Main.VOSScalableInference

SPARQL and Scalable Inference on Demand

By Orri Erling and Ivan Mikhailov

Abstract. This paper discusses integrating inference capabilities into OpenLink Virtuosos SPARQL implementation[1]. Our goal is to do inference at run time and on demand whenever possible, instead of materializing entailed facts ahead of demand. In an open web scenario, facts are liable to change, to be retracted, to be contradictory and to be malicious. Therefore, heavy investment in materializing consequences for a very large body of likely questionable facts is in our view not advisable. In the same spirit, we support partial query evaluation, so as to return possibly incomplete results within a fixed response time window. We present Virtuoso's run time implementation of `owl:sameAs`, inferred identity based on inverse functional properties, generic SPARQL extensions for arbitrary transitive subqueries and partial query evaluation. As future work, we suggest ways of generalizing these features for support of arbitrary backward and forward chaining rules.

Introduction

OpenLink Virtuoso[6] is a general purpose RDBMS with extensive SPARQL and RDF support. The background of the present work is hosting the entire Linked Open Data[5] cloud and various web crawls in Virtuoso's RDF store. From extensive experimentation with the Billion Triples Challenge data set[7][9], we find that many interesting uses of the data apply only to a fraction of the corpus and require run time intelligence, specifically for inferring identity and transitivity. Also, since the corpus is large, almost all queries use a combination of group by and order by, with occasional scalar subqueries. Returning individuals only, without ranking by frequency of occurrence or other aggregate properties does not generally yield a general view on the data set.

Therefore we have extended SPARQL in all the ways required for ad hoc exploration and analysis of complex data without, as a general rule, relying on materialization of entailment as a preprocessing step. This includes addition of aggregation, grouping and subqueries[3] as well as the features discussed below.

An often encountered counter-argument to publishing SPARQL end points is that the Web 2.0 world essentially never gives SQL access to their data, even though the data is generally stored in SQL databases. There are dual reasons for this: 1. data is proprietary, thus offering it for economical reuse by arbitrary third parties is not desired in terms of the business model and 2. SQL queries over large corpora, such as the millions of end user records and tens of millions of related rows of relational data in large Web 2.0 sites are potentially very expensive and constitute a very concrete denial of service threat.

The Linked Open Data world has a slightly different starting point. The data is usually not proprietary and the schema/ontology is open, with a view on making data ad hoc joinable. Web 2.0 silos on the other hand optimize their data model for their relatively fixed workload and do not publish their schema.

Thus, for supporting arbitrarily complex ad hoc queries on very large corpora within bounded

response times, we introduce an anytime query approach that applies to all types of queries and always returns the most complete answer that could be arrived at within the resource constraints. Answers indicate whether they are complete and come with a summary of resource utilization.

The examples below use the Yahoo and Falcon web crawls from the Billion Triples Challenge data set as sample data. The sample size is 135Mtriples, 20GB worth of used database pages. All the samples are run on a single 2.0GHz 2x2core machine with 8G RAM. The software is Virtuoso 6 Cluster with the data in 4 partitions on the same machine.

Transitivity

OWL allows defining a property as transitive. Typically, for a situation where `{ X P Y }` and `{ Y P Z }` and `P` is transitive, the implied fact `{ X P Z }` is added to the corpus of data by a reasoner such as forward chaining engine of Jena's `GenericRuleReasoner` [12]. There are however many cases, such as for example social network data, where the count of transitive `foaf:knows` [11] steps is significant, as well as the reciprocality of the `foaf:knows` relation, etc. In such densely connected and constantly changing graphs, it is difficult to keep the implied facts up to date if all the consequences are materialized.

Jena introduces a path extension for SPARQL[13]. This allows for example saying that `{<Alice> foaf:knows* ?x}`, meaning that `?x` is bound to the transitive closure of all people `<Alice>` knows.

In Virtuoso, we take a more general approach and allow an arbitrary subquery to be made transitive. This has the advantage of being able to also retrieve properties of steps and to have complex conditions for what conditions define relatedness. In the social network example, we can for example return the graph where the foaf:knows triples come from. In a project management case, we could return the length of time associated with each transitive step and so forth. The general form of the transitive subquery is:

```
{ select ?v1, ...
  ( from from-clause )*
    where where-body
}
  option ( transitive t_in ( variables )
               t_out ( variables )
              (t_step ( variable ) as alias)*
              (t_direction direction)?
              (t_min ( const-expn ))?
              (t_max ( const-expn ))?
              t_distinct?
              t_no_cycles?
              t_cycles_only?
              t_shortest_only? )
```

This may occur anywhere in the place of a triple pattern in a SPARQL query.

The variables in the selection are designated as either input, output or data. Conditions in the enclosing query must provide bindings for all input variables, or all output variables or both. For example, we could define sameAs using this feature as follows:

```
select ?syn
where {
  { select ?x ?syn
```

```
    where {
      { ?x owl:sameAs ?syn }
      union
      { ?syn owl:sameAs ?x }
    }
  }
  option (transitive t_in (?x), t_out (?syn), t_distinct,
t_min(0) )
  }
```

In order to use this to iterate over the sameAs closure of <Alice> we would write

```
select ?syn
where {
  { select ?x ?syn
    where
         {
           { ?x owl:sameAs ?syn }
           union
           { ?syn owl:sameAs ?x }
         }
  }
  option (
              transitive t_in (?x),
              t_out (?syn),
              t_distinct,
              t_min (0)
            )
  filter (?x = <Alice>)
        }
```

In this case, we provide a binding for ?x in the filter outside of the transitive subquery. The subquery therefore is made to run from in to out. The same effect would be accomplished if we bound ?syn and select ?x, the designations of in and out are arbitrary and for transitive steps that can be evaluated equally well in both directions this makes no difference.

To find out what graphs contain owl:sameAs for Dan York, we do

```
select ?g count (*)
where {
        { select ?x ?alias ?g
          where {
                  {
                    graph ?g {?x owl:sameAs ?alias }
                  }
                  union
                  {
                    graph ?g {?alias owl:sameAs ?x}
                  }
                }
        }
        option (
                 transitive,
                 t_in (?x),
                 t_out (?alias),
                 t_distinct,
                 t_min (1)
                 ) .
        filter (?
```

```
x=<http://www.advogato.org/person/dyork/foaf.rdf#me>) .
        }
group by ?g
order by desc 2
limit 30
```

For each sameAs alias, this will produce the path from the source, one result row per step, with ?g bound to the graph where the sameAs statement was found. Thus graphs with immediate sameAs get counted extra times, once for the immediate sameAs and once for each path containing this sameAs as an intermediate step.

If we bind both ?x and ?alias, then we get a row of result if there is some combination of owl:sameAs that implies that ?x and ?alias are the same.

We can use this feature to return information about how things are related. For example, using the social web data from the Billion Triples Challenge[8], we can write:

```
select ?o ?dist ((select count (*) where {?o foaf:knows ?xx}))
where {
  { select ?s ?o
    where {
              ?s foaf:knows ?o
            }
  }
    option (
               transitive,
               t_distinct,
               t_in(?s),
               t_out(?o),
               t_min (1),
               t_max (4),
               t_step ('step_no') as ?dist)
    }
  filter (?s = <http://myopenlink.net/person/kidehen#this>)
  }
  order by ?dist desc 3 limit 50
```

This query takes all the people known by kidehen, to a depth between 1 and 4 applications of the subquery. It then sorts them by the distance and the descending count of connections of each found connection. This is equivalent to the default connections list shown by LinkedIn[14].

More interestingly, we can find the distinct paths between two points in a network:

```
select ?link ?g ?step ?path
where {
  { select ?s ?o ?g
    where {
              graph ?g {?s foaf:knows ?o }
            }
  }
  option (
           transitive,
           t_distinct,
           t_in(?s),
           t_out(?o),
           t_no_cycles,
           t_shortest_only,
```

```
            t_direction 3,
            t_step (?s) as ?link,
            t_step ('path_id') as ?path,
            t_step ('step_no') as ?step
          )
  filter (?s = <http://myopenlink.net/person/kidehen#this>)
  filter (?o = <http://www.advogato.org/person/mparaz.rdf#me>)
        }
limit 20
```

This query binds both the t in and t out variables. The text?g is left as a free variable. The ?g is left as a free variable. Also, specifying ?s and the system defined constants step no and path id as with t step, we get for each transitive step a row of results with the intermediate binding of ?s, the count of steps from the initial ?s and a distinct identifier for the individual path, since there can be many distinct paths that link the ?s and ?o specified in the filter.

For evaluating this query, we note that the { ?s foaf:knows ?o } step can be evaluated equally well from ?s to ?o as the reverse, given the index scheme in use on the system. Thus the SPARQL/SQL compiler expands the query into two transitive subqueries, one starting from ?s and the other from ?o. The result is considered complete when the transitive closure being expanded breadth first from ?s first intersects the transitive closure expanded breadth first from ?o or vice versa. The t shortest only flag means that only paths of length equal to the shortest path found will be returned. The t distinct switch means that not all possible paths to intermediate steps on the complete result paths are generated.

After the set of shortest paths is found, the results are returned as a result set, one row per step in each path.

Cost Model and Optimization

When a query contains multiple transitive subqueries joined with each other, the optimal query plan is not readily obvious. The join order options are the same as in the case of any subqueries with distinct or aggregation. The presence of distinctess or aggregation means that the subquery cannot simply be inlined, with component patterns becoming direct component patterns of the enclosing group pattern.

The optimizer must be able to distinguish between tree and graph shaped cases. If the step consists of a predicate with a generally asymmetric cardinality like part-of, where there are more subparts than super-parts, the compiler will naturally prefer the path from the leaves to the root to the path from root to leaves when determining whether two points are parent and child, for example.

It turns out that the general cardinality statistics of the predicates making up the step provide reasonable grounds for such determinations. It is also possible to explicitly state that a transitive subquery must be evaluated from in to out, out to in or from both ends.

owl:sameAs and Identity with Inverse Functional Properties

While powerful, the general form of the transitive subquery is quite verbose and queries making extensive use of such become easily quite unreadable. For this reason we offer two built-in special cases of transitive queries: One for automatic run-time expansion of owl:sameAs and another for inferring identity between two subjects that share a value of an inverse functional property.

The default way of dealing with identity is ”smooshing” the supposedly same URI’s together. This means that all the properties of all the allegedly equal subjects are explicitly asserted for each. If the descriptions are identical, there is no new information but if not, we have duplication. Also, this process loses information since one no longer knows what was originally stated and what was copied.

Exceptions

For example, we could think that subjects that have an equal foaf:mbox sha1sum are the same. This may be so, except when the SHA1 sum is the one for "mailto://". This is filled in by many FOAF generators for an empty email address field. Thus, for each distinct IFP, we allow declaring a set of values which will be considered null, i.e. sharing a null value with another subject will not imply equality.

Distinctness

When multiple URI’s mean the same entity, we can get problems with counting, distinct and grouping. Thus, for better quality of results, we should, for purposes of distinct or group by, consider two instances that are the same through owl:sameAs or through sharing of IFP’s to be the same.

Whenever bindings that come from patterns for which the `sameAs` of IFP inference is enabled are used in a distinct or group by, the SPARQL compiler inserts an extra operation for canonicalizing the value. Since the values in question are IRIs, each with a unique internal ID, for convenience, we use the one with the smallest internal ID as the canonical IRI for these purposes.

Using a subset of the Billion Triples data set, we get:

```
select count (*)
where {?x foaf:knows ?y}

1080205


define input:same-as "yes"
select count (*)
where {
  { select distinct ?x ?y
    where {
                ?x foaf:knows ?y
              }
  }
}

1075161
```

Examples

Consider the graph:

```
<john1> <name> "John" .
<john2> <name> "John" .
<john1> <address> "101 A street" .
```

```
<john2> <address> "102 B street" .
<john2> <knows> <mike> .
<mike> <knows> <john1> .
<mike> <knows> <john2> .
```

We declare <name> to be inversely functional in the context below called ifps.

```
define input:inference "ifps"
select *from <ifps>
where {<john1> <address> ?a};

101 A street
102 B street
```

We get both addresses because <john1> and <john2> are the same by virtue of being called "John".

```
define input:inference "ifps"
select distinct ?x
from <ifps>
where { <mike> <knows> ?x};
```

We get only one because <john1> and <john2> are the same.

Comparison With Materialization

To provide a baseline, we materialized entailment of identity, where identity was defined as having a foaf:name in common and being both instances of foaf:Person. We used the Yahoo and Falcon data sets from the Billion Triples Challenge data set for the experiment. in the unmodified data, there were 3.32M triples in any graph where the subject was in some graph a foaf:Person and had a foaf:name in some graph. This is defined by the below query.

```
select count (*)
where {
  {
    select distinct ?person
    where
      {
        ?person a foaf:Person
      }
  }
  filter (bif:exists ( (select (1)
                               where
                               {
                                 ?person foaf:name ?nn
                               }) )
          )
  ?person ?p ?o
}
```

We collapsed these all into one graph, choosing a canonical ID for all the foaf:Person subjects with the same foaf:name and gave this subject all the properties of all the synonyms. If the object was a member of this set of subjects, the reference was canonicalized. This gave us 2.17M triples. Then we looked at leaving the identities be but collapsing all the persons into a single graph and giving each of the subjects all the properties of all other subjects with the same foaf:name that also were foaf:Persons. This gave us 167.4M triples. No reasoner was used, we

did the operations in SQL so as to get the performance baseline without unknown overheads.

Thus, if the set is static, normalizing identities pays and copying all to all is not profitable, as one would expect. Both approaches lose the provenance information since all are inserted into the same graph. Performing all these manipulations in SPARUL and SQL is error-prone and takes time. Inserting 167M triples does not happen in interactive time even with a lot of hardware. On the test system (4 core Xeon), inserting just one key of the 167M triples took 35m, for 77K inserts per second. On a larger system we could have 100-200K full triples per second. This is still slow. And if we have tens of users doing the same thing at the same time, it is worse still. Actual throughput depends on many factors beyond the scope of this paper but as a ballpark figure for such materialization, we are talking low hundreds of thousands of triples per second for a cluster of 2–4 commodity servers.

If the logic for inferring identity is more complex than comparing IFP values, doing the inference as a preprocessing step makes sense. The preprocessing step can insert owl:sameAs triples that are then followed at run time. Inserting a synthetic IFP value shared by all subjects that are to be considered equivalent is a little more efficient since these do not have to be followed transitively, like owl:sameAs. Such materialization should be done in a separate graph so as not to contaminate the source data.

Copying properties between IRI's considered equivalent is in general discouraged.

Partial Query Answering

The Linked Open Data community has recently seen discussion about the safety and wisdom of offering publicly available SPARQL end points. Also, projects such as Fly Web[15] have experience with hosting end points in Amazon EC2[16] and/or other cloud computing providers. The observation is that users, once they are given the option to compose queries, will compose complex queries that will take long or not complete at all. To this effect, Fly Web has installed diverse restrictions in front of their Jena based back end.

Our approach is different. From Dbpedia onwards, we have had public end points with processing timeouts enforced by the back end database. This is also not perfect since many interesting questions will end in a timeout, which is completely uninformative as concerns the data itself or the query and says nothing about the possibilities of further scoping the search.

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 par- titions 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.

Thus, after activity has timed out, the cluster node coordinating the query can read through the execution plan and find the first/innermost aggregation step that was interrupted. No more information will be added to this aggregate state. Thus the query graph nodes that produce more solutions for the aggregate can be canceled. Instead the aggregated data can now be read and fed to the next stage, for example, the state of a group by can flow into an order by for sorting. Since this is a continuation of the query evaluation, the timeout is reset and some extra time is allocated for postprocessing. If this is interrupted by a new timeout, then results that were unprocessed at this aggregation step are abandoned and processing moves to the output of the next outer aggregation. In this way, any query is guaranteed to finish in a fixed number of steps, each terminated by timeout or by natural completion.

A transitive operation is processed like an aggregation. If a timeout interrupts it, no more results are generated and the results known to date are sent onward. The same applies to subqueries. Typically, if a subquery is at the end of the plan, like in the query counting the friends of friendsand sorting by this count, we will be running the subquery concurrently for a large number of bindings. If this is interrupted, we have multiple partial counts that are then used as they are.

To make this more concrete, let us consider a query that looks for people with a common interest which few people share and who do not know each other.

```
sparql
select ?i ?cnt ?n1 ?n2 ?p1 ?p2
where
  {
    {
      select ?i count (*) as ?cnt
      where
        {
          ?p foaf:interest ?i
        }
      group by ?i
    }
    filter ( ?cnt > 1 && ?cnt < 10) .
    ?p1 foaf:interest ?i .
    ?p2 foaf:interest ?i .
    filter (?p1 != ?p2 &&
    !bif:exists ((select (1) where {?p1 foaf:knows ?p2 })) &&
    !bif:exists ((select (1) where {?p2 foaf:knows ?p1 }))) .
    ?p1 foaf:nick ?n1 .
    ?p2 foaf:nick ?n2 .
  }
order by ?cnt
limit 10
```

The result set has an interest, the count of people having this interest, two person URI's and their foaf:nick's.

The first subquery counts the interested for each interest. The part after this finds two different people with the interest. The next filter checks that they do not know each other. Finally the foaf:nicks are retrieved.

A sample run against the Yahoo and Falcon crawls gives us:

```
http://www.livejournal.com/interests.bml?int=zui
2 zuicidal xcaddlecott_ nodeID://5104696 nodeID://0380826
http://www.livejournal.com/interests.bml?int=zui
2 xcaddlecott_ zuicidal nodeID://0380826 nodeID://5104696
http://www.livejournal.com/interests.bml?int=lt.+george
2 falxxx bitsofstephen nodeID://4697073 nodeID://3662566
...
```

The first thing to time out is the subquery counting the interests. Then the operations feeding the final order by time out. After this the order by returns what state is accumulated. Thus the query produces results in no more than 3 timeouts worth of time.

As a point of comparison, the full evaluation on the Yahoo plus Falcons sample takes 165 seconds whereas the above portion run takes 3. The resource summary line for the full query is

```
22.56MR rnd 1.102GR seq 10P disk 1.341GB / 102.7K messages
```

(Numbers are explained below).

We note that partial results always satisfy all the query criteria. The exception is that when an aggregate is used as a criterion, as in the above query, the aggregate value may be based on incomplete data.

Resource Utilization Metrics

When a query is only partly evaluated, it is necessary to have an idea of how much work went into evaluating it and to have an idea of what percentage of the data set was accessed. Also, for purposes of billing for resource utilization, it is useful to be able to demonstrate the actual utilization. A query by query breakdown of resource utilization is also useful for distinguishing between having a heavily loaded system and a poorly optimized or overly complex query.

To this effect, we gather the count of all single row random accesses, all sequentially accessed rows, all disk reads and the byte and message count of cluster interconnect messages on behalf of a query. Whenever a partial result is returned, the result set ends with an error mentioning these metrics. These metrics are also programmatically available.

For example:

```
sparql select count (*) where {?s ?p ?o};
1933805
*** Error S1TAT: [Virtuoso Driver][Virtuoso Server]RC...:
Returning incomplete results, query interrupted by result
timeout.
Activity: 4R rnd 1.933MR seq 7.28KP disk 939B / 9 messages
```

This is 4 random lookups, one per server, the count's worth of sequential fetches, 28000 pages read from disk, 9 messages between cluster nodes for a total of 939 bytes.

Implications for Scalability

As discussed in [4] the key requirement for scalability on clusters is optimization of message flow, request batching and latency tolerance. All the work presented here is ipso facto

optimized for these operating conditions.

As seen in the previous section, distributed evaluation of partial queries differs only little from single process evaluation of same.

For transitivity, especially when doing this with owl:sameAs, where no or few synonyms are expected per each binding but checking for these is required, we must process the owl:sameAs lookup for batches of several thousand bindings at a time. After enough distinct bindings of a variable are gathered, the lookups are dispatched to the partitions responsible for the data. Results are received and if synonyms are found, these are fed back into the process. The transitive closure is constructed once per distinct binding.

The IFP identity inference is similar, except that it has a maximum depth of 1, i.e. all matches are found in the first lookup.

Future Work

We plan to generalize the Virtuoso transitivity support for evaluating arbitrary backward chaining rules. In such a situation, a step would produce two sorts of outputs: Bindings that are solutions for further processing in the query and lists of sub-goals from rule bodies whose heads matched the input bindings. Rule bodies can be thought of like SPARQL queries and they can be optimized as such, for example as concerns join order when matching physical data. When invoking a rule from a rule, one must pass enough state to allow the processing of the invoked rule to continue once the invoking rule produces bindings. This is no different from an implementation of Prolog.

To maintain good performance on a cluster of machines, we will run with a large set of concurrent bindings. When matching patterns to partitioned data, messages will be batched together and join order optimized as in regular queries.

Conclusions

We believe the features discussed herein to be essential for widespread exploitation of linked data. By eliminating potentially hours long preprocessing of IFP's and `sameAs` triples for identity, we allow the users to decide what sets of `sameAs` assertions they will trust or wish to see, changing this from query to query. Also, the sets of IFPs that will be considered as constituting identity are a matter of use case and opinion. Keeping data sets free of forward chained extra is also important for database performance. Database performance most critically depends on having things in cache. A thousand of random lookups can easily be made in the time of a single disk read. The likelihood of disk access increases if data is blown up to double size by forward chaining. A blow up from 1G to 2G is not significant on the desktop but for clusters, whether one needs 128G or 256G of RAM to have acceptable RAM to disk ratio already makes a significant difference in cost.

Also, providing partial answers in guaranteed time goes a long way towards dissolving the fear of publishing data for open query. For users, it makes the data discovery process more interactive and hence more rewarding. Avoiding materialization of entailment in many cases also makes it possible to make queries on the spur of the moment, without having to design queries days in advance and then waiting for the needed materialization. Materialization is not excluded in the cases where it is really justified, though. If one knows what materialization is needed, then one already has an idea of the workload profile and is no longer in the realm of the pure ad hoc.

Many of the queries discussed in this paper are live on the web in OpenLink's billion triples demo or its successors at http://b3s.openlinksw.com. We also plan to offer the software with pre-loaded data on Amazon EC2.

We believe that the techniques discussed here will significantly contribute to low-cost, creative use of structured data on the web. Together with SPARQL federation, these will enable new types of information value-add services, a whole ecosystem of mesh-ups.

References

1. Erling, O., Mikhailov, I.: RDF Support in the Virtuoso DBMS. Proceedings of the 1st Conference on Social Semantic Web (CSSW), 2007, Leipzig, Germany. LNI 113 GI 2007, ISBN 978-3-88579-207-9: 59-68
2. Erling, O., Mikhailov, I.: Integrating Open Sources and Relational Data with SPARQL. The Semantic Web: Research and Applications, 5th European Semantic Web Conference, ESWC 2008, Tenerife, Canary Islands, Spain, June 1-5, 2008, Proceedings. LNCS 5021 Springer 2008, ISBN 978-3-540-68233-2: 838-842
3. Erling, O., Mikhailov, I.: Business Intelligence Extensions for SPARQL. http://virtuoso.openlinksw.com/dataspace/dav/wiki/Main/VOSArticles/VOSArticleBISPARQL2.pdf
4. Erling, O., Mikhailov, I.: Towards Web-Scale RDF. http://virtuoso.openlinksw.com/dataspace/dav/wiki/Main/VOSArticles/VOSArticleWebScaleRDF.pdf
5. Linked Data — Connect Distributed Data across the Web. http://linkeddata.org/
6. OpenLink Universal Integration Middleware — Virtuoso Product Family. http://virtuoso.openlinksw.com/
7. Semantic Web Challenge. http://challenge.semanticweb.org/
8. Billion Triples dataset. http://www.cs.vu.nl/ pmika/swc/btc.html
9. OpenLink Billion Triple Demo queries. http://b3s.openlinksw.com/
10. OpenLink Billion Triple Demo: Social Connections a la LinkedIn. http://b3s.openlinksw.com/search.vsp?q=6
11. Brickley, D., Miller, L.: FOAF Vocabulary Specification 0.91. http://xmlns.com/foaf/spec/20071002.html
12. Reynolds D.: Jena 2 Inference support. http://jena.sourceforge.net/inference/
13. Seaborne A. (ed.): Jena SPARQL Extensions. http://jena.hpl.hp.com/wiki/SPARQL Extensions
14. LinkedIn – Business-Oriented Social Networking. http://www.linkedin.com/static?key=company info
15. FlyWeb — Linking Laboratory Image Data with Public Databases and Publication Repositories http://www.jisc.ac.uk/whatwedo/programmes/resourcediscovery/flyweb.aspx
16. Amazon Elastic Compute Cloud (Amazon EC2). http://aws.amazon.com/ec2/