Computing Continuous SPARQL Query over RDF Streams on Storm Platform

By Usman Younas and Sander Breukink

Introduction

The amount of data on the web has massively increased over the years, and it doesn't show any signs of stopping. Hence the demand for analyzing it in reasonable amount of time is also increasing. The Resource Description Framework (RDF) is a data model whose purpose is to form a comprehensive framework to integrate data from different fields. It is a flexible data model used in the Semantic Web (a Web of data) on which we can do querying or reasoning. In this report we discuss several ways of executing SPARQL queries on RDF data and compare their results based on several testing scenarios. For this we will use the Apache Storm Framework with different topologies. We will also run these topologies on a testing set and compare these results.

Related work

Stream

We are analyzing streams in this project. This means that data comes available triple by triple, but we already start analyzing as soon as the first triple arrives. The formal definition can be found at [6].

RDF

In this project, we analyze RDF triples. An RDF triple consists of 3 strings: a subject, a predicate and an object. Fore more information on RDF triples we refer to [5].

Bloom Filters

For communication between the spouts and bolts, we use bloom filters. Bloom filters are a quick way way to see if an element is in a set or not, but they have a risk of false positives. An example of a bloom filter with 18 bits, 3 insertions and 3 hash functions is shown in figure 1. More information on bloom filters can be found in [2].

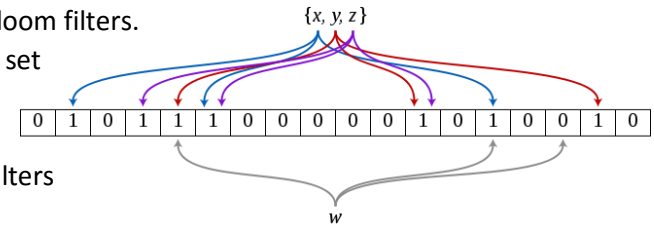

Figure 1

Storm

Apache Storm is the framework we used. This means that our application is designed as a topology in the shape of directed acyclic graph consisting of spouts and bolts acting as the graph vertices. For more information on Apache Storm, we refer to [1].

Our Terminologies

In our topology we use 4 kinds of classes: Spouts, BoltBuilders and BoltProbers and BuilderProberBolts. Note that all these

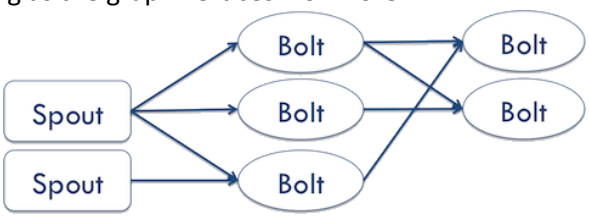

Figure 2

classes can run on different servers in parallel, meaning our solution is scalable enough for rather big streams of RDF data. A theoretical example is shown in figure 2.

Spout

In the RDF framework, we can create several spouts. In our program, a spout fetches the data and passes it on to a bolt, which we'll discuss after this section. In our topologies, we choose to use only a single tuple every time because we have only one data source.

The spout calls a function nextTuple() several times until the end of the input file is reached. This functions makes sure a new line from the input file get read, takes the data from this line and transforms it into a rdf triple which get's passed on to the collector that passes it on to the bolts.

BoltBuilder

The bolt builder analyzes all the incoming triples and creates bloom filters. All the triples that satisfy the query are stored in the bloom filter, all the triples that don't are left out. We use bloom filters because they take up very little memory, which means we can search through them really fast.

BoltProber

The BoltProber class takes tuples and bloom filters as input. It loops through all the tuples and see if it matches with all the bloom filter. If a tuple matches, the subject of that tuple corresponds with the selection query, otherwise it doesn't. Note that false positives can occur due to the use of bloom filters.

BuilderProberBolt

The BuilerProberBolt analyzes incoming triples and makes bloom filters based on this RDF data. These bloom filters are then sent to other bolts for further analysis.

Processing real-time RDF Data Streams

Data generation

For testing purposes we used a small set of synthetic data. A subset of this is shown in figure 3. This data set contained 3 different predicates, namely work, diplome and paper. Using these 3 predicates we can run all the different joins we have on this data set. This has proven very useful before executing the queries on larger data sets.

```
Sophie Paper kNN
Justine Paper kNN
Fabrice Paper kNN
Justine Work INRIA
Bob Work INRIA
John Work INRIA
Frederic Work ECP
Sophie Diplome Ph.D
Bob Diplome Ph.D
Lea Diplome Master
```

Figure 3

Queries Decomposition

In all queries we want to find subjects based on predicates values so. We splitted queries into 3, graph based categories. Namely one variable joins, two variable joins and multi variable joins.

One variable join

In a one variable join, all 3 predicates and objects must be the same. So one subject must match with all 3 predicates and all 3 objects. If it meets those requirements, the subject get's added to the final join. A general figure for this is shown on the left side of figure 4. An example query would be: select the names of all the people who wrote a paper for KNN, work for ECP and have a Ph.D Diplome, this query is

graphically represented on the right side of figure 4.

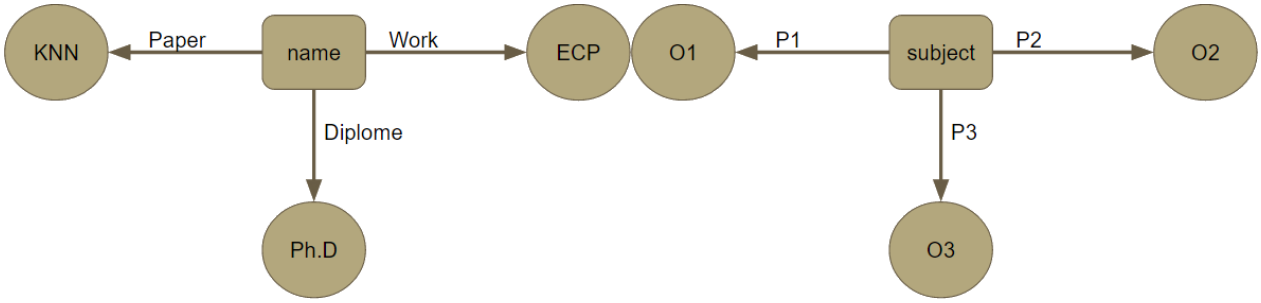

Figure 4

Two Variable Join

In a Two Variable Join, only 2 predicates and objects have to be the same. So this join is less strict than the previous join. But here we can also select the unknown object rather than just the tuples subject. A general figure for this is shown in the left side of figure 5. An example query would be: select all authors and their diplome who've written a paper for KNN and work for ECP, this query is graphically represented in the right side of figure 5.

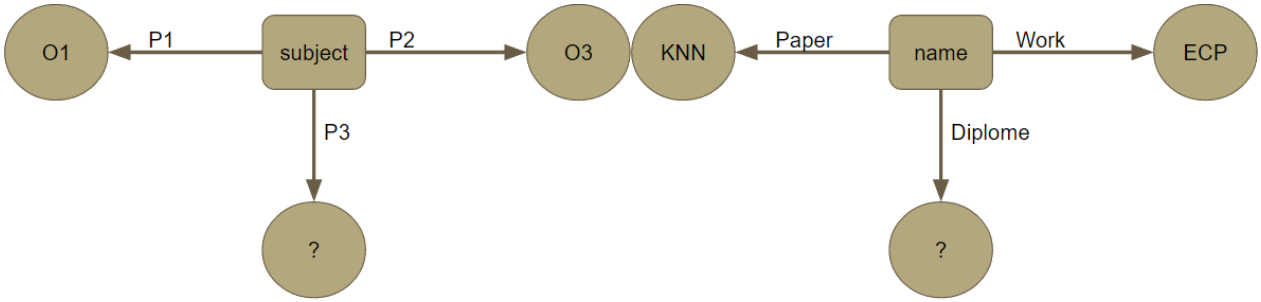

Figure 5

Multi variable join

In a Two Variable Join, only one predicate and object have to be the same. So this join is, again, less strict than the previous join. Here we can also select the 2 unknown object besides just the subject. A general figure for this is shown in the left side of figure 6. An example query would be: select all authors, their diplome and their work if they've written a paper for KNN, this query is graphically represented in the right side of figure 6.

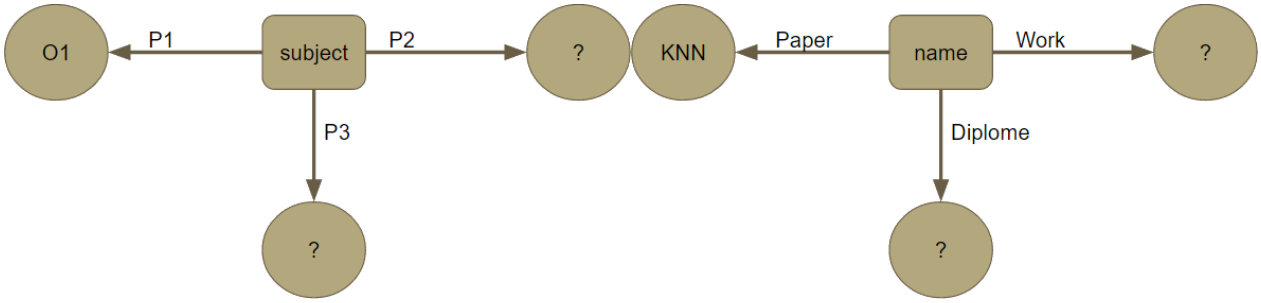

Figure 6

Joins topology

For each query, we also have a topology, this describes the way the bolts and spouts work together. Here we will describe the different topologies for the different joins we've just seen.

One and two Variable Joins Topology

The topologies for the one variable join and two variable join are the same and is graphically displayed in figure 7. firstly we have 2 builder bolts, one for the paper predicate and one for the work predicate. These 2 builders then generate bloom filters that are then passed on to the prober.

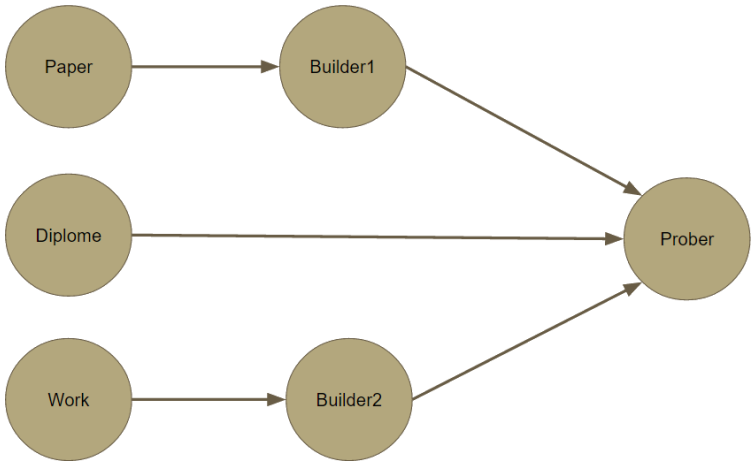

Figure 7

The prober also takes tuples with the diploma predicate as input. For each tuple it checks if it fits in the respective bloom filter from the paper predicate and if it fits in the respective bloom filter from the work predicate. If it meets both of the requirements, the tuple is allowed into the final join, otherwise it's not.

Multi Variable Join Topology

For the multi variable join, we only create bloom filters for the property that we're selecting on, so we only have one builder. But now we have 2 probers, one for each of the remaining predicates. This topology is also displayed in figure 8.

All the objects from diploma and work get passed through both probers, if an object matches with the bloom filter, it gets passed on and the respective subjects get stored as well.

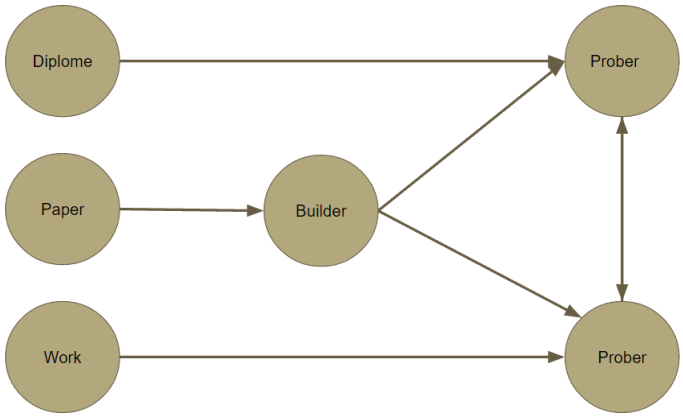

Figure 8

Sliding window integration

For the probers, it's more efficient to search data from more than one bloom filter for better real time analysis (in terms of more time flexibility). Hence we created a sliding window, which is shown in figure 9, the size of gets processed with each batch.

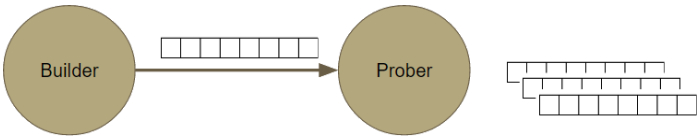

Figure 9

We can achieve this by increasing the size of bloom filter but it will increase the amount of data transfer from builder to prober and also some delay in data transfer. So using sliding window, data transfer can continue without any delay and the bloom filters can stay the same size. The size of the sliding window is defined like this:

$$Sliding\ window\ size = number\ of\ generations * generation\ size$$

Generation size means the size of the bloom filter.

The sliding window is used in the ProberBolt for the 1 variable join and the 2 variable join and in the BuilderProberBolt for the multi variable join.

Parallelism for topologies

Since the spouts, probers and builders are run on different servers, we have to carefully regulate the amount of computational power for each of them to prevent overloads and underloads. If we run our topology on a single server, we get the loads shown in figure 10. Here we observe that the 2 builders are doing fine, but the prober is heavily overloaded.

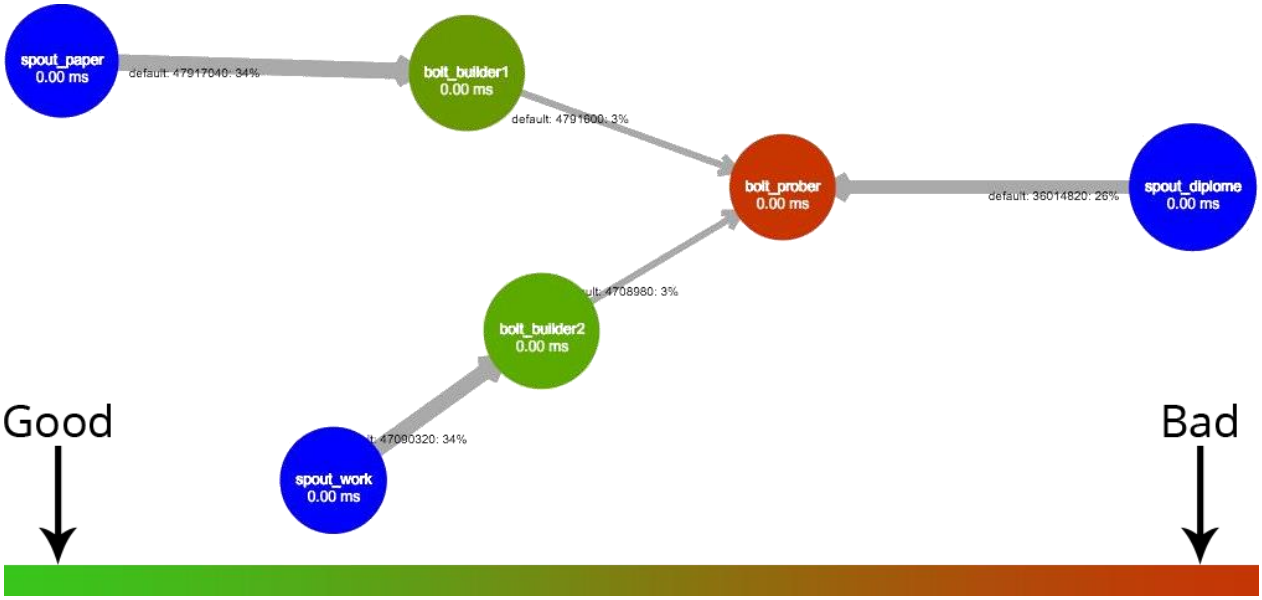

Figure 10

After these results, we tested on real servers with several different configurations to find out which one has the best load distribution. In the end, we found that a configuration with 2 servers for each bolt and 3 servers for the prober was optimal. This server distribution is also shown in figure 11.

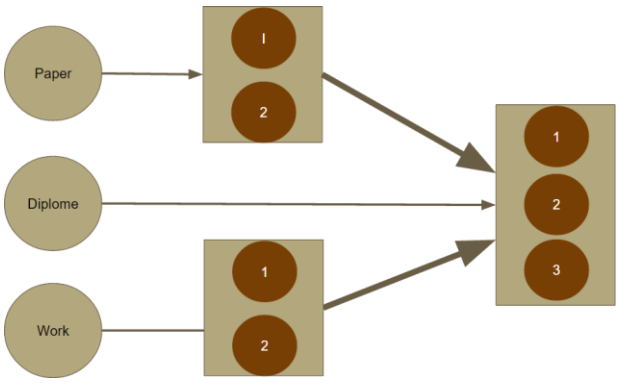

Figure 11

Deployment

Grid5000

For testing the several queries on our code, we used the Grid5000 network. This is a network that can be freely used for scientific research such as ours. The Grid5000 network consists of 1.000 servers and a total of 8.000 CPU cores. These servers are spread out over several cities in both France and Luxembourg. For more information on this network please look at [4].

Storm Cluster

In the cluster, we have 10 workers, each of which has its own supervisor. The workers connect with a zookeeper server which connects to a Nimbus server, for more information on these servers we refer to [7] and [8].

We can communicate with the nimbus server via the Storm UI and the tunnel. The entire overview of our cluster is shown in figure 12.

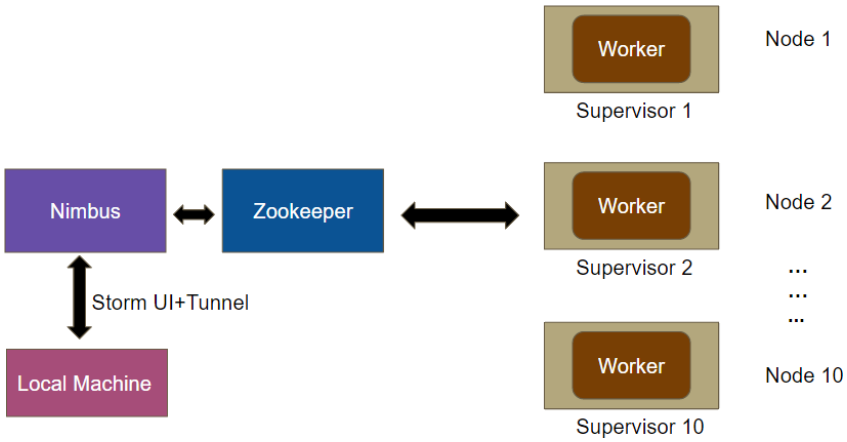

Figure 12

Results

Data processed

The graphs in figures 13, 14 and 15 show the amount of data processed by the spouts, in other words: the amount of data we fed to the bolts as a stream. To ensure the same testing environment for all the joins, we decided to keep this the same for all of them.

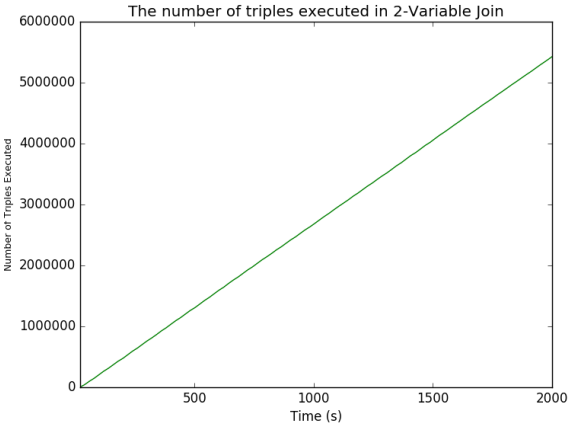

Figure 15

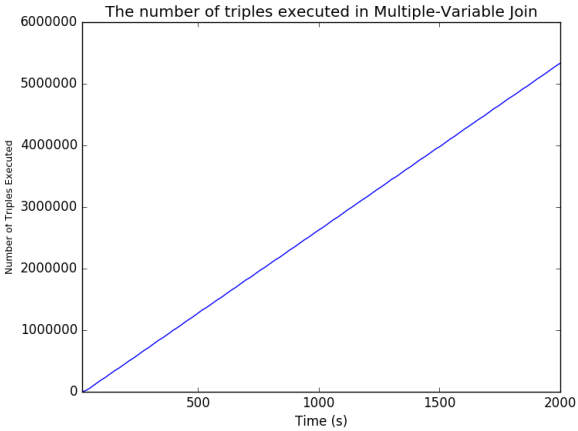

Figure 13

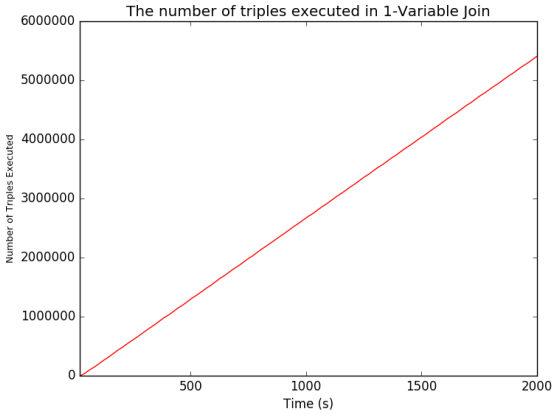

Figure 14

Process Latency

The process latency of the 3 different joins is shown in figures 16, 17 and 18. The process latency of the 1 variable join is less than the process latency of the other 2. This can be explained by the fact that the first join is more selective, in other words: it selects less data. As a consequence of this, the size of the problist will be smaller.

The process latency of the 2 variable join and the multi variable join is mostly the same due to the fact that we have 2 different probers in the multi variable join.

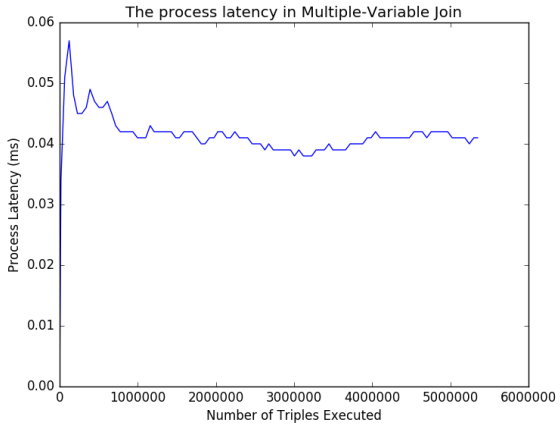

Figure 17

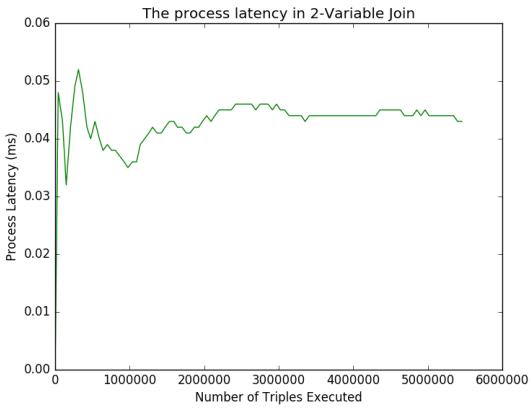

Figure 16

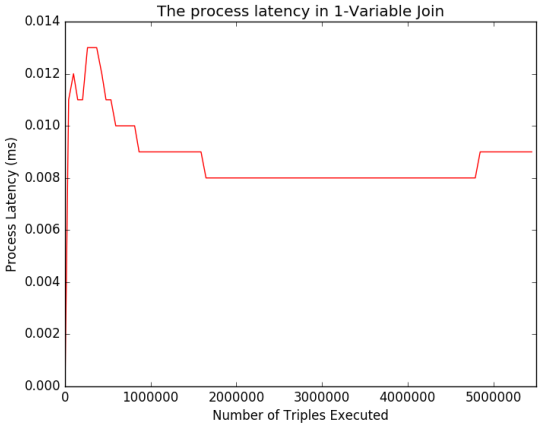

Figure 18

Data transferred

If we look at the amount of data transferred inside each sliding window, displayed in figures 19, 20 and 21. We can observe that the graph looks has roughly the same shape for all 3 joins. But we do observe that the one for the mutli variable join is about twice as much as the data transffered at the 2 other joins. This can be explained by the fact that there 2 probers rather than just one, hence there's twice as much data to be transferred.

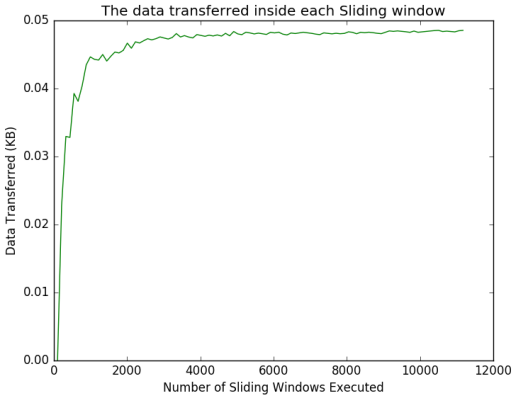

Figure 19

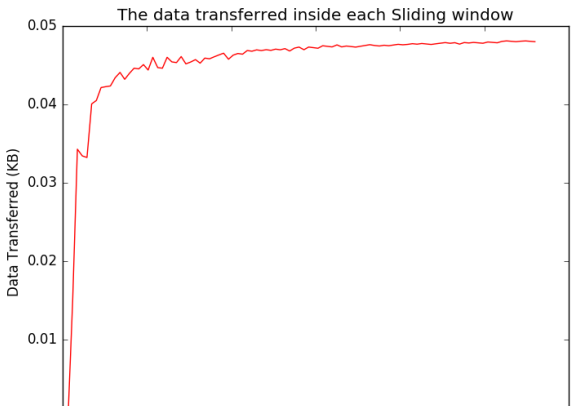

Figure 21

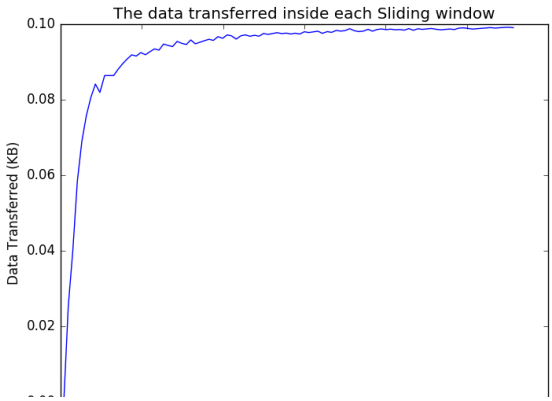

Figure 20

Analysis on processing with Sliding window

We experimented with several sliding window sizes, recall that the sliding window determines the amount of bloom filters in each batch by the prober. The sliding window size is defined as the multiplication of the generation size and the number of generations.

Impact of Number of Generations

We found that increasing the number of generations while maintaining the same window size has 2 consequences. The first one is that the amount of data transfer is decreased and the second one is that the amount of processing time is increased. These results can also be seen in the graphs shown in figures 22 and 23 which show the amount of data transferred and the process latency for several amounts of generations.

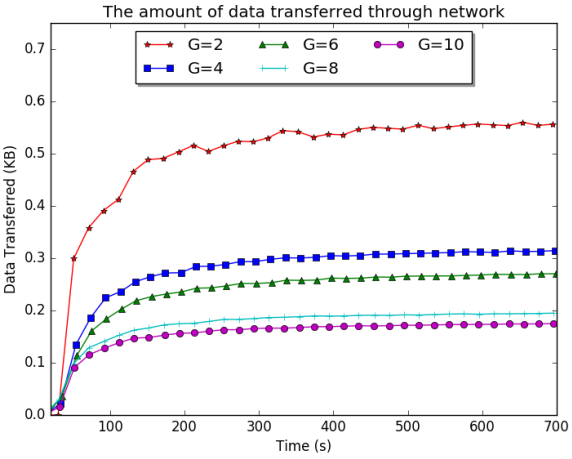

Figure 23

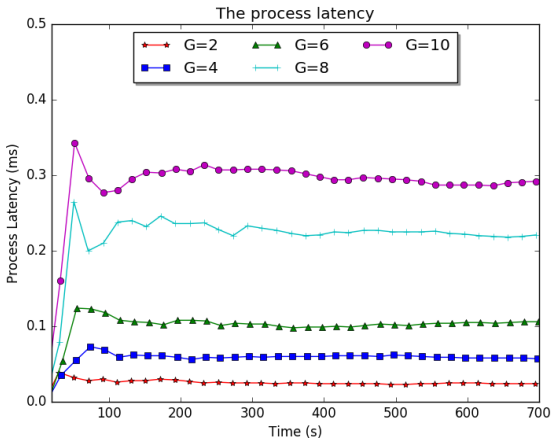

Figure 22

Impact of Sliding Window size

If we change the sliding window size and maintain a constant number of generations, we see somewhat different results. The increase again has 2 consequences, which are an increase in the amount of data transferred, but we also see a decrease in the processing time needed.

Lehigh University Benchmark

For benchmarking we use the Lehigh University Benchmark (LUBM) code, their website can be found at [3]. We choose this benchmark because it allows us to easily generate data and it comes with a lot of queries that we can use to compare the results.

Because a subset of all the queries it offers was sufficient, we decided to only test with the queries displayed in figure 24.

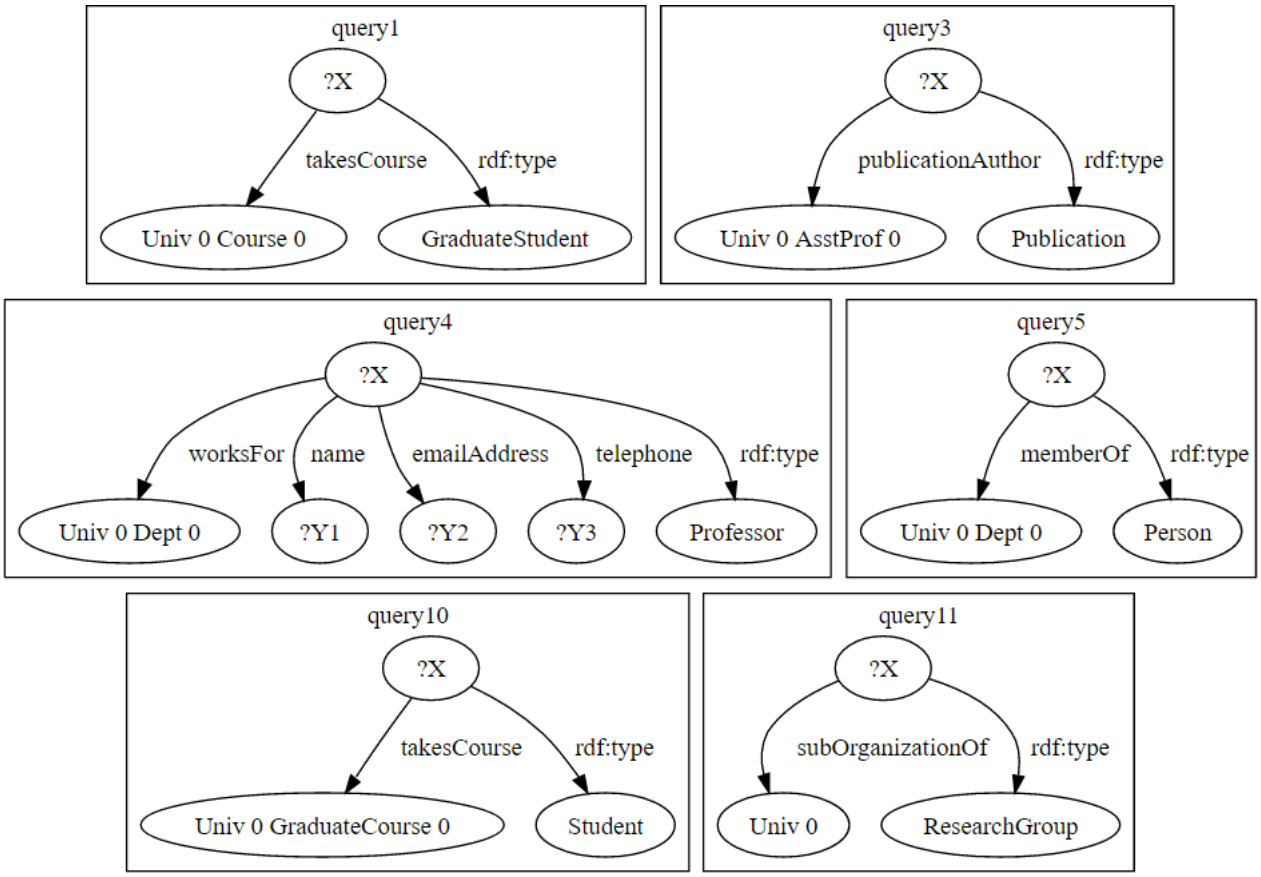

Figure 24

RDF Data Generation

The LUBM code generates a virtual university in RDF data. This university consists of publications, research groups, departments, students, courses, professors and lecturers. A graphical representation of this can be found in figure 25.

Since this data is very representative for real-world scenarios, it's a very good set to do tests on. Which is one of the reasons we choose this open source project for our benchmarks.

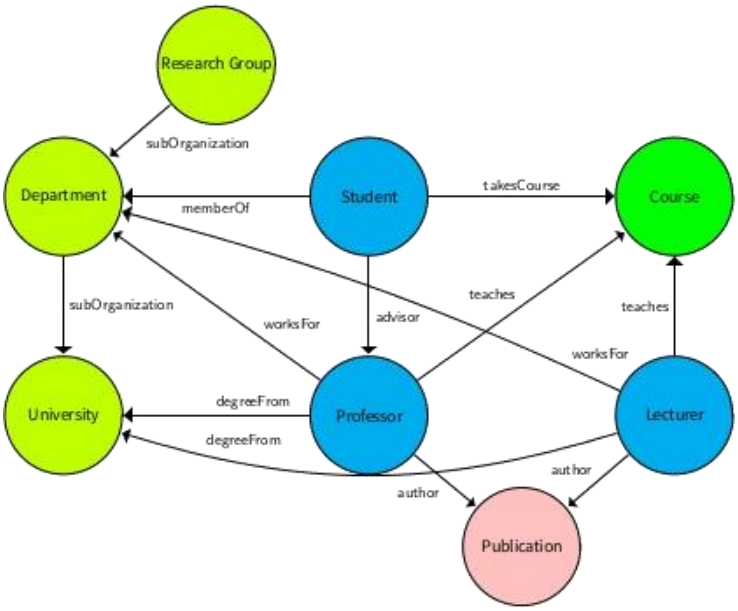

Figure 25

Triples Randomization

The triples generated by the LUBM code are randomized before we run tests on it, we do this for multiple reasons. The first one is that it gives us better distribution

of the data, which leads to better analysis with the bloom filter. Another one is that it's more realistic when the data is sent as a stream.

Read RDF Data through Jena API

For reading the RDF data, we used the Jena API, their website can be found at [9]. We used this API to combine the data from multiple sources and unify all these sources into RDF triples that can be understood by our own Java code.

Benchmark queries/test cases

Here we will discus 2 of the queries and the results they gave us.

Query 3

When we ran query 3 on the topology, we got the loads as shown in figure 26. We see 2 spouts, one that feeds data into a builder and the other that feeds data into the prober. Next we also observe that the builder sends bloom filters to the prober, the fact that this amount of data is far less than the amount of data that builder receives, shows how that the bloom filters do a really good job at saving memory.

Lastly, we also observe that the servers for both the builder and the prober are colored green, indicating that all the servers are not overloaded, which is good.

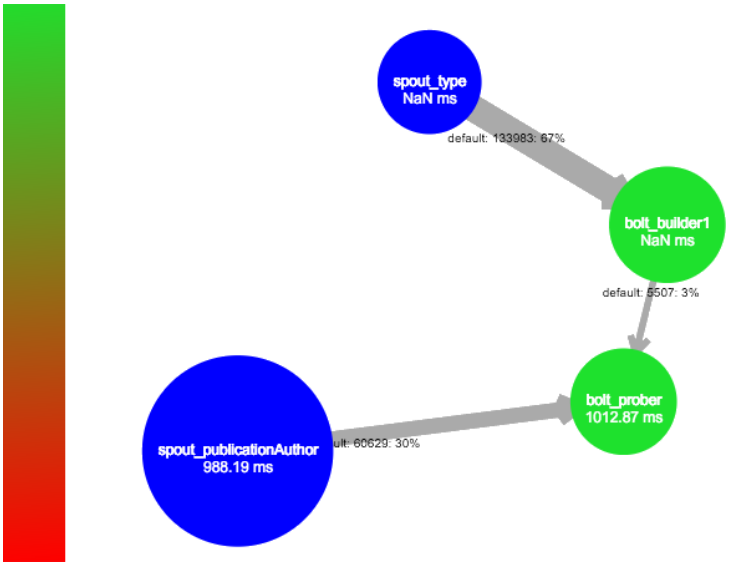

Figure 26

Query 4

Since query 4 is a multi variable join that selects all professors and their information, we use a different topology than in the previous query. This can also be observed by the visualization in figure 27 which shows 3 spouts, 1 builder and 2 probers.

Again we see that the amount of data coming from the builder is far less than the amount going into it, hence this confirms again that the bloom filters are doing their job well. We also see again that all the servers are again green, hence the load is evenly distributed.

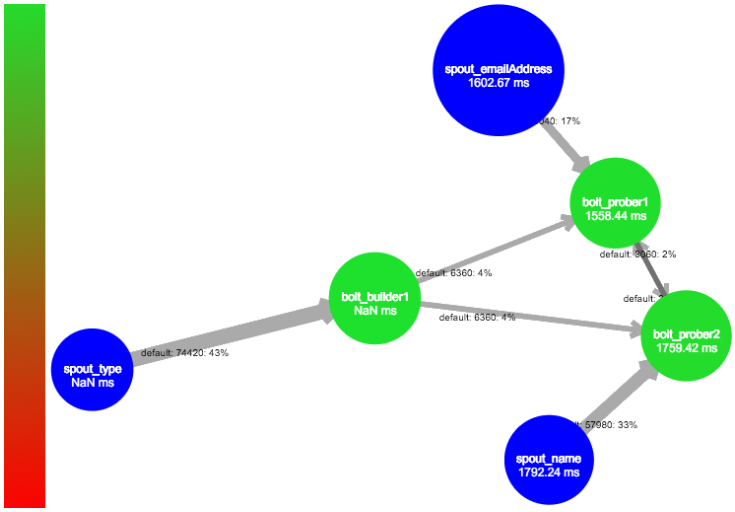

Figure 27

Benchmark results

From the benchmarks on these 2 queries, we received several outputs. Firstly the number of triples executed are shown in figures 28 and 29. We observe that both of these grow in a linear way, but that the number of triples executed in query 4 is a

lot more than the amount in query 3. This can be explained by the simple fact that there were more triples that needed to be analyzed for this query, in other words: there was more data that was useful.

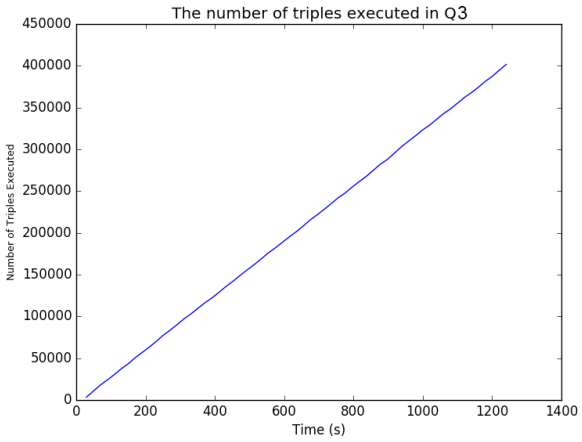

Figure 29

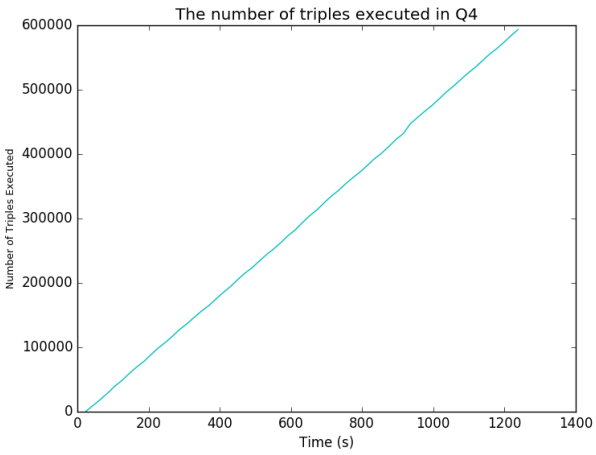

Figure 28

We can also look at the amount of data transferred, displayed in figures 30 and 31. We observe similar looking graphs, but again higher numbers for query 4. This can be explained that query 3 is highly selective, which means it returns a low amount of data, and that query 4 is lowly selective, which means it returns a rather large amount of data.

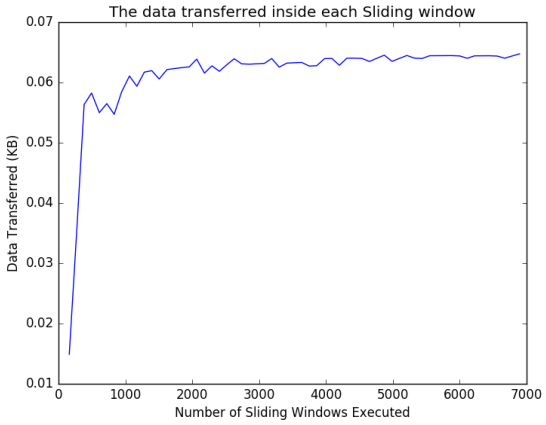

Figure 31

Figure 30

We also looked at the process latency, these are displayed in figures 32 and 33. We see that query 4 has a significantly higher latency than query 3, this can be explained by the simple fact that it just has more data to handle.

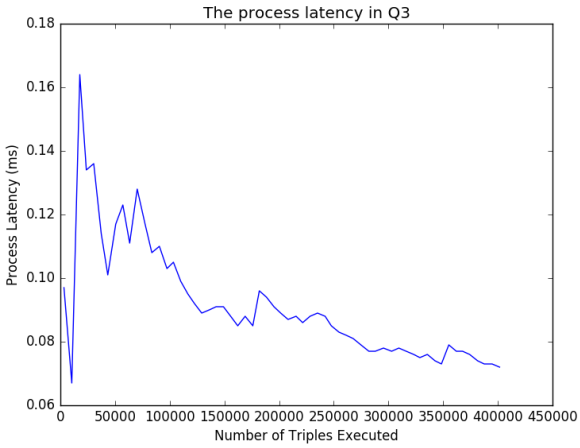

Figure 33

Figure 32

References

[1]. http://storm.apache.org/releases/2.0.0-SNAPSHOT/index.html
[2]. https://en.wikipedia.org/wiki/Bloom_filter
[3]. http://swat.cse.lehigh.edu/projects/lubm/
[4]. https://www.grid5000.fr/mediawiki/index.php/Grid5000:Home
[5]. https://www.w3.org/RDF/
[6]. https://en.wikipedia.org/wiki/Data_stream
[7]. https://zookeeper.apache.org/doc/r3.4.8/
[8]. http://www.nimbusproject.org/docs/2.10.1/
[9]. https://jena.apache.org/