Scalable RDF query processing on clusters and supercomputers

Jesse Weaver and Gregory Todd Williams

Rensselaer Polytechnic Institute, Troy, NY, USA
{weavej3,willig4}@cs.rpi.edu

Abstract. The proliferation of RDF data on the web has increased the need for systems that can query these data while scaling with their growing size and number. We present an application of parallel hash-joins for basic graph pattern matching over large amounts of RDF designed for shared nothing architectures including high-performance clusters and the Blue Gene/L. Our approach does not require any pre-processing of the RDF data or costly index building. Rather, we rely on a cluster's high bandwidth and fast memory to load and query data in parallel and in near-real time. We present an initial evaluation of our algorithm showing competitive results on clusters of up to 1,024 processors.

1 Introduction

The web has recently seen a proliferation of structured data. RDF data is now available from many sources across the web relating to a huge variety of topics. Examples of these RDF datasets include the Billion Triples Challenge[1] dataset (collected by a webcrawler from RDF documents available on the web), the Linking Open Data project[2] (in which a number of independent datasets are linked together using common URIs), and the recent conversion[3] of the data.gov[4] dataset to RDF.

With such a large and growing availability of RDF data, new and more efficient ways of querying these data are needed. While most existing systems rely on common database indexing techniques to allow fast retrieval of RDF data, the time required to load and index the data can be prohibitive. In this paper, we present a system for RDF query answering on clusters that does not require any pre-processing, global indexing, or particular assignment of RDF triples to processors. Our system is designed for use on shared-nothing clusters that can range from simple Beowulf clusters to the IBM Blue Gene/L.

By utilizing a cluster's parallelism, our system is able to load, index, and query a large dataset much more quickly than traditional approaches. After data is loaded, our system makes use of a parallel hash-join to answer basic

[1] http://challenge.semanticweb.org/
[2] http://esw.w3.org/topic/SweoIG/TaskForces/CommunityProjects/LinkingOpenData/
[3] http://data-gov.tw.rpi.edu/
[4] http://www.data.gov/

graph pattern queries. Hash-join is a join algorithm that derives its efficiency from partitioning data based on a hash value. We base our work on an existing parallelization of the hash-join algorithm [1]. Our join implementation utilizes an on-the-fly conversion between RDF node values and locally-unique node identifiers to allow efficient join processing without requiring global node identifiers. We differ from most previous work with parallel hash-joins by assuming the presence of a high performance cluster with enough system memory (between all processors) to keep the entire RDF dataset and all intermediate results in memory.

The architecture of our system allows for very fast querying of a dataset. Since RDF data is never pre-processed, this speed enables ad-hoc querying with the ability to add and remove arbitrary amounts of data in subsequent queries with little to no cost. We evaluate our system with several existing datasets on a Linux-based AMD Opteron cluster ranging from 2 to 128 processors, and on a Blue Gene/L from 32 to 1,024 processors.

The rest of this paper is organized as follows. Section 2 reviews related work on parallel hash-join algorithms and other approaches to processing RDF data in distributed and parallel environments. Section 3 presents specific details of our parallel hash-join implementation including parallel loading and indexing of RDF data, hash-based distribution and joining of intermediate results. Section 4 presents an evaluation of our system using several existing RDF datasets and queries. Finally, Section 5 concludes the paper and discusses possible future work in extending our system for reasoning and support for more complex queries.

2 Related Work

Other works on parallel and/or distributed RDF query processing include RDFPeers [2], Continuous RDF Query Processing over DHTs [3], YARS2 [4], Virtuoso[5] [5, 6], GridVine [7], Clustered TDB [8], and 4store[6]. While works like Marvin [9, 10], parallel OWL inferencing [11], and parallel RDFS inferencing [12] use parallelism for semantic web reasoning, they are not directly comparable to the system we present in this paper since we focus on RDF query and not inferencing.

RDFPeers creates a distributed RDF repository over a multi-attribute addressable network (MAAN) [13]. Triples are stored as three attribute-value pairs (subject=..., predicate=..., object=...) on three nodes based on hash values generated from the subject, predicate, and object. RDFPeers provides a query language which maps to MAAN's multi-attribute range queries allowing for distributed querying. [3] focuses on "continuous evaluation of conjunctive triples pattern queries over RDF data stored in distributed hash tables," and GridVine is also a DHT approach. These approaches do not address parallelism and thus differ from our work.

[5] http://virtuoso.openlinksw.com/
[6] http://4store.org/

YARS2, Clustered TDB, Virtuoso (cluster edition), and 4store provide support for RDF stores on clusters. The Clustered TDB work discusses several forms of parallelism: inter-query (running more than one query in parallel), intra-query (running subqueries in parallel and pipelining operators), and intra-operation (distributing single operations for concurrent execution). YARS2 provides fine-grained intra-operation parallelism in triple-pattern matching. The details of Virtuoso and 4store are less certain to us since the finer details of these stores' query evaluation techniques are not published to our knowledge. We differ from these approaches in that we address parallel query processing as a process involving both the loading and computation over RDF data rather than a process occurring over a persistent storage system. For clarity, the system presented herein is not interactive. Queries are queued for evaluation, and our system executes once in its entirety for each query.

Finally, in [1], DeWitt and Gerber show an extension of the hash-join to a multiprocessor environment, and demonstrate its effectiveness in parallel join execution. Our work is based heavily on this extension with two notable exceptions. We restrict our work to all in-memory environments, avoiding the need for variants of the hash-join such as Grace and Hybrid hash-joins that address optimizations in the presence of limited memory. Moreover, every node in our system acts as both a partitioning processor and a joining processor, allowing a join to utilize all available processing power.

3 Methodology

We implement our system in C using the Message Passing Interface[7] (MPI) for interprocessor communication. Each processor maintains an in-memory triple store consisting of three indexes that can directly answer any triple pattern.

In this section we assume the reader is familiar with the following notation from SPARQL[8]. A solution mapping μ is a partial function from variables to RDF terms, and a set of results from a query is a multiset, Ω, of solution mappings. An example solution mapping with two variable bindings might look like {name="Alice", email=<mailto:alice@work.example>}.

3.1 Parallel Hash-Join

In our parallel hash-join implementation, two subqueries are executed independently on each processor i, regardless of the data set held locally on each processor. The results of these subqueries ($\Omega_{1,i}$ and $\Omega_{2,i}$) are then redistributed among the processors in such a way as to ensure that the appropriate results for the join are collocated. This is done by hashing on the values of the variables shared between the two result sets. For example, if the results of the two subqueries join on variables $?a$ and $?b$, then for each solution mapping μ in the results, we hash

[7] http://www.mpi-forum.org
[8] http://www.w3.org/TR/2008/REC-rdf-sparql-query-20080115/#initDefinitions

on the values of $?a$ and $?b$ in μ, and based on that hash value, μ is sent to the appropriate processor. Therefore, solution mappings with the same terms bound to $?a$ and $?b$ will have the same hash value and will get sent to the same processor. After distributing the results, each processor performs the join locally on the received results ($\Omega'_{1,i}$ and $\Omega'_{2,i}$). The redistribution is illustrated in Algorithm 1, while the overall parallel hash-join is illustrated in Algorithm 2 (assuming $\Omega_{1,i}$ and $\Omega_{2,i}$ are available as input after executing the two subqueries). In Algorithm 2, we use "**pardo**" to mean "do in parallel." (For clarity, note that it is allowed for a processor to "send" a solution mapping to itself on line 5. This is a logical description of the algorithm; implementations may wish to handle such sends as a special case.)

For the query evaluation of a basic graph pattern containing n triple patterns, the parallel hash-join algorithm is run $n-1$ times, joining the triple patterns in a so-called left-deep query execution plan. The union on line six of Algorithm 2 represents the logical, complete results of the join. During basic graph pattern evaluation, however, instead of performing this union, each $\Omega'_{1\bowtie 2,i}$ simply becomes the input $\Omega_{1,i}$ for the subsequent join with the results from the next triple pattern in the query execution plan.

Algorithm 1: Distribute Solution Mappings (distmu)

Input: A (local) multiset of solution mappings Ω_i, a set of join variables V, and a number of processors p.
Output: A multiset of solution mappings Ω'_i from redistribution.

```
1  Ω'_i = ∅
2  foreach μ ∈ Ω_i do
3      μ' = project(V, μ)
4      recvr = hash(μ') % p
       // Send μ to recvr.
5      send(recvr, μ)
       // Receive solution mappings from any processor.
6      while recv(∗, μ_r) do
7          add μ_r to Ω'_i
8      end
9  end
10 return Ω'_i
```

3.2 Parallel RDF I/O

We utilize the same approach to loading RDF data in parallel as in [12]. Our only requirement on the input data is that it be in syntax similar to N-Triples[9]. We say similar to the N-Triples syntax because we do not require the data to

[9] http://www.w3.org/TR/2004/REC-rdf-testcases-20040210/#ntriples

Algorithm 2: Parallel Hash Join

Input: Two multisets of solution mappings $\Omega_1 = \bigcup_{i=0}^{p-1} \Omega_{1,i}$ and $\Omega_2 = \bigcup_{i=0}^{p-1} \Omega_{2,i}$, the set of join variables V, and a number of processors p.

Output: A multiset of solution mappings $\Omega_{1 \bowtie 2} = join(\Omega_1, \Omega_2)$.

`// Loop indicates parallelism where` i `is the rank of the processor.`

1 **for** $i = 0$ *to* $p - 1$ **pardo**

`// Ensure solution mappings that can join meet on same processor.`

2 $\Omega'_{1,i} = distmu(\Omega_{1,i}, V, p)$

3 $\Omega'_{2,i} = distmu(\Omega_{2,i}, V, p)$

`// Join local solution mappings.`

4 $\Omega'_{1 \bowtie 2,i} = join(\Omega'_{1,i}, \Omega'_{2,i})$

5 **end**

6 **return** $\bigcup_{i=0}^{p-1} \Omega'_{1 \bowtie 2,i}$

be encoded in 7-bit US-ASCII. The simple format of these N-Triples-like files make parallel reading of the data trivial. Each processor is assigned—in rank-order—a chunk of the input file to read; that is, the i^{th} processor reads the i^{th} consecutive chunk of data. The chunk of data may begin and/or end in the middle of a triple. To handle this, each processor of rank i simply sends the fragment at the beginning of its chunk to processor with rank $i - 1$. Process i then receives such a fragment from processor $i + 1$ and concatenates the triple fragment to the end of its chunk of data. Then, every processor has a set of complete triples which it loads locally into an indexed, in-memory store, converting the serialized RDF nodes into 64-bit identifiers and holding in memory a map for converting between the two (which we will refer to as the nodemap). Unlike most traditional databases and much like many RDF stores, the indexes themselves are the data; there are no data tables holding additional information. Note that while we index the local data on each processor, there is no global index for the entirety of the data (that is, an index over all the data distributed across all processors). This issue is discussed in the following subsection.

At the end of the query, each processor writes out its local set of solutions (the last $\Omega_{1 \bowtie 2,i}$) to its own file using RDF node values (as opposed to the local identifiers). Therefore, our entire query process starts with N-Triples-like files and ends with results containing full RDF node values.

3.3 Communicating Solution Mappings

As mentioned in the previous section, no global indexes are created at any point of the query evaluation. Each processor holds its local triples as 64-bit identifiers and a nodemap. From lines 6 and 7 of Algorithm 1, the solution mappings must be communicated in a way that is meaningful to all processors. This is done by converting the 64-bit identifiers back into string representations before sending

the solution mapping. This allows the receiving processor to assign its own local identifier to the RDF node. While this may incur a higher communication cost, it saves greatly on loading time. Generating, distributing, managing, and performing lookups on global 64-bit identifiers is an extremely time-consuming process, one which we found to be prohibitive. The savings on loading time far exceed the extra cost in communication during query time.

We note that this approach makes loading data inexpensive enough that we can afford to load data for every query evaluation. This allows our system to take advantage of the up-to-date state of the data without costly index maintenance.

While assigning local identifiers, we take advantage of a simple optimization. During line 2 of Algorithm 2, as results are received from the left-hand side of the join (from $\Omega_{1,i}$ of sending processors into $\Omega'_{1,i}$ of receiving processors), a processor assigns new local identifiers to RDF nodes from received solution mappings, placing the new identifiers in a new nodemap. Then, during line 3, as results are received from the right-hand side of the join (from $\Omega_{2,i}$ of sending processors into $\Omega'_{2,i}$ of receiving processors), the RDF terms bound to the join variables in the solution mappings are checked for local identifiers in the nodemap generated from the left-hand side of the query. For each solution mapping, if there is no local identifier assigned to one of its join variables' RDF terms, then we can be certain that there are no results from the left-hand side to which the solution mapping can join. In this case, we can eliminate the solution mapping immediately. Otherwise, if local identifiers exists for all the RDF terms bound to join variables, then the remaining (non-join) variables' RDF terms in the solution mappings are also added to the nodemap. In essence, this simply allows us to eliminate results from the right-hand side without actually attempting the join. This is similar to the effect of the use of bit vector filtering in [1].

4 Evaluation

We evaluated our system on a high performance cluster and a Blue Gene/L supercomputer at Rensselaer Polytechnic Institute's Computational Center for Nanotechnology Innovations[10] (CCNI). Each node of the CCNI high performance cluster is an IBM LS21 blade server running RedHat Workstation 4 Update 5 with two dual-core 2.6 GHz AMD Opteron processors with gigabit ethernet and InfiniBand interconnects. We ran tests on up to 128 processors on medium-memory nodes, each of which has 12GB of system memory. Our testing on the CCNI Blue Gene/L was performed on up to 1,024 nodes, each of which has two 700-MHz PowerPC 440 processors and 512–1024MB of system memory. We utilize three of the Blue Gene/L's specialized hardware networks: a 175MBps 3-dimensional torus for point-to-point communication, a 350MBps global-collective network, and a global barrier network.

[10] http://www.rpi.edu/research/ccni/

We read and write files to/from the large General Parallel File System[11] (GPFS) which has a block size of 1024 KB, scatter block allocation policy, and 256 KB RAID device segment size using a RAID5 storage system.

We evaluate query performance using the Lehigh University Benchmark [14] 20-university dataset (LUBM(20,0)) and on the Barton dataset using queries introduced in [15]. LUBM datasets are synthetically generated datasets containing information about universities. Since LUBM is well-known and widely evaluated against, we provide an evaluation on a LUBM dataset to allow for comparisons with other systems. After generating LUBM(20,0), we used the work from [12] to produce the RDFS closure so as to make the standard LUBM queries meaningful. (For example, for LUBM query 6, no results will be returned unless inferencing is performed to derive that, e.g., all graduate students are students.) The RDFS closure of LUBM(20,0) has 5,159,292 triples. The Barton dataset[12] is an RDF formatted version of the MIT Libraries Barton catalog, and contains 51,598,374 triples.

Much performance tuning can be done by tweaking parameters that affect how Algorithm 1 sends and receives solution mappings. Such parameters include the ratio of transient send messages to transient receive messages and also the frequency at which the processors collaborate to determine whether they have finished distributing solution mappings (a costly operation). The Blue Gene/L has a more sensitive network in that a high number of transient messages can cause the system to effectively fail (ultimately due to memory limitations), and so we set the number of allowable transient messages on the Blue Gene/L lower than on the Opteron cluster. The Blue Gene/L also has an optimized collective network allowing for processors to collaborate to determine termination of Algorithm 1 at a lower cost, and thus we allow the Blue Gene/L to check more frequently for termination than on the Opteron cluster. In our experience, these tuning parameters greatly affect performance and scaling characteristics, and for this evaluation, we have tuned them according to personal experience. However, we have yet to optimize these parameters, and so better performance may be possible.

All figures discussed below use logarithmic axes for both time and number of processors.

Figures 1 through 4 show performance of four of the LUBM queries on the Opteron cluster scaling from 2 to 16 processors, and in general, they show scaling of loading time and total time with respect to the number of processors. Only query two in Figure 1 shows an increase in query time from 8 to 16 processors. This is likely because query two is the only query of the four that requires a high number of joins and does not restrict results to data from university zero. Queries three and four have a bound term that—by the nature of LUBM data—restricts results to data from university zero, and query six has only a single triple pattern (no joins).

[11] http://www-03.ibm.com/systems/clusters/software/gpfs/index.html
[12] http://simile.mit.edu/rdf-test-data/barton/

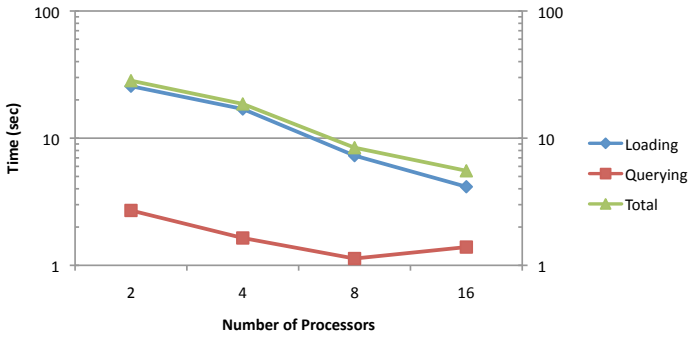

Fig. 1. LUBM(20,0) Query 2 evaluation on Opteron cluster

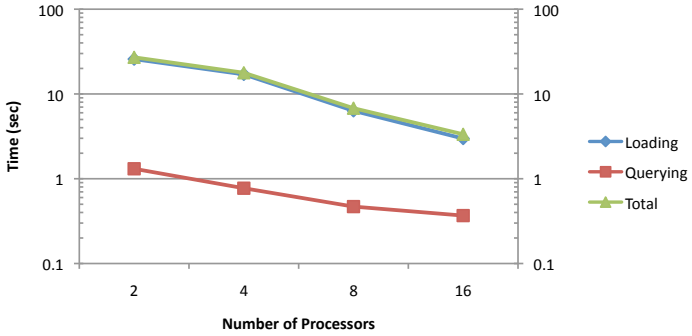

Fig. 2. LUBM(20,0) Query 3 evaluation on Opteron cluster

The Barton dataset is roughly ten times the size of the LUBM(20,0) RDFS closure, and so more memory is needed to perform the evaluation. Figure 5 shows the query execution of Barton query 7 on 32 to 128 processors. The loading time decreases greatly as the number of processors increases, but the query time increases after 64 processors.

In Figure 6, we also show query execution of LUBM query 3 on the LUBM(20,0) RDFS closure using the Blue Gene/L ranging from 32 to 1024 processors. Clearly, loading time scales linearly, but the query time increases after 128 processors.

We notice from Figures 1, 4, and 6 that there seems to be a "sweet spot" for query time only (excluding loading time). Further tweaking of the aforementioned parameters have shown that we can adjust the characteristics of the "sweet spot," but often at a cost. We believe that tuning the parameters based on the number of processors will provide better scaling, and such is left as future work.

Our system competes well with state-of-the-art RDF query systems in loading and query times. Anecdotal evidence indicates that Virtuoso provides the

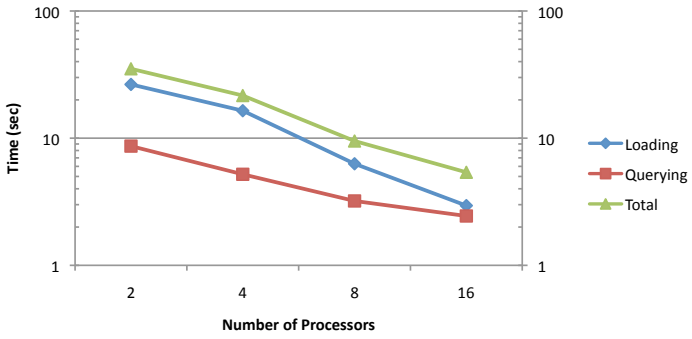

Fig. 3. LUBM(20,0) Query 4 evaluation on Opteron cluster

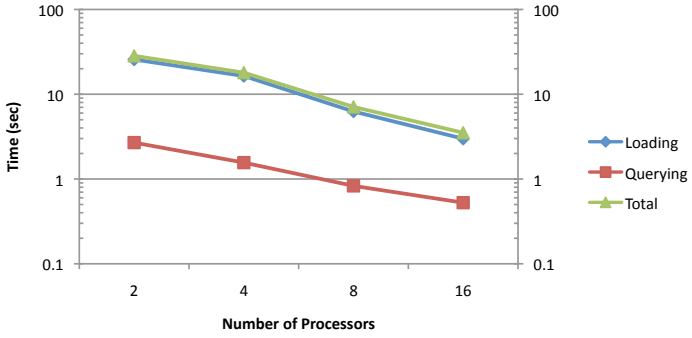

Fig. 4. LUBM(20,0) Query 6 evaluation on Opteron cluster

fastest RDF loading time of any RDF store at 110,532 triples-per-second on eight processors[13]. We report in Figure 3 a loading rate of 820,117 triples per second on eight processors, and we achieve a maximum loading rate of 3,088,172 triples per second on 1024 processors on the Blue Gene/L. RDF-3X [16] seems to be the state-of-the-art in query times. It is difficult to compare our query times to theirs since most of the Barton queries that they use for evaluation use non-standard SPARQL features (e.g., aggregation, "duplicates" keyword, "in" operator) and filters, features which we do not currently support. Therefore, we compare only Barton query 7, the single query that we both support. RDF-3X evaluates Barton query 7 in 32.61 seconds with cold caches (dropping to 1.26 seconds after five runs), whereas we perform the same query in 23.75 seconds on 64 processors. We emphasize, though, that RDF-3X requires 13 minutes to load the Barton dataset after an unreported amount of pre-processing time, whereas

[13] http://www.openlinksw.com/weblog/oerling/index.vspx?page=&id=1562

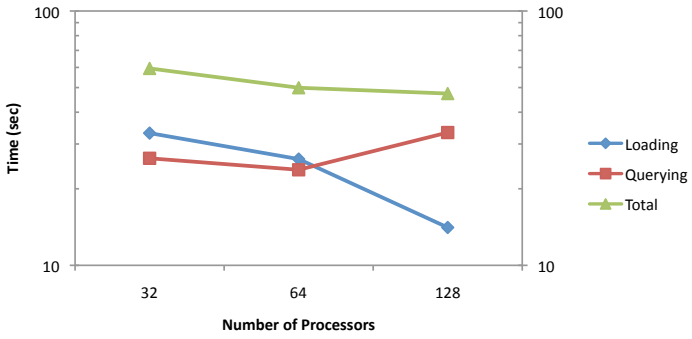

Fig. 5. Barton Query 7 evaluation on Opteron cluster

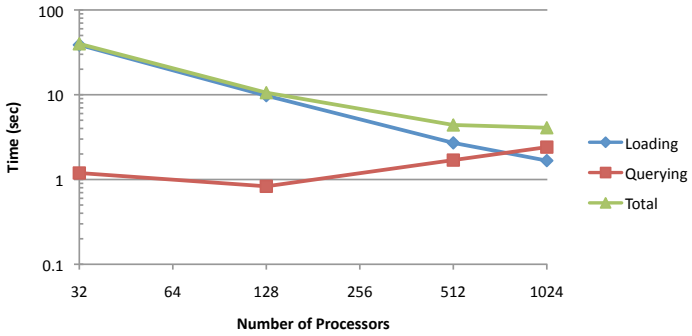

Fig. 6. LUBM(20,0) Query 3 evaluation on Blue Gene/L

our total time (loading and querying) is at lowest 49.92 seconds on 64 processors and 47.37 seconds on 128 processors.

Our system is capable of loading roughly 1.25 million triples from the tested datasets per 1GB of RAM. This total includes storing the full nodemap as well as the three covering indexes. We note that we have spent no time attempting to improve this storage density, but our system should be able to take advantage of compression techniques such as those discussed in [16], significantly improving storage density. While storage density is obviously a concern for an in-memory system like ours, we also note that the primary limitation we faced was not available memory but job queuing time on both the Opteron cluster and Blue Gene/L (both heavily used systems).

5 Conclusion and Future Work

In this paper we have presented a system for answering basic graph pattern queries over large RDF datasets on clusters. Our evaluation has shown our sys-

tem to be competitive with more traditional indexed, persistent triple stores without the need for expensive pre-processing, loading, or indexing of the data. Our results show that some datasets and queries exhibit a "sweet spot" for optimal execution dependent on the number of processors and tuning parameters while others show total time of loading data and query evaluation speed can scale with a constant factor as the number of processors increases.

There are many areas where our system can be improved. Beyond further evaluation and tuning on both the Opteron cluster and Blue Gene/L, we hope to pursue some of the following areas in future work. Currently our system only handles basic graph patterns, but a natural extension would include coverage of SPARQL operators (including optional patterns, named graphs, and filters). Our current hash-join implementation seems to perform well on selective queries, but can have trouble with unselective queries or triple patterns. We are currently investigating a second parallel join algorithm to address queries with unselective triple patterns. We are also pursuing evaluation on larger datasets such as the RDFS closure of LUBM(350,0) (containing roughly 90 million triples) and the Billion Triples Challenge 2009 dataset. Finally, we hope to integrate the work presented in [12] with our system to allow parallel inferencing to occur during query evaluation.

Acknowledgements. We thank Gunnar AAstrand Grimnes for his insightful comments on this work.

References

1. DeWitt, D.J., Gerber, R.H.: Multiprocessor Hash-Based Join Algorithms. In: Proceedings of the 11th International Conference on Very Large Data Bases. (1985) 151–164
2. Cai, M., Frank, M.R.: RDFPeers: a scalable distributed RDF repository based on a structured peer-to-peer network. In: Proceedings of the 13th International World Wide Web Conference. (2004) 650–657
3. Liarou, E., Idreos, S., Koubarakis, M.: Continuous RDF Query Processing over DHTs. In: Proceedings of the 6th International Semantic Web Conference and the 2nd Asian Semantic Web Conference. (2007) 324–339
4. Harth, A., Umbrich, J., Hogan, A., Decker, S.: YARS2: A Federated Repository for Querying Graph Structured Data from the Web. In: Proceedings of the 6th International Semantic Web Conference and the 2nd Asian Semantic Web Conference. (2007) 211–224
5. Erling, O., Mikhailov, I.: RDF Support in the Virtuoso DBMS. In Auer, S., Bizer, C., Müller, C., Zhdanova, A.V., eds.: Proceedings of the 1st Conference on Social Semantic Web. Volume 113 of LNI., GI (2007) 59–68
6. Erling, O.: Toward web scale RDF. In: Proceedings of the 4th International Workshop on Scalable Semantic Web Knowledge Base Systems. (2008)
7. Cudré-Mauroux, P., Agarwal, S., Aberer, K.: GridVine: An Infrastructure for Peer Information Management. IEEE Internet Computing **11**(5) (2007) 36–44
8. Owens, A., Seaborne, A., Gibbins, N., mc schraefel: Clustered TDB: A Clustered Triple Store for Jena. http://eprints.ecs.soton.ac.uk/16974/1/www2009fixedref.pdf (2008)

9. Anadiotis, G., Kotoulas, S., Oren, E., Siebes, R., van Harmelen, F., Drost, N., Kemp, R., Maassen, J., Seinstra, F.J., Bal, H.E.: MaRVIN: a distributed platform for massive RDF inference. http://www.larkc.eu/marvin/btc2008.pdf (2008)
10. Oren, E., Kotoulas, S., Anadiotis, G., Siebes, R., ten Teije, A., van Harmelen, F.: MaRVIN: A platform for large-scale analysis of Semantic Web data. In: Proceeding of the WebSci'09: Society On-Line. (March 2009)
11. Soma, R., Prasanna, V.K.: Parallel Inferencing for OWL Knowledge Bases. In: ICPP '08: Proceedings of the 2008 37th International Conference on Parallel Processing, Washington DC, USA, IEEE Computer Society (2008) 75–82
12. Weaver, J., Hendler, J.A.: Parallel Materialization of the Finite RDFS Closure for Hundreds of Millions of Triples. In: Proceedings of the 8th International Semantic Web Conference. (2009)
13. Cai, M., Frank, M.R., Chen, J., Szekely, P.A.: MAAN: A Multi-Attribute Addressable Network for Grid Information Services. Journal of Grid Computing **2**(1) (2004) 3–14
14. Guo, Y., Pan, Z., Heflin, J.: LUBM: A benchmark for OWL knowledge base systems. Journal of Web Semantics **3**(2-3) (2005) 158–182
15. Abadi, D.J., Marcus, A., Madden, S.R., Hollenbach, K.: Scalable Semantic Web Data Management Using Vertical Partitioning. In: Proceedings of the 33rd International Conference on Very Large Data Bases, VLDB Endowment (2007) 411–422
16. Neumann, T., Weikum, G.: RDF-3X: a RISC-style engine for RDF. Proceedings of the VLDB Endowment **1**(1) (2008) 647–659

Appendix

Below we list the four LUBM queries and one Barton query (defined in [14] and [15], respectively) used in our evaluation. We chose these four LUBM queries as representative and ranging from a single triple pattern (query 6) to a six-way join (query 2). Out of seven original Barton queries, only two can be represented in SPARQL (the others cannot due to their use of aggregates). Of the remaining two queries, we chose query 7 because it can be represented in SPARQL as a simple basic graph pattern.

LUBM Query 2

```
PREFIX : <http://www.lehigh.edu/~zhp2/2004/0401/univ-bench.owl#>
SELECT DISTINCT * WHERE {
  ?z a :Department .
  ?z :subOrganizationOf ?y .
  ?y a :University .
  ?x :undergraduateDegreeFrom ?y .
  ?x a :GraduateStudent .
  ?x :memberOf ?z .
}
```

LUBM Query 3

```
PREFIX : <http://www.lehigh.edu/~zhp2/2004/0401/univ-bench.owl#>
SELECT DISTINCT * WHERE {
  ?x a :Publication .
  ?x :publicationAuthor
    <http://www.Department0.University0.edu/AssistantProfessor0> .
}
```

LUBM Query 4

```
PREFIX : <http://www.lehigh.edu/~zhp2/2004/0401/univ-bench.owl#>
SELECT DISTINCT * WHERE {
  ?x a :Professor .
  ?x :worksFor <http://www.Department0.University0.edu> .
  ?x :name ?y1 .
  ?x :emailAddress ?y2 .
  ?x :telephone ?y3 .
}
```

LUBM Query 6

```
PREFIX : <http://www.lehigh.edu/~zhp2/2004/0401/univ-bench.owl#>
SELECT DISTINCT * WHERE {
  ?x a :Student .
}
```

Barton Query 7

```
PREFIX : <http://simile.mit.edu/2006/01/ontologies/mods3#>
SELECT ?s ?bo ?co
WHERE {
  ?s :point "end" .
  ?s :encoding ?bo .
  ?s a ?co .
}
```