kTBS Bench Documentation

Release 0.1

Vincent

April 30, 2014

Contents:

CHAPTER 1

Reports

1.1 Benchmarking insert into triple stores

1.1.1 The kTBS structure

The kTBS hierarchy is as follows:

- a RESTful API speaks over HTTP to:
- the kTBS itself, it uses:
- rdflib for managing the RDF parts, which in turn uses:
- stores, such as Sleepycat, Virtuoso, PostgreSQL (over rdflib-sqlalchemy) and much more, for storing data.

Our first concern was about the rdflib and store stages. So we let alone the kTBS and its HTTP layer in the first place.

1.1.2 In search of a good bench

Before implementing anything, we searched for existing benchmarks on triple-stores and anything SPARQL related.

We found some interesting work:

- SP2Bench can make consistent set of an arbitrary number of triples, defines some queries to test different scenari.
- BSBM: benchmarks in a e-commerce scenario.
- DBPSB: SPARQL benchmark using DBpedia.
- LUBM: inference and reasoning capabilities of RDF engines.

We settled for SP2Bench because we can use it to make graphs of different sizes and it already defines interesting queries to benchmark the store that handles the graphs.

1.1.3 Benchmarking triples insertion

The first task we did was to benchmark different stores for the insertion of triples.

What we have: stores and triples

The stores we used are:

- Sleepycat

- Virtuoso
- Jena/Fuseki
- 4store
- PostgreSQL (with rdflib-sqlalchemy and rdflib-postgresql)
- SQLite (with rdflib-sqlalchemy and rdflib-sqlite)
- MySQL

We used a set of approximetly 32 000 triples.

What we found

This was the first experiment I did with RDFLib and various stores. Therefore, it is prone to inaccuracy and I think that the tests should be re-run for accurate time measures.

However, this experiment raised some interesting points and bugs.

Time measurement

There are different time measures: *usr*, *sys*, *wall* (also called *real*).

We ended up choosing *real* because our goal is to know how much time a user will wait. *usr* and *sys* times don't seem to account for work doing by the triple-store but only by the time spent in Python.

Bulk inserts are better than iterative inserts

If a store supports it, use bulk insertion with `graph.addN()`. It is much faster. Another way of using `graph.addN()` is to parse a file in memory in tempory graph `mem`, then load this graph into the one you want: `graph += mem`. It will use `addN()`.

Bug in rdflib-sqlachemy: `addN()` don't write anything

Reported here: https://github.com/RDFLib/rdflib-sqlalchemy/pull/8

Slow bulk insertion with rdflib-sqlachemy

There is no real bulk insertion with rdflib-sqlachemy, as it commits for each triple. Reported here: https://github.com/RDFLib/rdflib-sqlalchemy/issues/9 and https://github.com/RDFLib/rdflib/issues/357

Cannot insert blank nodes in sparqlstores

Fix provided by @pchampin as an alternative SPARQLStore: `bnsparqlstore.py`. It converts blank nodes to special URIs.

Insert time rises with respect to store size?

Only a suspicion for Sleepycat, needs real testing.

1.2 Benchmarking query capabilities for triple-stores

1.2.1 Context

After some experiments with *Benchmarking insert into triple stores*, we decided to refocus on store queries because it is a widely used scenario for kTBS users.

The goal is to explore the capabilities of several triple stores for different query types.

The queries we used are taken from SP2Bench.

1.2.2 Results

Exploration

We first explored the different stores against the queries. Each store had one graph of 500 triples. Queries ran anywhere from 5 ms to 5000 s depending on the store and the query. Results here: query32000_full.ods or in `bench_results/query32000_full.ods`.

It's a bar plot: each bar is query for a store (resulting in nb queries * nb store bars). The y-axis is the query time (real time). Tested stores are:

- Virtuoso
- 4store
- PostgreSQL
- Sleepycat
- In-memory
- Jena/Fuseki

Results also showed variation between runs. We decided to make more runs to have a good idea of the mean and look at the standard variation to see if the results were ok.

Changing perspective

We changed how we looked at results. We wanted to see the evolution of query times against the number of triples in one graph, or the number of graphs per store.

We also discarded some triple stores, see *Discarding store in the benchmark*.

Results:

- f(number of graphs in one store) = query time
- f(number of triples in one graph) = query time

Comments on f(number of graphs in one store) = query time

We observe that all measures for 5 graph / store are greater than the other ones. This lead us to dismiss these points.

For *4store* there are only measures for 1 graph / store. When trying to insert triples for 5 graphs / store in 4store, the computer started to swap and never finished. A decision was made to stop testing this store.

For *Sleepycat*, *Postgres* and *Virtuoso*, most of the query times seem to be constant with respect to the number of graph per store. Except for a few queries like q2 and q12a for Sleepycat and Postgres, and q2, q3b, q3c and q9 for Virtuoso.

We observe a weird behavior of *Sleepycat* and *PostgreSQL* on query 12a. This query is the same query as q5a, except q12a is a `ASK` and q5a is a `SELECT`. It turns out that q12a takes longer than q5a. This only appears on stores directly managed by RDFlib, which lead us to think that it's a bug in how RDFlib handles some `ASK` queries.

q2 acces large strings (abstract of articles), which is a reason why it takes longer than the other queries.

Comments on f(number of triples in one graph) = query time

We tested the query times as function of the number of triples in one graph in the stores: Sleepycat, PostgreSQL and Virtuoso.

The measures were:

- 32000 triples
- 256000 triples
- 1000000 triples

For PostgreSQL, queries 2 and 12a were not done for 1m triples, as it would have taken too much time.

For Sleepycat, all queries were not done for 1m triples. I was unable to insert 1m triples in a Sleepycat store, running the insert for 1 day was not sufficient. Plus, the python process was at state *sleeping*. Another try at this should be done.

Queries for PostgreSQL are constant in times (except q2 and 12a). It seems to be the same thing for Sleepycat, but we don't have points for 1m triples. Both PostgreSQL and Sleepycat queries (except q2 and q12a) are in the range 10-100 ms, which is acceptable.

Almost all Virtuoso queries are in the range 10-5000 ms. There are greater variation between queries than with PostgreSQL and Sleepycat. Further more, we don't have a clear understanding of a how the queries behave. We see that most queries takes more time when running on a 256k triples graph than on a 32k triples graph. But most queries takes approximately the same time, or even less time, when performing on 1m triples than on 256k triples graph.

This tests should be run another time for more accurate and understandable results. Additional points should be measured (64k, 128k, 512k, 700k triples / store).

Discarding queries

In order for the benchmarks to take less time, we removed some queries that took a lot of time (q2, and q12a).

The queries left are: query all, q1, q3abc, q4, q5ab, q6, q7, q8, q9, q10, q11, q12c.

Forking

Our goal was to simulate multiple user using kTBS by doing queries at the same time. The benchmarks we did so far didn't test that.

We decided to use fork to make multiple parallel queries.

We first tried to do a fork around `graph.query()` call, but this failed for Sleepycat as the `open()` call was done by the parent process only once (see info on error in the *Sleepycat memory error*).

Therefore, we put the fork around `graph.open(); graph.query(); graph.close()` to make it work with Sleepycat.

We benchmarked this against Sleepycat for this configuration. Each `open`/`query`/`close` query took around 1 s. But the open/close was taking the whole time (~ 1 s for `open`/`close` and 0.01 s per `query`). The benchmarks were not relevant, we were really benchmarking the `open`/`close` and not what we were interested in: the `query`.

To overcome this problem, we put together a lot of queries inside an `open`/`close`. It looked like this cocktail:

- fork starts
  - store `open()`
    - Doing 50 times this:
      - query all, q1, q3abc, q4, q5ab, q6, q7, q8, q9, q10, q11, q12c
  - store `close()`
- fork stops

Each fork took around 10 s (1 s for `open`/`close` and 9 s for the queries). This time we were really benchmarking the queries rather than the `open`/`close`.

Result figure

The figure that compares parallel queries (forks) vs. sequential queries is here.

On the x-axis is the number of queries in parallel (for forks, in green) and the number of sequential queries (in blue). On the y-axis is the time taken to run the *cocktail* of queries (see above).

We see that for a number of queries greater than or equal to 2, it is more efficient to do parallel queries. There is a two-fold factor between sequential queries and parallel queries.

Furthermore we see that there is only a tiny time difference between 1 fork and 2 forks, meaning that parallel queries really is best.

1.3 Discarding store in the benchmark

1.3.1 Discarding Jena

When doing the benchmark on each store with 1 graph of 32000 triples, Jena proved to be very inefficient.

For query 4, it took around 1000 s. Some other queries took around 100 s. The total time taken by Jena in the benchmark is too high if we want to make more tests with repetitions.

1.3.2 Discarding 4store

When doing the benchmark on each store with 5 graph of 32000 triples per store, 4store began to swap on query 2.

The test was stopped when swap exceeded 3 Go.

1.3.3 Discarding IOMemory

IOMemory is not tested for store with more than one graph because it can't handle more than one graph.

1.3.4 Discarding query 12b

After discarding Jena and 4store, the last biggest query time was with Postgres on query 12b. So this query was discarded for all store.

CHAPTER 2

Notes

2.1 Sleepycat memory error

2.1.1 The problem

When trying to do some benchmarks with Sleepycat, the following error was raised:

> Cannot allocate memory – BDB2034 unable to allocate memory for mutex; resize mutex region

The solution

To solve this, go inside the Sleepycat database folder, and do a `db_recover`.

Origin of the problem

The problem probably came from the lack of `graph.close()` after opening a Sleepycat database in RDFLib with `graph.open()`.

2.1.2 The second problem

Another error occured:

> bsddb.db.DBRunRecoveryError: (-30973, 'DB_RUNRECOVERY: Fatal error, run database recovery – PANIC: Invalid argument')

Solution

Same as before, run `db_recover` in the sleepycat folder.

2.2 Load RDF files in triple stores

2.2.1 How to load rdf files into triple stores?

Virtuoso

Using the `isql` command line interface. Then:

```
DB.DBA.RDF_LOAD_RDFXML_MT (file_to_string_output ('mydata.rdf'), '', 'http://graph_uri');
```

Note: the virtuoso server must be run in a location parent to the directory containing `mydata.rdf`.

Jena

Use the cli `s-put` as:

```
s-put {data_store uri} {graph_uri} {data_file}
```

4-store

Use the cli `4s-import` as:

```
4s-import {dataset_name} --model {graph_uri} {data_file}
```

Note: the http backend (`4s-http`) must not run at the same time.

Other stores (SQLAlchemy, etc.)

Use `graph.parse()` from RDFLib.

Note: this doesn't seem to work for Virtuoso.

2.3 Granting SPARQL privileges with Virtuoso

Launch the virtuoso server.

Go to the conductor, then:

1. System Admin
2. User Accounts
3. SPARQL, click edit.
4. Account roles, put `SPARQL_*` in the Selected field.

CHAPTER 3

See also

The *foundation of these benchmarks*: kTBS Bench Manager.

Some useful decorators that we rely on: sutils.

Development files:

3.1 bench.py: Automating benchmarks

A script to execute many benchmarks at once. Run benchmarks.

Usage: bench.py <bench_folder> [<output_folder>]

Options: -h –help Show this help screen.

bench.**scan_bench_files**(*directory*)

Scan a directory for existing benchmark scripts.

Parameters **directory** (*str*) – path to the directory containing the benchmark scripts

Returns tuple

CHAPTER 4

Indices and tables

- *genindex*
- *modindex*
- *search*

b