Effective Benchmarking for RDF Stores Using Synthetic Data

Alisdair Owens[1], Nick Gibbins[1], mc schraefel[1]

[1] Department of Electronics and Computer Science, University of Southampton.
SO18 2LT, United Kingdom,
{ao,nmg,mc}@ecs.soton.ac.uk

Abstract. RDF stores are showing consistent performance improvements, with benchmarks showing that several are capable of effectively storing and querying over 10^9 triples. However, detailed information regarding the capabilities of the available systems is limited due to the fact that current benchmarks provide little configurability, and little depth on the strengths and weaknesses of the stores they test. This paper considers the deficiencies of current benchmarks with regards to measuring the performance of RDF stores, and goes on to describe the creation of a new system to run a greater variety of tests using highly configurable synthetically generated datasets. Finally, the benchmark is applied to existing large scale stores, and the results interpreted. This work is intended to inform future RDF store development, and allow application developers to choose a system appropriate to their specific needs.

Keywords: RDF, benchmarking, semantic web, synthetic data

1 Introduction

This paper describes the creation of a new system for benchmarking RDF stores using synthetically generating RDF data. We argue that given the difficulty of creating and maintaining ontologies, it is likely that there will always be a significant amount of RDF data available on the Semantic Web which has not been written to conform to a particular ontology, and that there will thus always be a need to store and retrieve RDF without reasoning. Given this, a means for testing the performance of RDF storage and retrieval is extremely useful.

There are already a variety of benchmarks in existence that are used to test the performance of Semantic Web data stores. Pre-eminent amongst these is the Lehigh University Benchmark (LUBM) [1]. Section 2 explores the strengths and weaknesses of existing systems for the purposes of measuring RDF store performance, as well as considering other benchmarks in use for testing both other DBMSs.

We go on to describe the creation of a new benchmarking system based on highly configurable synthetic data, articulating important test factors for RDF stores and the reason for their inclusion. This new testing system is motivated by a desire to explore

at a low, detailed level the capabilities of a store, and predict its effectiveness for a wide variety of scenarios. We believe that this work will inform the development of future RDF stores by allowing developers to explore in detail the strengths and weaknesses of their systems.

Whilst work is still in progress, this paper describes the current state of the benchmarking system, and provides preliminary benchmark results on pre-release versions of two upcoming stores that claim to be able to effectively store over 10^9 triples: AllegroGraph 3 and BigOWLIM 3.

2 Existing Benchmarks

This section explores benchmarks that are already in existence that can be used to test the performance of RDF stores, and then goes on to explore benchmarking systems for Relational Database Management Systems (RDBMSs).

2.1 RDF Store Benchmarks

There are a variety of benchmarks currently in existence that are commonly used to test RDF stores. As noted, the Lehigh University Benchmark is the most popular of these. LUBM allows the creation of an arbitrarily large amount of RDF data based on a variable number of iterations over a simple OWL ontology. The number of properties attached to any class instance is varied by the use of simple bounded randomisation.

While this benchmark is effective for the purposes of testing OWL inference performance, it is not designed for the purpose it is often used for currently: testing stores based on their RDF query performance. The data produced has a small, heavily repeated structure with few predicates [2].

The method LUBM uses for querying the stores does not reflect likely use cases for RDF stores: each query is repeated ten times, with the average response time being the result [1]. This mechanism gives query caches a large influence over the final result. Further, one query is performed at a time, with no provision for concurrent access. This testing mechanism does not accurately reflect the reality of an open data node on the Semantic Web, where a system might expect to be processing many highly unpredictable queries concurrently, potentially requiring the ability to perform updates while under query load. Finally, it should be noted that LUBM is somewhat out of date, and does not test all of the features (such as regular expression object matching) available in SPARQL.

Despite these issues, LUBM offers a convenient standard for comparing the performance of RDF stores at a high level, and has been used in many tests of both RDF and OWL stores [3-5].

Alternative datasets to that provided by LUBM have emerged. Some of these utilise real world information, such as the UNIPROT[1] and DBpedia[2] sets. While these have the advantage of providing data that is realistic, they are of limited size and do not provide the ability to easily scale the datasets involved. Further, two new use case-based benchmarks are expected to be published in the near future: The SP²B SPARQL Performance Benchmark[3], and the Berlin SPARQL Benchmark[4]. These tests can be expected to fill important roles in offering a wider variety of tests of clear format, testing a greater proportion of the features available in SPARQL than are seen in LUBM. It should be noted, however, that they cannot be expected to provide a highly detailed assessment of the stores that they test: they offer a limited number of queries, and a limited structure to their datasets.

2.2 RDBMS Benchmarks

The pre-eminent benchmarks in the RDBMS world are the Transaction Processing Performance Council family, in particular the TPC-C benchmark [6], which is based on five transactions that model business order and stock management systems, with an expectation that multiple queries will run in parallel. The test produces simple figures for total throughput (overall performance) and throughput per unit cost of the machine. Developers running the tests are expected to provide detailed reports of the hardware upon which the tests were run.

TPC-C provides a convenient standard for judging the overall performance of an On-line Transaction Processing (OLTP) system, but does not make an effort to individually test the components of the system [7], or give a report of what exactly it is that makes the system fast or slow. This is in some ways advantageous, since it simplifies performance comparison, but does not give much information to those who require specific performance characteristics.

The Wisconsin benchmark [8], developed prior to the TPC family, takes a different approach. This system performs a large number of transactions, attempting to test each subcomponent of the system, such as the query optimiser, join mechanism, and so on. This approach gives detailed feedback regarding the performance characteristics of the DBMS, in particular the intelligence of the query optimiser [9]. It attracted criticism for not reducing the reported figures to a single overall metric [7], rendering results harder to interpret and not giving a clear ‘winner’ when comparing two DBMSs. Further, it did not test multi-threaded scenarios, which are extremely likely in real-world situations.

---

[1] http://dev.isb-sib.ch/projects/uniprot-rdf/
[2] http://dbpedia.org/About
[3] http://dbis.informatik.uni-freiburg.de/index.php?project=SP2B
[4] http://www4.wiwiss.fu-berlin.de/bizer/BerlinSPARQLBenchmark/

3 Creating Tests

To create a new testing mechanism, inspiration was drawn from the Wisconsin benchmark for relational DBMSs [8]. We believe that SPARQL [10] is still sufficiently simple for it to be practical to individually test the language features it supports, along with the processing components of the RDF store.

In proposing a new means for generating synthetic data and benchmarking RDF stores, our intention is to produce a means of examining stores at a low, diagnostic level: to not simply be able to judge if a store is overall ‘fast’ or ‘slow’, but to be able to judge which particular features of a store are effective or not, and how they may impact the overall system. We anticipate that this will provide worthwhile insights to inform the development of future generations of RDF stores, as well as for those who have specialised needs for their applications running on top of RDF stores. Finally, we also expect that the test factors identified in this paper will provide insight for the creation of future use-case based RDF benchmarks.

As noted in [11], it is likely that there will be a great many potential use cases for Semantic Web data stores, which may be satisfied by a use case based test for every scenario. We do not believe that it will be practical to manually develop separate datasets and queries for all the conceivable uses of RDF stores, however, and propose this system as a useful tool for testing a variety of scenarios.

In order to accomplish this, we identified a large number of factors that could be tested, divided into the broader categories of assert, delete, and query (for now, a study is made only of SELECT queries). We make no attempt to distinguish the relative importance of these factors beyond identifying them as non-trivial: this is left to the creator of the tests to decide, based on their needs. Ideally, query design should make a significant effort to change queries by as little as possible for each test, to create results that are comparable with other results within the test set, as well as with the results of other systems.

We do not suggest quad-store specific tests, as we expect that for current needs most queries are likely to be limited to specifying a single graph, or all graphs. The individual tests are listed below:

Assertion:

- Bulk assertion time from scratch: this metric tests the store’s performance when asserting a new data graph (including creation of appropriate indexes), assuming that the store is under no other activity.
- Assertion time for a significant addition to a previously existing data graph.
- Assertion time for multiple simultaneous writes: This examines the store’s locking mechanisms: some schemes require a greater or lesser portion of indexes to be locked to perform an update, which can affect read/write times.

- Assertion time when operating under query load: This again tests locking mechanisms.

Deletion:

- Deletion of entire graphs: most stores offer the ability to create multiple graphs, or 'models', to separate distinct datasets. If the store does not perform cross-model inference, deletion of a model and its associated statements ought to be simple.
- Deletion of statements: deletion of individual (or patterns of) statements ought to be a relatively simple process for an RDF-only store. If the store produces forward chained entailments for RDF-S or OWL, it becomes a complex issue due to the need to determine how the inferred statements need to be altered [12].

Query:

- Simple queries that match against a specified subject, predicate, or object (or pair of these), returning a moderate number of results. This provides baseline information, as well as testing storage/indexing methodology: many stores will be indexed and sorted on subjects, and may exhibit decreased performance when testing over predicate or object. The choice of relatively disconnected URIs keeps the result set small for this simple query.
- Repeat of the above, using queries that return large numbers of results. This examines the effect of the number of results returned on query performance, which can be a particular significant bottleneck in stores that internally use id/hash mappings to describe URIs and literals.
- Queries that specify multi-triple graph patterns. These patterns should provide a large capacity for optimisation: one triple should result in the retrieval of very few results compared to the others. This tests the query optimiser's intelligence.
- Repeat of the above, specifying the triples in another order. If the same queries are to be used, this test should be run some time after the above, to reduce any effect of caching.
- Queries that specify multi-triple graph patterns, returning many results, not amenable to optimisation, that is, each graph pattern returns many results: this is a test of the store's ability to perform joins over large quantities of data.
- Complex queries that specify many triple patterns. This tests join performance and query optimisation.
- Performance of the system in an environment where it experiences multiple simultaneous queries: does it degrade in performance gracefully as the load increases, or not?
- Queries that return no result. This again tests the query optimiser's ability to perform operations in an optimal manner: if the triple that causes no result to be found is run early, this should return very quickly.

- The effect of using FILTER or UNION to specify more than one distinct URI for a predicate, subject, or object. This tests the intelligence of the query optimiser with regards to its ability to optimise use of FILTER.
- The effect of specifying OPTIONAL sections within a query.
- The effect of the use of the REGEX expression when searching for a range of object values within a result set.
- Performance of numerically restrictive FILTER statements.
- Performance of LIMITed queries versus their non-LIMITed equivalents, and the time taken to retrieve subsequent ranges of information from previously run LIMITed queries.
- Performance of cached queries versus their non-cached equivalents.

After the bulk assert, a set of 'warm up' queries should be run to give the store a chance to reach peak efficiency. Generally, any query that runs for a relatively short period of time should be repeated to account for short-lived external factors affecting the result. To reduce the impact of query caching on the result, the query should be modified each time it is run, but stay very similar in style and number of results.

We believe that running these tests would provide a comprehensive test of an RDF store, with a useful impression of its strengths and weaknesses. We also believe that this will be useful for informing future development of the stores, as well as providing a large amount of data to those trying to decide the optimal store on which to run their application.

Currently, the synthetic data generation application does not provide the ability to automatically create queries, which makes the generation of this large number of queries a fairly complex manual task. Accordingly, time constraints limited the ability to test all of these factors in the test results listed in this paper. It is expected that future versions of this software will provide the ability to generate queries that perform these tasks based on the data being generated.

4 Benchmarking Software

In order to support automatic testing, a set of perl scripts was developed. These scripts read from a test set description file, which contains a series of lines describing the tests to run. Each line contains a base filename, along with a description of the test, and the number of concurrent operations to perform. It is expected that files will be found at "<filename>.<thread number>". A test with a base file name of "foo", running two concurrent operations at any one time, expects to find files at "foo.1" and "foo.2".

Each of these specified files contains a set of operations to perform, each again divided into phases: for example, 'assertion', 'queryset1', 'queryset2', etc. Operations are run in the order in which they appear in the file. The result set has no effect other than simple grouping of output results in the final report.

The operations specified in these files are one of the following:

- **bulk_assert:** assert a new data graph from a file. It is intended that no other operations will be performed upon the store while this operation is running.
- **assert:** assert new data file, potentially into an existing graph.
- **deletegraph:** delete an entire graph from the store.
- **delete:** delete statements from the store.
- **query:** perform a query upon the stored data.

It is expected that bulkassert will be run in a non-concurrent fashion – i.e. the assertion will be the only operation acting on the store when it is running (it should be noted that this is not currently enforced by the test scripts). All other operations may be run concurrently with others.

The test scripts require a "db interface" perl module to be specified to them on the command line. When the script encounters an operation, it runs the operation-specific function (passing along associated parameters) on the module. In addition to providing these operations, each db interface module is expected to provide functions 'name' and 'processes'. These provide the name of the store and the processes that the store is likely to cause significant activity in, for the purposes of recording system statistics.

Another script (again, specified on the command line) is used to provide system statistics on these specified processes. The current script is UNIX-specific, providing information on the percentages of memory and CPU used by each of the specified processes, as well as information on the amount of time the system has spent waiting on I/O. The statistics recorded by the system-stats script can be altered without requirement for altering any other scripts.

The test scripts run through the specified tests, and if the test files indicate that results should be recorded, record the time taken for each, along with system statistics while they are running. Finally, once the tests are complete, the results are examined and summaries and statistics produced in .csv format for simple examination by humans.

5 Data Generation

The generation of synthetic data is a significant challenge. Aside from the LUBM generator, the SIMILE project has also produced a simple synthetic random RDF data generator[5]. This generator produces RDF based on a number of random graph generation algorithms.

[5] http://simile.mit.edu/wiki/RDFizers

As noted, the LUBM generator produces data against quite a simple template, with a limited amount of randomisation. This results in a predictable dataset, and does not allow a great deal of configuration for the purposes of producing alternative, more interesting sets. On the other hand, while the SIMILE system produces much more random graphs, its lack of configurability makes it inappropriate for the production of datasets with certain defined characteristics.

Given the lack of an appropriate pre-existing method for generating synthetic data, it was decided to create a script to accomplish this. The approach taken for script creation was in many ways similar to that used by LUBM: the script iterates a specified number of times over a given ontology, producing RDF in the style dictated by that ontology. URIs are given names based on their type combined with an incrementing number. This means that it is not necessary to store information on all the URIs that have been generated for the purposes of URI reuse, but merely to store the current number of URIs that have been generated of that type.

The major difference between this method of generation and that provided by LUBM is the fact that the script also automatically produces the ontology against which data is generated, allowing the creation of a wide variety of RDF data graphs. This ontology currently takes the form of a tree, and is created based on a large number of factors that can be configured by the individual running the script. The ontology is not described in the output RDF, beyond identifying URI types.

Ontology creation begins by creating a class, designated as the starting point. The system then recursively attaches property types and classes to this point, down to a depth specified to the script. At this point, the script also specifies the bounds for the number of property instances that will be attached to a given instance of this class during data generation (for example: humans may have 0..15 e-mail addresses).

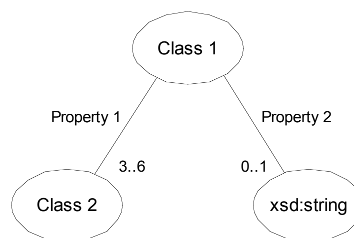

Fig. 1. Information determined by the ontology generation phase.

The shape of the ontology tree is affected by a wide variety of options. When considering attaching property types to a class, for example, the script considers the following options, applying them in order:

- **Base property types per class:** This is the number of property types to attach to each type, prior to any randomisation or other alteration. Alterations do not apply to the first level of the tree.

- **Expansion with depth:** This denotes the expansion of the number of properties to any given type with level. This allows the creation of expanding or contracting trees, as shown in figure 2. This modifier is generated by: base_properties multiplied by $depth^{expansion}$.
- **Max_multiply/divide:** The script offers a 50% chance each that the current number of properties will be multiplied or divided, by a random number between 1 and max_multiply or max_divide. This allows the creation of distributions other than the completely uniform: for example, one might wish for a usual situation of one property type attached to a class, but with some classes having more. One could then specify a base number of 1 with a max_multiply of 5, and a max_divide of 1. In future the chance of a multiply or divide will also be configurable to enhance this effect.
- **Cutoff probability:** A chance (0..1) that the tree will simply be terminated at this point.

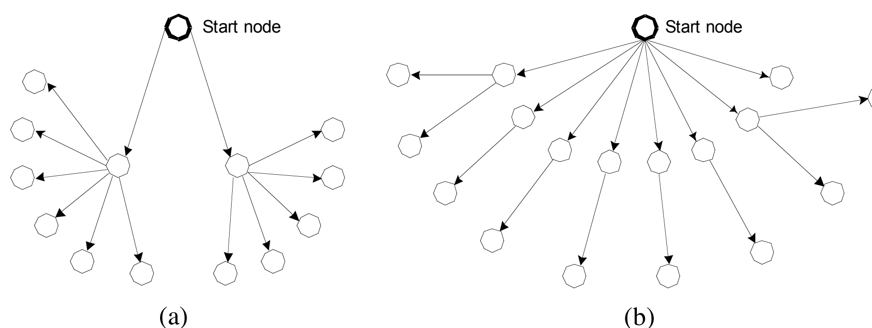

Fig. 2. (a) depicts an 'expanding' ontology tree, with the number of property types attached to a class increasing as distance from the start node increases. (b) depicts a contracting tree.

Similar alterations can be applied to factors like the number of classes attached to a property (in effect the range of the property). Other variables like the chance of a property range being a literal offer only linear randomisation, but as the script is developed further a consistent set of alterations will be offered for every one of the defined variables. This should improve both configurability and clarity.

There are a number of issues with this method of producing data that have not yet been resolved:

- Readability of the data is limited due to the use of randomly generated URIs and literals. RDF graph visualisers help, but these generally do not have support for visualising extremely large datasets.
- URIs of the same type currently have a very high degree of similarity to each other: they simply have an increment number at the end. This conceivably affords an unrealistic advantage to stores that compress data, or store repeating URI portions separately.

- In order to simplify the process of template generation, the system currently has no provision for allowing types or properties to exist at more than one place in the template.
- There is a need for automatic generation of literals that are similar to each other, and reuse of literals. This would enable easier testing of FILTER conditions. In the benchmarks described in this document, literal testing is performed against the manually created files asserted as part of the tests.
- Literal lengths are currently based on linear randomisation. In reality we might expect very different distributions.

Real world datasets exist that do not suffer from these issues. Unfortunately, there are as yet a limited number of datasets of very large size (>100 million triples), so synthetic data systems remain a convenient way to produce a dataset matched to a required data shape, and of an easily configurable size. Further, many of these issues can be eliminated or mitigated with further development.

6 Benchmark Configuration

This section documents the hardware system that the preliminary tests were run on, and the configuration steps used to optimise them for this benchmark. The queries used and structure of the test data are too large to reproduce in this document, and can be found at: http://www.ecs.soton.ac.uk/~ao/rdfbenchmarking. The test data consisted of approximately 235 million triples, structured in a relatively sparse manner with 112 million unique nodes. Every node had a type, and 10% of nodes were given a label. The range of 10% of properties was a literal, and a large proportion of the number of unique entities can be attributed to the fact that each of these literals was unique. The structure of the template tree is contracting, as shown in figure 2b.

It became clear over the course of asserting data that the system statistics monitoring system was causing an unacceptable degradation in performance, despite being located on a separate drive to the DBMS files. We believe that this may be due to both disks residing on the same disk controller. Due to time constraints, it proved impossible to resolve this issue, so to provide accurate results, system statistics were not monitored during these tests. This issue will be resolved in future iterations. Finally, it should be noted that performance of deletions was not studied in these tests.

5.1 Assumptions

Each of the systems tested were configured with the following assumptions in mind, based on the contents of the test set:

1. Queries may be of any form.
2. No reasoning is required.
3. Bulk asserts are allowed to use all system resources, so multi-threaded file loading can be used where appropriate.

Due to the pre-release nature of both the BigOWLIM and AllegroGraph installs, we chose to perform the initial bulk assert for each system by hand, to allow for steps that would have been difficult to perform using a scripted environment.

5.2 Test Environment

The configuration for the tests was a single machine, as follows:

- 2x 1.8GHz AMD Opteron (single core, 1MB L1 cache each)
- 8GB RAM
- 1x Seagate Barracuda SATA HD (250GB, 7200rpm, 8ms average seek)
- 1x Seagate Barracuda SATA HD (500GB, 7200rpm, 8ms average seek)
- Red Hat Enterprise Linux 4.1.2-14, kernel version 2.6.18-53

For the purposes of the test, the database files were located on disk one, and the script and data files were placed on disk two. The script also writes its data to disk two. This minimises interference with the disk that the stores are interacting with, to enhance test accuracy.

5.3 Store Configuration

This section details the configuration of the individual stores. Each store was queried through its HTTP interface, while assertion configuration varied depending on the store.

AllegroGraph 3:

AllegroGraph 3 was configured using a script provided by the authors. This allows data to be loaded using more than one simultaneous thread, significantly improving bulk assert performance. This process has a tradeoff: it creates a federated (rather than single, homogenous) store, which is slightly slower to query. Since the test machine has two processors, two assertion threads were used. Default indexes are applied by the script after assertion.

BigOWLIM 3:

BigOWLIM 3 was configured as a Sesame SAIL. Sesame exists in an Apache Tomcat container, which was allowed to use up to 6.5 GB RAM. The following settings were modified:

Number of cached pages: 15,000
Entity index size: 40,000,000
Rule set: empty (no reasoning)

7 Results

This section describes the results obtained when the benchmarking system was applied to the test stores. It should be noted that both of these stores are pre-release versions, and flaws experienced in these tests may not be apparent in release versions. Due to the nature of the tests, we have made no study of BigOWLIM's notably strong reasoning capabilities.

7.1 Assertion

The test results show interesting data regarding assertion times. All the assertion-related tests described in section 3 were applied. None of the asserted files had links to URIs in any of the other files, but they were asserted into the same model:

- One file totalling approximately 235 million triples (bulk)
- One large file totalling around 550 thousand triples (largeadd)
- A set of 3 small files totalling around 90 thousand triples (smalladd)
- 3 sets of 3-4 files, each set totalling around 90 thousand triples, asserted concurrently (concurrentadd)
- A set of 3 small files, totalling around 90 thousand triples, asserted concurrently while queries were operating on the store. (queryadd)

Table 1. Assertion data, comparing BigOWLIM and AllegroGraph.

Test	BigOWLIM time(s)	AllegroGraph time(s)
bulk	14086.0	15777.6
largeadd	187.5	42.9
smalladd	8.57	28.0
concurrentadd (mean)	15.0	56.3
queryadd	68.85	47.2

These results show an advantage for BigOWLIM on the initial bulk upload, and also when subsequently asserting small files. AllegroGraph, on the other hand, is faster at performing the larger subsequent addition, and when performing additions while under query load. Both stores degraded in performance gracefully during concurrent

assertion. Finally, it should be noted that there was an additional cost of approximately five seconds to index all the new triples in AllegroGraph after they had been asserted.

7.2 Querying

Test results for querying are more complex to analyse. Due to the number of results retrieved from these tests, we have chosen to describe areas of interest with regard to each store, rather than the entire result set. Raw result sets are available at http://www.ecs.soton.ac.uk/~ao/rdfbenchmarking. It should be noted that these tests are designed to be challenging for the stores, and there is no expectation that they will complete them all.

Overall query performance appeared to be better in AllegroGraph: it was capable of completing all tasks, while BigOWLIM exceeded the cutoff point of twenty minutes on several occasions. On occasions when the cutoff point was exceeded, the server was restarted and warmed up again for the next result set, to prevent queries still being executed in the store from interfering with other results.

Both stores were very capable of executing simple one triple pattern queries, with AllegroGraph completing the set in 3.9 seconds against BigOWLIM's 12.76. BigOWLIM's slower result was largely due to a relatively slow retrieval of queries that specified only an object. Retrieval of larger result sets (several hundred and tens of thousands) resulted in closer results, with AllegroGraph retrieving them in approximately one half to one third the time of BigOWLIM.

Both stores exhibited excellent behaviour when attempting to retrieve previously cached queries, as well as when using LIMITs to retrieve small portions of a large dataset. They were also capable of quickly returning when a query had no result.

The stores were capable of answering most queries containing two to three triple patterns in a short period of time, but flaws were exhibited in the query optimisation of both, with some queries that were answered in less than a second by one store taking several minutes in the other.

Neither store was capable of effectively optimising queries that used FILTERs rather than UNIONs to bind multiple URIs to a variable, as seen in figure 3. AllegroGraph was capable of completing these queries, taking 4-15 minutes to do so, while BigOWLIM timed out when attempting to perform them. It should be noted, however, that BigOWLIM performed the UNION queries in a more reliably short period of time.

```
SELECT ?y ?z WHERE
{ ?x ?y ?z .
 FILTER (?x = <example:1> || ?x = <example:2>) . }"
```

Fig. 3. SPARQL query demonstrating a FILTER disjunction equivalent to a UNION statement. Prefixes are omitted in all queries for brevity.

The use of FILTER for the more conventional means of restricting over a regular expression or numeric range gave comparable results for both stores, except in situations where a very large proportion of the data set had to be iterated over. In these situations, AllegroGraph performed exceptionally well: retrieving a simple REGEX over the entire population of rdfs:label in less than a second, and an integer filter over the entire store in just over 60. BigOWLIM took 500 seconds for the former, and was unable to answer the latter before cutoff.

The tests on multi-threading exhibited better scaling on BigOWLIM. Tests were performed by measuring the response times of sets of queries of similar style (the multi-triple bulk joining test described in section 3), running concurrently with secondary threads containing various styles of query. These secondary threads were not timed, but merely used to provide load on the system. Table 2 shows the results of these tests.

Table 2. Multi-threading tests

Test	BigOWLIM time(s)	AllegroGraph time(s)
1 thread	128.0	44.2
2 threads	76.1	56.0
7 threads	149.3	2365.3

The reasons for the dramatic decline in AllegroGraph's performance at 7 threads are not clear. The test was repeated on the store to ensure accuracy. These results bear further investigation to determine the cause, and whether the style of background queries used by the 7 thread test had an influence on results.

Finally, we examine the test for complex queries, where the systems process queries with over three graph patterns, potentially requiring the processing of a large number of results internally before arriving at a result. An example of this is shown in figure 4.

```
SELECT ?x WHERE {?x ?y <example:subject1> .
?x ?y2 ?z .  ?z ?y3 ?z2 .
?z2 ?y4 <example:object2> .}
```

Fig. 4. Example of a complex query with few results.

BigOWLIM was unable to answer any of these queries before the cutoff. AllegroGraph was able to arrive at the answers in approximately 300 seconds, with two out of three of the queries taking less than a second.

8 Conclusions and Future Work

In this paper we have described systems for measuring the performance of RDF stores in a detailed, diagnostic manner with a large variety of test cases. We have shown results that could not have been achieved using any other benchmark commonly in use for the purposes of benchmarking RDF stores, including tests for concurrent access and assertion. This work will help to develop the performance of upcoming RDF stores by identifying weak areas, and provide detailed information to application developers.

Although these tests bear some similarity to use case-based benchmarks such as LUBM, it is not expected to compete with or replace them: a large amount of data is generated by these tests, and the quantity of information requires significant interpretation to discover what is important for the purposes of a given individual. This does not offer the relative simplicity of comparison afforded by systems such as LUBM or TPC-C.

We are planning a number of improvements to this work, particularly in the area of automatic data generation. The most important addition to the current system is automatic query generation. It is necessary to generate new queries whenever the template ontology is altered, and this is currently a labour-intensive manual process. We propose to automatically produce interesting queries based on the specifications of configuration files or plugins.

While the existing configuration options for data generation are extensive, it would be useful for the purposes of generating a wide variety of queries to be able to specify different settings for different subgraphs. As noted in section 4, there are several feasible improvements to the realism of the data. Uniformity of URIs can be reduced by passing URIs through a function that performs alterations on them based on their content. Support for the positioning of properties/types at more than one point in the template ontology would enable greater data variety and realism.

Currently, it is challenging to get more than an overall impression of what the template ontology will look like prior to data generation. We intend to develop a graphical visualisation tool to aid the user in selecting appropriate parameters for the generated data.

Finally, when a user has specific requirements regarding the use of certain ontologies, we propose to allow them to specify an ontology to use as the template, and generate data and queries from that. This would enable users with a dataset that they expect to

expand to test the scalability of their infrastructure, with a dataset extremely similar to what they might expect to be using.

References

1. Guo, Y., Pan, Z., Heflin, J.: LUBM: A benchmark for OWL knowledge base systems. Web Semantics: Science, Services and Agents on the World Wide Web **3** (2005) 158--182
2. Weithoner, M.L.T., Liebig, T., Luther, M., Bohm, S.: What's Wrong with OWL Benchmarks? Proc. of the Second Int. Workshop on Scalable Semantic Web Knowledge Base Systems (SSWS 2006) 101-114
3. Guo, Y., Pan, Z., Heflin, J.: An Evaluation of Knowledge Base Systems for Large OWL Datasets. Proc. of the Third Int. Semantic Web Conf.(ISWC 2004), LNCS. Springer Verlag (2004)
4. Liu, B., Hu, B.: An Evaluation of RDF Storage Systems for Large Data Applications. Proceedings of the First International Conference on Semantics, Knowledge and Grid (2005)
5. Rohloff, K., Dean, M., Emmons, I., Ryder, D., Sumner, J.: An Evaluation of Triple-Store Technologies for Large Data Stores. LECTURE NOTES IN COMPUTER SCIENCE **4806** (2007) 1105
6. Council, T.P.P.: TPC Benchmark C, Standard Specification, Revision 5.9. (2001)
7. DeWitt, D.J.: The Wisconsin Benchmark: Past, Present, and Future. The Benchmark Handbook for Database and Transaction Processing Systems **1** (1991)
8. Bitton, D., DeWitt, D.J., Turbyfill, C., Dept, C.S., of Wisconsin--Madison, U.: Benchmarking Database Systems: A Systematic Approach. Computer Sciences Dept., University of Wisconsin-Madison (1983)
9. Cattell, R.: The Engineering Database Benchmark. The Benchmark Handbook for Database and Transaction Processing Systems (1991)
10. Prud'hommeaux, E., Seaborne, A.: SPARQL Query Language for RDF. W3C Candidate Recommendation. World Wide Web Consortium, April (2006)
11. Guo, Y., Qasem, A., Pan, Z., Heflin, J.: A Requirements Driven Framework for Benchmarking Semantic Web Knowledge Base Systems. IEEE TRANSACTIONS ON KNOWLEDGE AND DATA ENGINEERING (2007) 297-309
12. Broekstra, J., Kampman, A.: Inferencing and Truth Maintenance in RDF Schema. Proceedings of the First International Workshop on Practical and Scalable Semantic Systems (2003)