph. The ontology file is loaded into its own graph and this is then used to construct the context. Multiple ontologies and their equivalences can be loaded into a single graph which then makes another context which holds the union of the information.

```
define input:inference "yago"
prefix cy: <http://dbpedia.org/class/yago/>
select distinct ?s1 as ?c1,
  (bif:search_excerpt (
    bif:vector ('shakespeare'), ?o1 ) ) as ?c2
where {
    ?s1 ?s1textp ?o1 .
    filter (bif:contains (?o1, '"shakespeare"')) .
    ?s1 a cy:Performer110415638 .
  } limit 20
```

This selects all Yago performers that have a property that contains "Shakespeare" as a whole word.

To see the count of subclasses of Yago performer, we can do:

```
prefix cy: <http://dbpedia.org/class/yago/>
select count (*)
from <http://dbpedia.org/yago.owl>
where {
    ?s rdfs:subClassOf cy:Performer110415638
            option (transitive, t_distinct) }
```

There are 4601 distinct subclasses, including indirect ones. We see that an implementation that would for each individual with Shakespeare in some property, of which there are

```
select count (*) where {
    ?s ?p ?o .
    filter (bif:contains (?o, 'shakespeare')) }
```

There are 10267 subjects with Shakespeare mentioned in some literal.

```
define input:inference "yago"
prefix cy: <http://dbpedia.org/class/yago/>
select count (*) where {
    ?s1 a cy:Performer110415638 . }
```

There are 184885 individuals that belong to some subclass of performer.

This is the data that the SPARQL compiler must know in order to have a valid query plan. Since these values will wildly vary depending on the specific constants in the query, the actual database must be consulted as we go for all this information. This is regular query processing technology but is now specially adapted for deep subclass and subproperty structures.

Conditions in the queries are not evaluated twice, once for the cardinality estimate and once for the actual run. Instead, the cardinality estimate is a rapid sampling of the index trees that reads at most one leaf page. With this method, the guiess for the count of performers is 114213, which is acceptably close to the real number.

For example, in the query for performers with Shakespeare, the full text criterion is done first, then the class is retrieved and checked against a memory resident copy of the Yago hierarchy to determine if performer is a superclass.

However, the query

```
define input:inference "yago"
prefix cy: <http://dbpedia.org/class/yago/>
select distinct ?s1 as ?c1,
  (bif:search_excerpt (
      bif:vector ('shakespeare'), ?o1 ) ) as ?c2
where {
    ?s1 ?s1textp ?o1 .
    filter (bif:contains (?o1, '"shakespeare"')) .
    ?s1 a cy:ShakespeareanActors .
  }
```

will start with Shakespearian actors since this is a leaf class with only 74 instances and then check if the properties contain Shakespeare and return their search summaries.

In principle, this is common cost based optimization but is here adapted to deep hierarchies combined with text patterns. An unmodified SQL optimizer would have no possibility of arriving at these results.

The implementation reads the graphs designated as holding ontologies when first needed and subsequently keeps a memory based copy of the hierarchy on all servers. This is used for quick iteration over sub/superclasses or properties as well as for checking if a given class or property is a subclass/property of another. Triples with OWL predicates `equivalentClass`, `equivalentProperty` and `sameAs` are also cached in the same data structure if they occur in the ontology graphs.

Also cardinality estimates for members of classes near he root of the class hierarchy take some time since a sample of each subclass is needed. These are cached for some minutes in the inference context, so that repeated queries will not redo the sampling.

3. INVERSE FUNCTIONAL PROPERTIES AND SAME AS

Specially when navigating social data, as in FOAF[7] and SIOC[8] spaces, there are many blank nodes that are identified by properties only. For this, we offer an option for automatically joining to subjects which share an IFP value with the subject being processed. For example, the query for the friends of friends of Kjetil Kjernsmo returns emppty:

```
select count (?f2) where {
    ?s a foaf:Person ; ?p ?o ; foaf:knows ?f1 .
    ?o bif:contains "'kjetil kjernsmo'" .
    ?f1 foaf:knows ?f2 };
```

But with the option

```
define input:inference "b3sifp"
select count (?f2) where {
    ?s a foaf:Person ; ?p ?o ; foaf:knows ?f1 .
    ?o bif:contains "'kjetil kjernsmo'" .
    ?f1 foaf:knows ?f2 };
```

we get 4022. We note that there are many duplicates since the data is blank nodes only, with people easily represented 10 times. The context `b3sifp` simple declares that `foaf:name` and `foaf:mbox_sha1sum` should be treated as inverse functional properties (IFP). The name is not an IFP in the actual sense but treating it as such for the purposes of this one query makes sense, otherwise nothing would join.

This option is controlled by the choice of the inference

context, which is selecteable in the interface duscussed below.

The issues of run time vs precomputed identity inference through IFP's and `owl:sameAs` are discussed in much more detail at[9].

Our general position is that identity criteria are highly application specific and thus we offer the full spectrum of choice between run time and precomputing. Further, weaker identity statements than sameness are difficult to use in queries, thus we prefer identity with semantics of `owl:sameAs` but make this an option that can be turned on and off query by query.

4. QUERY EVALUATION TIME LIMITS

When scaling the Linked Data model, we have to take it as a given that the workload will be unexpected and that the query writers will often be unskilled in databases. Insofar possible, we wish to promote the forming of a culture of creative reuse of data. To this effect, even poorly formulated questions deserve an answer that is better than just timeout.

If a query produces a steady stream of results, interrupting it after a certain quota is simple. However, most interesting queries do not work in this way. They contain aggregation, sorting, maybe transitivity.

When evaluating a query with a time limit in a cluster setup, all nodes monitor the time left for the query. When dealing with a potentially partial query to begin with, there is little point in transactionality, thus timeouts will occur approximately at the same time in all places, lock waiting not being involved. A read committed query will never block since it will see the before-image of any transactionally updated row.

Thus, when having a partitioned count, for example, we expect all the partitions to time out around the same time and send a ready message with the timeout information to the cluster node coordinating the query. This timeout differs from a run time error in that it leaves the query state intact on all participating nodes. This allows the timeout handling to come fetch any accumulated aggregates.

Let us consider the query for the top 10 classes of things with "Shakespeare" in some literal. This is typical of the workload generated by the faceted browsing web service:

```
define input:inference "yago"
select ?c count (*) where {
    ?s a ?c ; ?p ?o .
    ?o bif:contains "shakespeare" .
  } group by ?c order by desc 2 limit 10
```

On the first execution with an entirely cold cache, it times out after 2 seconds and returns:

yago:class/yago/Entity100001740	566
yago:class/yago/PhysicalEntity100001930	452
yago:class/yago/Object100002684	452
yago:class/yago/Whole100003553	449
yago:class/yago/Organism100004475	375
yago:class/yago/LivingThing100004258	375
yago:class/yago/CausalAgent100007347	373
yago:class/yago/Person100007846	373
yago:class/yago/Abstraction100002137	150
yago:class/yago/Communicator109610660	125

The next repeat gets about double the counts, starting with 1291 entities.

With a warm cache, the query finishes in about 300 ms (4 core Xeon, Virtuoso 6 Cluster) and returns:

yago:class/yago/Entity100001740	13329
yago:class/yago/PhysicalEntity100001930	10423
yago:class/yago/Object100002684	10408
yago:class/yago/Whole100003553	10210
yago:class/yago/LivingThing100004258	8868
yago:class/yago/Organism100004475	8868
yago:class/yago/CausalAgent100007347	8853
yago:class/yago/Person100007846	8853
yago:class/yago/Abstraction100002137	3284
yago:class/yago/Entertainer109616922	2356

The well known fact is that running from memory is thousands of times faster than from disk.

The query plan begins with the text search. The subjects with "Shakespeare" in some property get dispatched to the partition that holds their class. Since all partitions know the class hierarchy, the superclass inference runs in parallel, as does the aggregation of the `group by`. When all parttitions have finished, the process coordinating the query fetches the partial aggregates, adds them up and sorts them by count.

If a timeout occurs, it will most likely occur where the classes of the text matches are being retrieved. When this happens, this part of the query is reset, but the aggregate states are left in place. The process coordinating the query then goes on as if the aggregates had completed. f there are many levels of nested aggregates, each timeout terminates the innermost active, thus a query is guaranteed to return in no more than n timeouts, where n is the number of nested aggregations/subqueries.

5. FACETS WEB SERVICE

The Virtuoso Facets web service is a general purpose RDF query facility for facet based browsing. t takes an XML description of the view desired and generates the reply as an XML tree containing the requested data. The user agent or a local web page can use XSLT for rendering this for the end user. The selection of facets and values is represented as an XML tree. The rationale for this is the fact that such a representation is easier to process in an application than the SPARQL source text or a parse tree of SPARQL and more compactly captures the specific subset of SPARQL needed for faceted browsing. The web service returns the SPARQL source text also, thus this can serve as a basis for handcrafted queries.

The query has the top level element `<query>`. The child elements of this represents conditions pertaining to a single subject. A join is expressed with the property or property-of element. This has in turn children which state conditions on a property of the first subject. Property and property-of elements can be nested to an arbitrary depth and many can occur inside one containing element. In this way, tree-shaped structures of joins can be expressed.

Expressing more complex relationships, such as intermediate grouping, subqueries, arithmetic or such requires writing the query in SPARQL. The XML format is for easy automatic composition of queries needed for showing facets, not a replacement for SPARQL.

Consider composing a map of locations involved with

Napoleon. The XML representation of the search is edited by simple user actions.

- Enter in the search form "napoleon":

```
<query inference="" same-as="" view3=""
  s-term="e" c-term="type">
  <text>napoleon</text>
  <view type="text" limit="20" offset="" />
</query>
```

- Select the "types" view:

```
<query inference="" same-as="" view3=""
  s-term="e" c-term="type">
  <text>napoleon</text>
  <view type="classes" limit="20" offset="0"
    location-prop="0" />
</query>
```

- Choose "MilitaryConflict" type:

```
<query inference="" same-as="" view3=""
  s-term="e" c-term="type">
  <text>napoleon</text>
  <view type="classes" limit="20" offset="0"
    location-prop="0" />
  <class iri="yago:ontology/MilitaryConflict" />
</query>
```

- Choose "NapoleonicWars":

```
<query inference="" same-as="" view3=""
  s-term="e" c-term="type">
  <text>napoleon</text>
  <view type="classes" limit="20" offset="0"
    location-prop="0" />
  <class iri="yago:ontology/MilitaryConflict" />
  <class iri="yago:class/yago/NapoleonicWars" />
</query>
```

- Select "any location" in the select list beside the "map" link, then hit "map" link:

```
<query inference="" same-as="" view3=""
  s-term="e" c-term="type">
  <text>napoleon</text>
  <class iri="yago:ontology/MilitaryConflict" />
  <class iri="yago:class/yago/NapoleonicWars" />
  <view type="geo" limit="20" offset="0"
    location-prop="any" />
</query>
```

This last XML fragment corresponds to the below SPARQL text:

```
select ?location as ?c1 ?lat1 as ?c2 ?lng1 as ?c3
where {
    ?s1 ?s1textp ?o1 .
    filter (bif:contains (?o1, '"napoleon"')) .
    ?s1 a <yago:ontology/MilitaryConflict> .
    ?s1 a <yago:class/yago/NapoleonicWars> .
    ?s1 ?anyloc ?location .
    ?location geo:lat ?lat1 ; geo:long ?lng1 .
  }
limit 20  offset 0
```

The query takes all subjects with some literal property with "Napoleon" in it, then filters for military conflicts and Napoleonic wars, then takes all objects related to these where the related object has a location. The map has the objects and their locations.

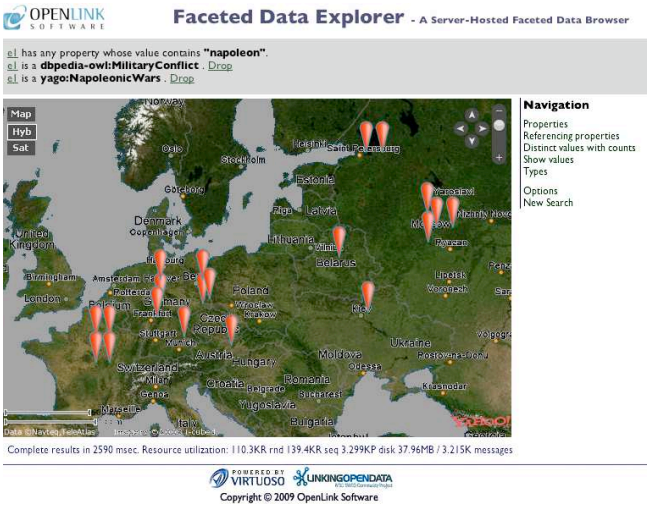

Figure 1: The displayed result

6. VOID DISCOVERABILITY

A long awaited addition to the LOD cloud is the Vocabulary of Interlinked Data (VoID)[10]. Virtuoso automatically generates VoID descriptions of data sets it hosts.

Virtuoso incorporates an SQL function `rdf_void_gen` which returns a Turtle representation of a given graph's VoID statistics.

7. TEST SYSTEM AND DATA

The test system is 2 8 core Xeon 5345 (2.33GHz) servers with 16G RAM and 4 disks each. The machines are connected by 2x 1Gbit ethernet. The software is Virtuoso 6 Cluster. The Virtuoso server is split into 16 partitions, 8 for each machine. Each parrtition is managed by a separate server process.

The test database has the following data sets:

- Dbpedia 3.2
- Musicbrainz
- Bio2RDF
- Neurocommons
- Uniprot
- Ping The Semantic Web (1.6 million miscellaneous files from `http://www.pingthesemanticweb.com`).

Ontologies:

- Yago
- Open CYC
- Umbel
- Dbpedia

8. FUTURE WORK

All the functions discussed above are presently being productized for delivery with Virtuoso 6, so that single servers are open source and clusters commercial only. The most relevant future work is thus final debugging and tuning of existing functionality.

The technology will be first commercially used as a platform for an Amazon EC2 offering of the whole LOD cloud on a cluster of servers. This complements the existing line of data sets pre-packaged by OpenLink[11].

For more sophisticated, also editable user facing functionality, OpenLink is presently working with the developers of OntoWiki[12] on integrating the functionality discussed here into OntoWiki as a new large-scale back-end. From this developement, we expect to have the functional equivalent of Freebase[13], except with more data, working with open, standard data models, being more integrable and above all having a full range of deployment options. This means anything from the desktop to the data center with either software as service or installation at end user sites as options.

For better relevance of results, we are planning to look into applying technologies from web serch to the data web, for example ranking results by a form of page rank. We expect having semantics associated with links to open new possibilities in this domain.

9. CONCLUSIONS

We have shown that we can support a point and click interface for interactive query composition against a database of well over a billlion triples. We have also shown that it is possible to create complex sequences of joins with a few clicks, while constantly seeing what types of data are available for selecting.

Packaging this as a web service is justified, as we expect most applications to require domain specific tailoring in navigation, preferred data display formats and the like. Thus we keep the kernel and our demo interface clearly separate.

The principal technical conclusion from this is the absolutely central role of a good query cost model. Specifically, the cost model must be aware of class and property hierarchies, must have a cardinality estimate for text conditions and must work off the data itself. The range of queries and joins is too diverse and the data distribution too uneven for histograms or other precomputed statistics to work well.

10. REFERENCES

[1] Suchanek, F.M.; Kasneci, G.; Weikum, G.: YAGO: A Core of Semantic Knowledge Unifying WordNet and Wikipedia. WWW2007, ACM 978-1-59593-654-7/07/0005.

[2] Overview of OpenCyc. http://www.cyc.com/cyc/opencyc/overview

[3] UMBEL Ontology, Vol. 1: Technical Documentation, TR 08-08-28-A1. http://www.umbel.org/doc/UMBELOntology_vA1.pdf

[4] Auer, S.; Bizer, C.; Lehmann, J.; Kobilarov, G.; Cyganiak, R.; Ives, Z.: DBpedia: A Nucleus for a Web of Open Data. In Aberer et al. (Eds.): The Semantic Web, 6th International Semantic Web Conference, 2nd Asian Semantic Web Conference, ISWC 2007 + ASWC 2007, Busan, Korea, November 11-15, 2007. LNCS 4825 Springer 2007, ISBN 9783-540762973.

[5] The National Center for Biomedical Ontology: Resources. http://bioontology.org/repositories.html

[6] OpenLink Software, Inc. Virtuoso 6 FAQ. http://virtuoso.openlinksw.com/Whitepapers/html/Virt6FAQ.html

[7] Brickley, D.; Miller, L.: FOAF Vocabulary Specification 0.91. http://xmlns.com/foaf/spec/

[8] Bojars, U.; Breslin, J.G. (eds.): SIOC Core Ontology Specification http://rdfs.org/sioc/spec/

[9] Erling, O.: "E Pluribus Unum", or "Inversely Functional Identity", or "Smooshing Without the Stickiness". http://www.openlinksw.com/dataspace/oerling/weblog/Orri%20Erling's%20Blog/1498

[10] Hausenblas, M.: Discovery and Usage of Linked Datasets on the Web of Data. NodMag #4. Available at http://www.talis.com/nodalities/pdf/nodalities_issue4.pdf

[11] OpenLink Software, Inc. Virtuoso Universal Server (Cloud Edition) AMI for EC2. http://virtuoso.openlinksw.com/wiki/main/Main/VirtuosoEC2AMI

[12] Auer, S.; Dietzold, S.; Riechert, T.: OntoWiki A Tool for Social, Semantic Collaboration. 5th International Semantic Web Conference, Nov 5th–9th, Athens, GA, USA. In I. Cruz et al. (Eds.): ISWC 2006, LNCS 4273, pp. 736-749, 2006. Springer-Verlag Berlin Heidelberg 2006.

[13] Metaweb Technologies, Inc.: What is Freebase? http://www.freebase.com/view/en/what_is_freebase