Report on Summer Internship Work For the AKT Project: Benchmarking RDF Triplestores

Michael Streatfield, Hugh Glaser
{mrs204,hg}@ecs.soton.ac.uk

1. Introduction

This report details the work done on the benchmarking of RDF triplestores over the summer. It talks about the structure of the tests performed and the scripts produced which were used to run the tests. It is intended to be useful in both assessing the results produced and for anyone who wishes to take the work on further in the future. This work was done in conjunction with Dr Jeremy Frey and Kieron Taylor of the University of Southampton's Chemistry department. I have gone into as much detail as possible whilst trying to keep it relevant, so that the scripts I have written can be understood in context.

It should be noted that this report does include the results from all of the tests performed (~12 million triples in 3store's case), although it might be better not to publish those until there are similar results for all of the stores.

2. Testing Environment

The environment in which the tests were run was a machine in from the ECS undergraduate computing labs (specifically the one named 'zombie'). It is an "Intel(R) Pentium(R) 4 CPU 3.00GHz" with 905400 kB of RAM (1Gb). The machine was given a clean install of Debian Linux 3.1 (www.debian.org) , which was the latest release at the time. The reasoning behind this was that the tests would be more reliable if they were performed on as clean a machine as possible, and as far as possible only the packages required for the benchmarks to run were installed.

Because it may be useful in understanding the scripts, the file system was structured as follows:

/home/new.benchmark: this was where the benchmarking scripts were kept – each under its own directory. In each directory there were scripts to run the benchmarks, in the case of Kowari, Jena and Sesame there were also the libraries themselves since these needed to be accessible as part of a classpath for the scripts. When results were generated, these were placed in a **results** directory under each triplestore's directory.

/home/storage: this was where the MySQL database was kept.

/home/synthdata: this was where the generated data set was stored (see below). Each generated molecule data file was given an MD5 hash of its unique INChI value as a filename. These files were placed in directories dependent on the filename – thus `038e3f8c94be892a68ee09708a2369e2.rdf` was placed in the /home/synthdata/0/3/8 directory, for example. This was useful in defining the discrete elements used in the tests themselves. This structure was borrowed from some earlier tests performed by Kieron.

In the actual event, the data was split into another set of directories labeled of the form X.1 (where X was the original set's 0-f top level directory.) Half of the files from the original set were placed in this new structure. This was in order to provide a sensible number of files in each directory for testing, since time was running out and it was a useful way to generate more results using less data.

3. Data set

Kieron provided a selection of RDF files created from data on chemical molecules in order to run benchmarks on, however this data was sparse and in any case not available to the general public. Thus a consistent data set was generated with ontology which the example data used, using the Redland library's Perl bindings (version 1.0.2.1). The generated RDF data is based on numerical analysis of the original data, since it was considered useful to base the benchmark on existing real world data. The actual makeup of the generated data is given in Appendix A. It is worth noting that any strings required were produced using random alphabetical and numerical characters, under the assumption that the size of data (i.e. the length of the string) made more difference than the actual values of the data itself. However, having a diverse set of data made it possible to run specific queries. The script which was used to generate the data is given with this report (`synthesizemolecule.pl`).

The data was placed in the file system as described above. The elements used in the tests were the second level directories (so /home/synthdata/X/Y where X and Y are one of 0-f and X could also be X.1) and these elements were tested in order (so 0/0 was benchmarked first, then 0/1 then 0/2 and so on.) Each of these directories contained ~800 files, with each containing 49-51 triples. There were ~400,000 generated molecules in total, given ~20 million triples of data altogether. The tests themselves were only run up to ~2.3 million due to time constraints, however.

3.1 Meta data

In order to aid the construction of scripts, meta data was produced for the data set (created using the `generateinfo.pl` script). This was stored in the `datainfo` file in the top level of the synthesized data set directory. This took the form of comma separated values, in the following order:

> **Directory:** the top level directory of the element to be tested. This is the "key" for the meta data and is used in the scripts to locate the directory to benchmark. The scripts perform benchmarks on the elements in the order of the directories in the meta data file.
>
> **Total Files:** the total number of RDF files in the element (i.e. the number of RDF files in the element's subdirectories). This was originally intended to be used in checking the integrity of results, but in practice was not.
>
> **Total Triples:** generated by iterating over all the RDF files in the element and using Redland's "`rdfproc`" utility to count the "raw" number of triples in each. This value is the total number of triples in the element.
>
> **Random File Names:** two files in this element, chosen at random. These are

guaranteed to be different, and are used in queries. This means that a different molecule is used each time in the first two queries. This was originally intended to provide some variety in the queries run, although I am unsure as to whether this makes a difference in terms of the results. However, by including the same files in each store's queries, it does make the benchmark fair on each.

4. General Structure of Benchmarking Tests

The benchmarks were developed in order to measure the triplestores' efficiency in both asserting and querying data, as well as how efficient their use of disk space was. Each store was set up to run on a locally stored repository, using the most efficient storage available to them. With the exception of Kowari, this was a MySQL database. The MySQL database version used was 4.1.12, the most recently available at the time.

Each benchmark was given a clean MySQL to use. This was achieved by dropping any databases but the one currently being used at any given time. Each store was granted `ALL PRIVILEGES` on its respectively named database, and each store had a database name and password related to the store name (3store used 'rdf' and 'rdf' as it was designed to, the others used similar ones, for example Jena was 'jena' and 'jena').

4.1 Assertion

Efficiency in importing data was measured by asserting each element of the data set into the store using the most local method possible. By this I mean that the scripts tried to use the least possible number of layers to assert data. This meant that the results were based on how fast the stores were able to parse, organize and write the data to disk rather than how quickly they dealt with the protocols which might be used to run the stores remotely. This meant using the Java based stores APIs directly in a Java program, and command line applications where available. This was also used as a rough method for guaging each store's ease of use.

The time taken was measured as how fast it took the stores to assert all the RDF files in a directory – where possible this involved fetching a list of the filenames before beginning the time measurement, in order that the benchmarks were not affected by disk access and file system searching.

4.2 Disk Usage

Disk usage was measured using the "`du -s`" Unix command in the directory that the data was stored in. This measures the disk used by files in that location by blocks on the disk. It was realized later that this might be unreliable, but luckily the file system used seems to use a block size of 1kb. However, this should be taken into account given any discrepancies in the results.

Another problem with using this method is that the data is not necessarily stored in the same place for each store – for example, MySQL tables may be stored in both a subdirectory of the data directory within the MySQL structure, or they could be stored at the top level data directory in the MySQL structure dependent on the table type. This means that some of the benchmarks' data is produced in

terms of subdirectories, and some as top-level directories. Thus I would consider measurements of this data to be unreliable.

4.3 Queries

Four query templates were chosen to benchmark, based on queries which Kieron suggested would be useful in the research he is performing.

The first of these selects the URI of a single molecule given a known predicate and object, in this case being the INChI value of the molecule as lifted from the meta data produced as described above. This has a real world application as the first step in locating information about a specific molecule given some knowledge of the molecule being searched for.

The second query selects information about a given molecule with a known URI (again lifted from the meta data), listing all of the predicates and objects attached to that node. This again has applications in discovering information about a given specific molecule.

The third query is more complex than the first two, and is designed to harvest a set of data on which more processing can be performed. Specifically, it selects information about the original files that any "Structure" properties considered to have "Good" quality came from, listing all predicates and objects attached to that node.

The fourth query is an extension of the third designed to test numerical comparison using the query language. It selects the same data as the previous query along with the molecule that the property is related to and the relative molecular mass (RMM), given that the RMM is less than 500. This means that roughly 2/3 of the data in the store should be selected by this query (given the values decided in creating the data set, see appendix A for details).

Not all query languages used supported the fourth query, but where possible the queries were kept to the same format. The actual queries used by language and store are listed in appendix B.

Each of the queries is run ten times consecutively, and for some of the stores the number of results returned was recorded (about halfway through the tests I realized this would be a good way to test the accuracy of the results).

4.4 Results

A log file was kept of any output considered interesting, although the output from each of the queries had to be cut from this since it resulted in very large files (of the order of gigabytes). The results themselves were kept in a directory underneath each store's scripts directory called 'results'. Data was kept for individual elements under the assert/ and query.X/ (where X is one of 0, 1, 2 or 3 dependent on the relevant query) directories with a number of the form XXXX (where X is an integer) indicating which element of the test it was. This included data which was not essential to note but would be useful in explaining discrepancies in the results, perhaps.

Data which could be used in analysis was recorded in `assert.data.csv` and `query.X.csv` files

in the top level of the results directory, as comma separated values. The data recorded from store to store varied and is described in the next section.

In case details about the working environment were useful, the outputs of "`cat /proc/meminfo`", "`cat /proc/cpuinfo`" and "`ps aux`" were recorded before each test was run.

4.5 Fair test

The benchmarks were run with the idea of a fair test in mind. To this end, the machine was rebooted before each benchmark was run, in order to ensure a similar working environment in terms of memory usage and running processes. The specific scripts were also written to use similar capabilities in terms of being tested – such that the same aspect was being tested in each and thus they could be compared.

Triplestore Specifics

All of the specific triplestore scripts have a `setupbenchmark.sh` file with them. This was written and used to clean the environment in which the stores would be used before running the benchmark itself. They are used to save any previous set of results and clean out any persistent database.

3store (http://www.aktors.org/technologies/3store)

Version used: 3store 2.2.22 (slightly modified).

This is a Perl script which calls the 3store command line utilities to perform the tasks of asserting and querying data. The Perl Time::HiRes module was used to measure the time taken to complete tasks, using the `gettimeofday` subroutine, which was chosen since it was the most accurate timer available.

The model was specified directly on assertion and query, in order to restrict the results set to only those in that model (since querying the store in general returned some internal 3store implementation triples as well). Initially `tstore_rebuild_taxonomy` and `tstore_optimize` were used after each assertion run. After discussion with Steve Harris these were removed from the script. Optimize was not required after every run, and the rebuild taxonomy command was being used to work around an inferencing bug in 3store, which was fixed by removing a specific facet of a query in code (the specifics are included with the README file in the 3store script directory).

The query language used was RDQL and the specific queries can be found in appendix B.

Jena (http://jena.sourceforge.net)

Version Used: Jena 2.2, with MySQL Java connector 3.0.17-ga (since 3.1.x was incompatible).

This was written as a Java program utilizing the Jena API directly. A remote model interface is setup

to the MySQL store, and methods invoked from there to import and query data. The time taken for each element of the test is measured using Java's `System.currentTimeMillis` method, which is less precise than that used by the Perl based scripts, but still precise enough to measure the longer periods of time taken for complex actions. The program is deliberately written to use only static methods, since this was the least complex option.

Initially, I wrote some scripts to use the Jena command line utilities, but it was apparent that the overhead generated by the Java virtual machine in running these each time was the biggest factor, so it was decided to move it all into a specifically written Java program which accessed the API directly. I also experimented with using the Joseki server written by the same developers, however this is intended mostly for publishing RDF models on the web, rather than working directly with them, and merely imposed an unneeded layer on top of what was required by the general test structure.

The query language used was RDQL and the specific queries can be found in appendix B.

Sesame (http://www.openrdf.org)

Version used: Sesame 1.2.1 with MySQL Java connector 3.1.10.

This followed the same template as the Jena benchmark, using the same technique for measuring time taken. The benchmark sets up a remote model which connects to the MySQL database through its "sails", which are layered interfaces on the model. This benchmark specifically uses a synchronization sail on top of the MySQL sail, as this was recommended in the documentation..

The query language used was SerQL and the specific queries can be found in appendix B.

Redland (http://librdf.org)

Version used: Redland 1.0.2.

This followed the same template as the 3store benchmark script, except using the Redland "`rdfproc`" utility. The query language used was RDQL and the specific queries can be found in appendix B.

It should be noted that Redland did not appear to return the number of results from queries which the other stores in which this was measured did. The method I used to count these was to count the number of lines which an rdfproc query returned, and this may have been affected if some of the queries were not separated by newline characters. Also, this may have been caused by the query engine aggregating some of the returned results (some query engines return the same result multiple times if copies are given, whereas others tend to aggregate them into a single result).

Kowari (http://www.kowari.org)

Version used: Kowari 1.1.0-pre2.

This followed the same template as the 3store and Redland benchmarks, using a similar Perl script. The

server itself is set running with the `startserver.sh` script, and is the equivalent of the MySQL server in the other scripts. The server is accessed via Java command line applications in the Kowari jar files. The iTQL application only seems to accept scripts which are given in a file on disk. Thus to load triples into the store the script writes an iTQL script using "load" commands for each file, and calls the iTQL command line application with this file as an argument. It is important to note that the script turns "autocommit" off, loads all of the files, then commits them as a single block, since this significantly improves the performance of the server. Queries are performed in a similar fashion, the only thing which changes is the command used.

The query language used was iTQL, although it did not have the features available to run the fourth query.

Results

The results are included with this report in in Open Office documents and comma separated value files. The specific results for the stores are available in the "results" subdirectories in the specific script directories, and the exact values they store can be gleaned from the scripts. The results mentioned here are specifically designed for comparing the stores.

assert.results.sxc: This contains the results for the stores for the assertion times, in seconds. It includes all the results collected, but the graphs given are only up to 2.3 million triples or so, since this is the largest amount of data collected for all of the stores.

query.0.csv and query.1.csv: These contain the results for the first two queries. Since they tend to be very fast across all of the stores, it was not considered worth plotting a graph of the activity – in the case of the Java based APIs (Jena and Sesame), the timing mechanism was too imprecise to measure the time taken accuracy anyway. They contain the minimum and median times out of the 10 taken for each query for each of the stores (in seconds).

query.2.sxc and query.3.sxc: These contain the same information as queries 0 and 1, but provide a graph plot to go with them.

Appendix A: Detailed description of the generated molecule data

Simple numerical analysis of the original molecule data was used to create a template for the synthetic data. Any reference to a random string refers to a string of randomized uppercase letters and digits, created using the String::Random Perl module. This is based on the assumption that the length of the string is the important factor, rather than the content. If the content of a string is important in terms of querying, then it is dealt with on an individual basis. The number in brackets next to the predicate is its cardinality, although this is not explicitly stated in the RDF schema (the included `chemschema.rdf`). This data is based on a subset of that ontology.

Namespaces:

```
"rdf" for "http://www.w3.org/1999/02/22-rdf-syntax-ns#"
"dc" for "http://purl.org/dc/elements/1.1/"
"dcterms" for "http://purl.org/dc/terms/"
"ch" for "http://someuri.somewhere/rdf/chemschema.rdfs#"
"unit" for "http://someuri.somewhere/rdf/units.rdfs#"
```

The URL "someuri.somewhere" was used instead of the specific chemistry department server that was originally in the schema, since the internal networking details of the chemistry department need not be exposed to the world at large.

This next part is a little haphazard, but consists of a subject, then beneath that a list of predicates and an explanation of what the objects they have are. The length and standard deviation of strings are based on the mean and standard deviation of the strings in Kieron's actual data. An arrow (**->**) indicates a predicate of a node which is described under the main node.

Unique URI of the form <file:/path/to/file.rdf>

- **ch:has-inchi(1)**: A random string, of mean length 102, standard deviation 43.
- **ch:has-empirical-formula(1)**: A random string of mean length 9, SD 2.
- **ch:has-cas(0..1)**: present 3/4 of the time. A random string mean length 9, SD 1.
- **ch:has-name(0..1):** present 1/200 of the time. A random string mean length 15, SD 9.
- **ch:has-stereocentres(1):** an integer value. The data is randomized, and biased as follows:
  0: 43/50
  1: 4/50
  2: 1.5/50
  3-17: 1.5/50
- **ch:has-rmm(1)**: Not calculated as carefully as the stereocentres, but a random number (with a precision of 7 significant figures) between 100 and 700. This is likely to be used in queries (searching for values <500 - which means 2/3 of the data is selected on average). I didn't consider it a good use of time to come up with a more true-to-life random value for this.
- **rdf:type(1)** – chosen at random (evenly distributed) from <ch:OrganicMolecule>, <ch:InorganicMolecule>, <ch:OrganometallicMolecule> and <ch:OrganicSalt>.
- **ch:has-property(3)**: URIs of nodes with further data, one each of <ch:Structure>, <ch:MeltingPoint> and <ch:Solubility>, as described below.

All three types of property (Structure, MeltingPoint and Solubility) have the following predicates:

Property URI, of the form <http://someuri.somewhere/resolver/property/n> where n is a number above 1000, padded with 0s if it is less than 10,000. This is such that the numbers roughly match the patterns in the original data. The property numbers count up sequentially from 1000.

- **ch:has-source(1)**: one of (evenly distributed) <ch:NCI> <ch:PhysProp> <ch:Detherm> <ch:MerckIndex> <ch:CCDC>. This does not match the example data, but I felt that since the ontology calls for it to be present then it should be.
- **dcterms:created(1)**: Random ISO 8601 Date of the format "YYYY-MM-DDTHH:MM:SSZ", created using the Data::Random Perl module.
- **dcterms:provenance(1)**: In theory 0..1, but in practice it's almost always present, so we'll just assume that all properties have provenance. This is a blank node.
- **dcterms:provenance->rdf:type(1)**: this is the only node beneath the provenance blank node - I decided that it would take too long to create accurate provenance data, and it need not necessarily be present to perform benchmarks anyway. Consists of one of <ch:Laboratory> or <ch:Calculated>, in the ratio 3:50.
- **ch:of-quality(1)**: In theory 0..1, but it may be useful in queries. Almost all of Kieron's data had this property present. This is a blank node.
- **ch:of-quality->rdf:type(1)**: One of <ch:Good> <ch:Bad> <ch:Suspect> in the proportions 3800:800:1.
- **ch:of-quality->dc:creator(1)**: a random name chosen from a group of 20 which are generated using the Perl Data::Faker::Name module from cpan.org. The rationale being that one person could have created data relating to many molecules. Names:

  Arthur Romaguera V,
  Nasir Tromp,
  Carmella Green,
  Aaliyah Botsford,
  Earl Nitzsche,
  Allison Roberts,
  Meaghan Mitchell,
  Evelyn Reynolds DVM,
  Haylee Monahan,
  Aryanna McCullough,
  Rod Herman,
  Yasmeen Ondricka,
  Odell Reichel,
  Alessia Ledner,
  Dusty Braun,
  Jay Wolf PhD,
  Cecelia Koch,
  Bobby Streich,
  Abbigail Reilly,
  Baylee Buckridge

- **ch:of-quality->dcterms:created(1)**: Random ISO 8601 Date of the format "YYYY-MM-DDTHH:MM:SSZ", created using the Data::Random Perl module.

The Structure property has the following in addition to the above:

- **rdf:type(1)**: <ch:Structure>
- **ch:has-file(1)**: has the same value as the property URI, except with "file" instead of "property".
- **ch:has-file->dcterms:created(1):** A random date, as above.
- **ch:has-file->ch:filename(1)**: A random string mean length 74, sd 0.
- **ch:has-file->dc:type(1):** <ch:Mol>

The MeltingPoint property has the following in addition to the common set (there is a 50:50 chance of the melting point being in Celsius or Kelvin. The values aren't intended to match real life, but were intended to potentially be used in queries):

- **rdf:type(1)**: <ch:MeltingPoint>
- **ch:has-quantity(1)**: Blank node.
- **ch:has-quantity->ch:has-value(1)**: A random integer between 0 and 1400 in Kelvin, or -300 to 1100 if Celsius.
- **ch:has-quantity->unit:has-unit(1)**: Blank node.
- **ch:has-quantity->unit:has-unit->rdf:type(1)**: <ch:Celsius> or <ch:Kelvin>
- **ch:has-quantity->unit:has-unit->unit:power-of(1)**: "1"

The Solubility property has the following in addition to the common set:

- **rdf:type(1)**: <ch:Solubility>
- **ch:has-quantity(1)**: Blank node.
- **ch:has-quantity->ch:has-value(1)**: A random integer between 0 and 1,000,000.
- **ch:has-quantity->unit:has-unit(1)**: Blank node.
- **ch:has-quantity->unit:has-unit->rdf:type(1)**: <ch:Gram>

Appendix B: Specific Queries

The queries described here are in the order presented in section 4.3, labelled 1, 2, 3 and 4. In the code they are referred to as 0, 1, 2 and 3, although this is because that is how they are referenced in the arrays when used. Where RANDOM appears in these queries, it is replaced by regular expression with the random files from the data set's meta data file.

3store

Language used: custom RDQL implementation.

Notes: The $model variable is from the Perl scripts and references the model name, which is required in the triples used by 3store version 2's RDQL engine to reference a specific model.

Queries:

```
1)   SELECT ?x WHERE (?x <ch:has-inchi> "RANDOM" $model) USING ch FOR
  <http://someuri.somewhere/chemschema.rdf#>

2)   SELECT ?pred ?obj WHERE (<RANDOM> ?pred ?obj $model)

3)   SELECT ?pred ?obj WHERE (?property <rdf:type> <ch:Structure>
  $model) (?property <ch:of-quality> ?quality $model) (?quality
  <rdf:type> <ch:Good> $model) (?property <ch:has-file> ?file $model)
  (?file ?pred ?obj $model) USING ch FOR
  <http://someuri.somewhere/chemschema.rdf#>

4)   SELECT ?molecule ?mass ?pred ?obj WHERE (?molecule <ch:has-rmm>
  ?mass $model) (?molecule <ch:has-property> ?property $model)
  (?property <rdf:type> <ch:Structure> $model) (?property <ch:of-
  quality> ?quality $model) (?quality <rdf:type> <ch:Good> $model)
  (?property <ch:has-file> ?file $model) (?file ?pred ?obj $model)
  AND ?mass<500 USING ch FOR <http://someuri.somewhere/chemschema.rdf#>
```

Jena, Redland

Language used: RDQL.

Queries:

```
1)   SELECT ?x WHERE (?x <ch:has-inchi> "RANDOM") USING ch FOR
  <http://someuri.somewhere/chemschema.rdf#>

2)   SELECT ?pred ?obj WHERE (<RANDOM> ?pred ?obj)

3)   SELECT ?pred ?obj WHERE (?property <rdf:type> <ch:Structure>)
  (?property <ch:of-quality> ?quality) (?quality <rdf:type>
  <ch:Good>) (?property <ch:has-file> ?file) (?file ?pred ?obj) USING
  ch FOR <http://someuri.somewhere/chemschema.rdf#>
```

```
4)  SELECT ?pred ?obj WHERE (?property <rdf:type> <ch:Structure>)
  (?property  <ch:of-quality>  ?quality)  (?quality  <rdf:type>
  <ch:Good>) (?property <ch:has-file> ?file) (?file ?pred ?obj) USING
  ch FOR <http://someuri.somewhere/chemschema.rdf#>
```

Sesame

Language used: SerQL

Queries:

```
1)  SELECT x FROM {x} ch:has-inchi {"RANDOM"} USING NAMESPACE ch =
  <http://someuri.somewhere/chemschema.rdf#>

2)  SELECT pred, obj FROM {<RANDOM>} pred {obj}

3)  SELECT  pred,  obj  FROM  {property}  rdf:type  {ch:Structure},
  {property} ch:of-quality {quality} rdf:type {ch:Good}, {property}
  ch:has-file  {}  pred  {obj}  USING  NAMESPACE  ch  =
  <http://someuri.somewhere/chemschema.rdf#>

4)  SELECT molecule, mass, pred, obj FROM {molecule} ch:has-rmm
  {mass}, {molecule} ch:has-property {property}, {property} rdf:type
  {ch:Structure},  {property}  ch:of-quality  {quality}  rdf:type
  {ch:Good}, {property} ch:has-file {} pred {obj} WHERE mass <
  \"500\"^^xsd:float  USING  NAMESPACE  ch  =
  <http://someuri.somewhere/chemschema.rdf#>
```

Kowari

Language used: iTQL

Notes: $model is the Perl variable holding the name of the model to be queried. These queries are run inside script files, where the namespaces are declared first using the "alias" command like so:

```
alias <http://www.w3.org/1999/02/22-rdf-syntax-ns#> as rdf ;
alias <http://someuri.somewhere/chemschema.rdf#> as ch ;
```

Queries:

```
1)  select  $x  from  <rmi://localhost/server1#'.$model.'>  where  $x
  <ch:has-inchi> \'RANDOM\' ;

2)  select  $pred  $obj  from  <rmi://localhost/server1#'.$model.'>
  where <RANDOM> $pred $obj ;

3)  select  $pred  $obj  from  <rmi://localhost/server1#'.$model.'>
  where  $property  <rdf:type>  <ch:Structure>  and  $property  <ch:of-
```

quality> $quality and $quality <rdf:type> <ch:Good> and $property <ch:has-file> $file and $file $pred $obj ;

4) *Kowari's iTQL language does not have the features required to implement a query of this form. Specifically, it does not have the capability to run a numerical comparison on the output.*