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.

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 discusses how we use Virtuoso Cluster Edition for providing interactive browsing over billions of triples, combining full text search, structured querying and result ranking. We discuss query planning, run time inferencing and partial query evaluation. This functionality is exposed through SPARQL, a specialized web service and a web user interface.

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, partial query evaluation, entity ranking, large ontologies

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, preferably without blow up of data and concomitant slowdown.

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].

Using SPARQL or a web/web service interface, 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.

The system being described is being actively developed as of this writing, early March of 2009 and is online at `lod.openlinksw.com`. The data set is a combination of Dbpedia, Musicbrainz, Freebase, web crawls from `www.pingthesemanticweb.com`, Uniprot, Neurocommons, Bio2RDF.

The hardware consists of 2 8 core servers with 16G RAM and 4 disks each. The system runs on Virtuoso 6 Cluster Edition. All application code is written in SQL procedures with limited client side Ajax, the Virtuoso platform itself is in C.

The facets service allows the user to start with a text search or a fixed URI and to refine the search by specifying classes, property values etc., on the selected subjects or any subjects referenced therefrom.

This process generates queries involving combinations of text and structured criteria, often dealing with property and class hierarchies and often involving aggregation over millions of subjects, specially at the initial stages of query composition. To make this work with in interactive time, two things are needed:

1. a query optimizer that can almost infallibly produce the right join order based on cardinalities of the specific constants in the query

2. a query execution engine that can return partial results after a timeout.

It is often the case, specially at the beginning of query formulation, that the user only needs to know if there are relatively many or few results that are of a given type or

LDOW2009, April 20, 2009, Madrid, Spain.
.

involve a given property. Thus partially evaluating a query is often useful for producing this information. This must however be possible with an arbitrary query, simply citing precomputed statistics is not enough.

It has for a long time been a given that any search-like application ranks results by relevance. Whenever the facets service shows a list of results, not an aggregation of result types or properties, it is sorted on a composite of text match score and link density.

The paper is divided into the following parts:

- SPARQL query optimization and execution adapted for run time inference over large subclass structures.
- Resolving identity with inverse functional properties
- Ranking entities based on graph link density
- 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 ontology information from the merged source ontologies.

Let us consider a sample query combining a full text search and a restriction on the class of the desired matches:

```
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.

The `define input:inference "yago"` clause means that subclass, subproperty and inverse functions property statements contained in the inference context called yago are considered when evaluating the query. The built-in function `bif:search_excerpt` makes a search engine style summary of the found text, highlighting occurrences of Shakespeare. The `bif:contains` function in the filter specifies the full text search condition on `?o1`.

This query is a typical example of queries that are executed all the time when a user refines a search. We will now look at how we can make an efficient execution plan for the query. First, we must know the cardinalities of the search conditions:

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.

Next we look at how many Shakespeare mentions 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 needed while preparing the execution plan. 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.

Consider a B tree index, which we descend from top to the leftmost leaf containing a match of the condition. At each level, we count how many children would match and always select the leftmost one. When we reach a leaf, we see how many entries are on the page. From these observations, we extrapolate the total count of matches.

With this method, the guess for the count of performers is 114213, which is acceptably close to the real number.

Given these numbers, we see that it makes sense to first find the full text matches and then retrieve the actual classes of each and see if this class is a subclass of performer. This last check is done against a memory resident copy of the Yago hierarchy, the same copy that was used for enumerating the subclasses of performer.

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 Shakespearean 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 the 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 empty:

```
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 be found.

This option is controlled by the choice of the inference context, which is selectable in the interface discussed below.

The IFP inference can be thought of as a transparent addition of a subquery into the join sequence. The subquery joins each subject to its synonyms given by sharing IFP's. This subquery has the special property that it has the initial binding automatically in its result set. It could be expressed as:

```
select ?f where {
    ?k foaf:name "Kjetil Kjernsmo" .
      { select ?org ?syn where {
            ?org ?p ?key .
            ?syn ?p ?key .
            filter ( bif:rdf_is_sub ("b3sifp", ?p,
                 <b3s:any_ifp>, 3) &&
              ?syn != ?org ) }
      } option (transitive,
t_in (?org), t_out (?syn), t_min (0), t_max (1) )
    filter (?org = ?k) .
    ?syn foaf:knows ?f . }
```

It is true that each subject shares IFP values with itself but the transitive construct with 0 minimum and 1 maximum depth allows passing the initial binding of `?org` directly to `?syn`, thus getting first results more rapidly. The `rdf_is_sub` function is an internal that simply tests whether `?p` is a subproperty of `b3s:any_ifp`.

Internally, the implementation has a special query operator for this and the internal form is more compact than would result from the above but the above could be used to the same effect.

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. ENTITY RANKING

It is a common end user expectation to see text search results sorted by their relevance. The term entity rank refers to a quantity describing the relevance of a URI in an RDF graph.

This is a sample query using entity rank:

```
prefix yago: <http://dbpedia.org/class/yago/>
prefix prop: <http://dbpedia.org/property/>
select distinct ?s2 as ?c1  where {
    ?s1 ?s1textp ?o1 .
    ?o1 bif:contains  'Shakespeare' .
    ?s1 a yago:Writer110794014 .
    ?s2 prop:writer ?s1 .
  } order by desc (<LONG::IRI_RANK> (?s2))
  limit 20  offset 0
```

This selects works where a writer with Shakespeare in some property is the writer.

Here the query returns subjects, thus no text search sum-

maries, so only the entity rank of the returned subject is used. We order text results by a composite of text hit score and entity rank of the RDF subject where the text occurs. The entity rank of the subject is defined by the count of references to it, weighed by the rank of the referrers and the outbound link count of referrers. Such techniques are used in text based information retrieval.[15]

One interesting application of entity rank and inference on IFP's and owl:sameAs is in locating URI's for reuse. We can easily list synonym URI's in order of popularity as well as locate URI's based on associated text. This can serve in application such as the Entity Name Server[14].

Entity ranking is one of the few operations where we take a precomputing approach. Since a rank is calculated based on a possibly long chain of references, there is little choice but to precompute. The precomputation itself is straightforward enough: First all outbound references are counted for all subjects. Next all ranks of subjects are incremented by 1 over the referrer's outbound link count. On successive iterations, the increment is based on the rank increment the referrer received in the previous round.

The operation is easily partitioned, since each partition increments the ranks of subjects it holds. The referrers are spread throughout the cluster, though. When rank is calculated, each partition accesses every other partition. This is done with relatively long messages, referee ranks are accessed in batches of several thousand at a time, thus absorbing network latency.

On the test system, this operation performs a single pass over the corpus of 2.2 billion triples and 356 million distinct subjects in about 30 minutes. The operation has 100% utilization of all 16 cores. Adding hardware would speed it up, as would implementing it in C instead of the SQL procedures it is written in at present.

The main query in rank calculation is

```
select O, P, iri_rank (S)
from rdf_quad table option (no cluster)
where isiri_id(O) order by O;
```

This is the SQL cursor iterated over by each partition. The `no cluster` option means that only rows in this process' partition are retrieved. The `RDF_QUAD` table holds the RDF quads in the store, i.e. triple plus graph. The `S`, `P`, `O` columns are the subject, predicate and object respectively. The graph column is not used here. The textttiri_rank is a partitioned SQL function. This works by using the `S` argument to determine which cluster node should run the function. The specifics of the partitioning are declared elsewhere. The calls are then batched for each intended recipient and sent when the batches are full. The SQL compiler automatically generates the relevant control structures. This is like an implicit map operation in the map-reduce terminology.

An SQL procedure loops over this cursor, adds up the rank and when seeing a new `O`, the added rank is persisted into a table. Since links in RDF are typed, we can use the semantics of the link to determine how much rank is transferred by a reference. With extraction of named entities from text content, we can further place a given entity into a referential context and use this as a weighting factor. This is to be explored in future work. The experience thus far shows that we greatly benefit from Virtuoso being a general purpose DBMS, as we can create application specific data structures and control flows where these are efficient. For example, it would make little sense to store entity ranks as triples due to space consumption and locality considerations. With these tools, the whole ranking functionality took under a week to develop.

5. 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. Therefore the facet service uses read committed isolation. A read committed query will never block since it will see the before-image of any transactionally updated row. There will be no waiting for locks and timeouts can be managed locally by all servers in the cluster.

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. The condition raised by hitting a partial evaluation time limit 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, this 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
```

It is a well known fact 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 partitions 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. If there are many levels of nested aggregates, each timeout terminates the innermost aggregation that is still accumulating results, thus a query is guaranteed to return in no more than n timeouts, where n is the number of nested aggregations or subqueries.

6. FACETS WEB SERVICE

The Virtuoso Facets web service is a general purpose RDF query facility for facet based browsing. It 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. All such queries internally generate SPARQL and the SPARQL generated is returned with the results. One can therefore use this is a starting point for hand crafted 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. Below we list user actions and the resulting XML query descriptions.

- 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 text of SPARQL query:

```
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 200  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.

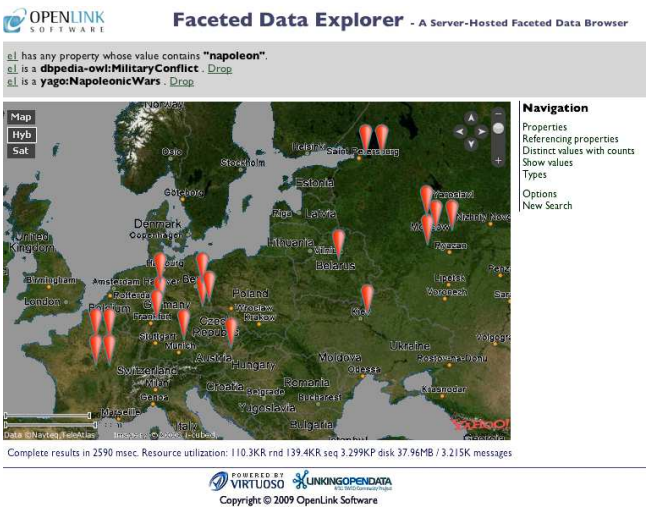

Figure 1: The displayed result

7. 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.

8. TEST SYSTEM AND DATA

The test system consists of two 2x4 core Xeon 5345, 2.33 GHz servers with 16G RAM and 4 disks each. The machines are connected by two 1Gbit Ethernet connections. The software is Virtuoso 6 Cluster. The Virtuoso server is split into 16 partitions, 8 for each machine. Each partition is managed by a separate server process.

The test database has the following data sets:

- Dbpedia 3.2
- Musicbrainz
- Bio2RDF
- Neurocommons
- Uniprot
- Freebase (95M triples)
- Ping The Semantic Web (1.6 million miscellaneous files from `http://www.pingthesemanticweb.com`).

Ontologies:

- Yago
- Open CYC
- Umbel
- Dbpedia

The database is 2.2 billion triples with 356 million distinct URI's.

9. 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 development, 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.

We presently rank search results on text match scores and link density around the URI's related to the text hits. We expect having semantics associated with links to open new possibilities in this domain. We plan to leverage link semantics for ranking but as of this writing have not extensively explored this.

10. CONCLUSIONS

We have presented a set of query processing techniques and a web service and user interface for interactive browsing of a large corpus of linked data. We have shown significant scalability on low cost server hardware, with open ended scale out capacity for larger data set sizes and more concurrent usage.

The service described is online and is also packaged with Virtuoso 6 open source distributions.

The technical experience derived from developing this service emphasizes the following:

- Central importance of a SPARQL/SQL cost model that is aware of hierarchies and is capable of sampling data as needed. Without the right execution plan, no amount of hardware will save the day.
- The importance of enforcing a cap on resource usage.
- The need for scale-out in order to have enough data in memory. Disk is a far greater bottleneck than processor or network speed. Scaling out in a shared nothing fashion is by far the most economical and scalable means of increasing total memory, disk bandwidth and processing power.
- Additional verification of our capacity to schedule parallel query processing on a distributed memory cluster without being killed by latency.
- Confirmation of the Virtuoso platform's flexibility for building additional data intensive services, such as entity ranking.

Present work is therefore concentrated on refining and productizing the platform and its RDF applications. We believe this to be a significant infrastructure element enabling the take off of linked data.

11. 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

[14] Stoermer, H.: Entity Name System: The Back-bone of an Open and Scalable Web of Data. In: Proceedings of the IEEE International Conference on Semantic Computing, ICSC 2008, number CSS-ICSC 2008-4-28-25. IEEE, August 2008. Available at http://www.okkam.org/publications/stoermer-EntityNameSystem.pdf/at_download/file

[15] Brin, S., Page, L.: The Anatomy of a Large-Scale Hypertextual Web Search Engine. In: Seventh International World-Wide Web Conference (WWW 1998), April 14-18, 1998, Brisbane, Australia. Available at http://ilpubs.stanford.edu:8090/361/