HOBBIT

Collaborative Project

Holistic Benchmarking of Big Linked Data

Project Number: 688227 **Start Date of Project:** 2015/12/01 **Duration:** 36 months

Deliverable 6.2.1
First version of the Faceted Browsing Benchmark

Dissemination Level	Public
Due Date of Deliverable	Month 18, 31/05/2017
Actual Submission Date	Month 18, 31/05/2017
Work Package	WP6.2 - First version of the Faceted Browsing benchmark
Task	T6.2
Type	Report
Approval Status	Approved
Version	1.0
Number of Pages	16

Abstract: This report describes the achievements to build the first version of the benchmark on Faceted Browsing. It includes a description of the work that has been carried out as well as the details of the implementation behind the benchmark.

Project funded by the European Commission's Horizon 2020 Program.

History

Version	Date	Reason	Revised by
0.1	26/04/2017	First draft created	Henning Petzka
0.2	02/05/2017	Internal review comments	Bastian Haarmann
0.3	12/05/2017	Peer review comments	Irini Fundulaki
1.0	19/05/2017	Approval	Henning Petzka

Author List

Organization	Name	Contact Information
IAIS	Henning Petzka	henning.petzka@iais.fraunhofer.de

Executive Summary

This document describes the first version of the Faceted Browsing Benchmark. It covers the workload conducted from M7 until M18.

The benchmark on Faceted Browsing aims to benchmark systems on their performance to support browsing through linked data by iterative transitions performed by an intelligent user. By developing realistic browsing scenarios through a dataset that comprises of different structural challenges for the system, we aim to test its performance with respect to several choke points in a real world scenario.

Contents

1 Introduction **5**

2 An overview of the work performed **5**

3 The benchmark in details **6**

3.1 The underlying dataset 6

3.2 The choke points 7

3.3 The KPI's 8

3.4 The benchmark; Scenarios, gold standard and randomness 9

4 The benchmark **11**

5 Experiment **12**

6 Preliminary Results **13**

7 Future work on the benchmark **14**

List of Figures

1 The ontology of the underlying dataset . . . 7
2 Two example scenarios annotated with related choke points in blue numbers . . . 10
3 SPARQL queries of Scenario 3, Query 2 . . . 10
4 Part of the results of our baseline system "tenforce/virtuoso" . . . 13
5 MOCHA results for instance retrieval: correctness . . . 14
6 MOCHA results for instance retrieval: speed . . . 14
7 MOCHA results for facet counts . . . 15

1 Introduction

Faceted browsing stands for a session-based and state-dependent interactive method for query formulation over a multi-dimensional information space. It provides a user with an effective way for exploration of a search space. After having defined the initial search space, i.e., the set of resources of interest to the user, a browsing scenario consists of applying (or removing) filter restrictions of object-valued properties or of changing the range of a property value of various data types.

For a well-established example of an implementation of faceted browsing consider an online shopping portal. There, the search space could be a certain type of clothes and, amongst others, the facets could be the size, color and price of clothes. Using filtering operations, the user is able to browse between different states of the search space in order to select the items with the desired properties. Here, a state consists of the chosen facets, their corresponding facet values and the set of instances satisfying all chosen constraints.

As a second example, consider logs of a machine in an industrial setting. An engineer might want to browse through the log data of the machines to search for malfunctions. He or she may select and unselect specific machines, specific variables, and narrow down the data selection by specifying a range of measurement results (for example to find incidences of very high temperature).

To support a user-friendly and efficient browsing experience, the underlying system has to be able to quickly compute the underlying statistics. The statistics can then help the GUI to decide on which of the facets should be displayed. In particular, to name a simple example, no facet should be suggested to the user which leads to an empty search space. In a more advanced setting, consider a numerical property. The variance of the distribution of remaining values after a potential facet selection may be used within a computation of a score for each facet (where high scores suggest to display the corresponding facet). Further, in an ideal setting, for each facet and facet-value, the number of instances that remain after applying this facet should be displayed to the user.

Of course, displaying promising facet selections is not the main task of the underlying system. More importantly, the system has to move from one state to the other in reasonable time (as fast as possible). The browsing application is of little interest, if each filter restriction requires a long time to be processed. For the computations, approximating algorithms may be useful to significantly reduce computing time.

The goal of a benchmark for faceted browsing techniques and tools is to equip solution providers with a way to check their software for their capabilities of enabling faceted browsing through large-scale structured datasets. That is, it analyses their efficiency in navigating through large datasets, where the navigation is driven by intelligent iterative restrictions. We develop browsing scenarios through a dataset, which reflect an authentic use-case and challenge participating systems on different points of difficulty. To distinguish several solutions, we aim to measure the performance relative to dataset characteristics, such as overall size and graph characteristics. The progress made in this task has as its outcome a first version of a benchmark which allows systems to check for their performance/capabilities on certain choke points of faceted browsing.

2 An overview of the work performed

We list specific steps of the progress that was made since the start of this task and which sum up the current state, before going into details of each point of progress.

- We **collected** the main **choke points** of faceted browsing, that is, the difficulties that arise for a system in enabling efficient browsing through structured datasets. Further, we **set the KPI's.**
- We explored our possibilities to develop benchmarking scenarios that resemble realistic browsing scenarios.
- We investigated the characteristics of several datasets for their suitability to benchmark systems according to the choke points collected in (i).
- We **enriched the ontology of one of the datasets** to better enable the benchmarking of systems according to the collected choke points of faceted browsing.
- We wrote **SPARQL queries** making up in total eleven scenarios of browsing scenarios **simulating a user searching through the dataset.**
- We wrote SPARQL queries necessary for a preparation of the actual browsing queries. These preparatory queries **pre-compute parameters** which are used in the queries of the actual benchmark and **allow for the randomization** of the SPARQL queries.
- We explored possibilities for the **computation of the gold standard**.
- We implemented the modules of the benchmark so as to function as part of the HOBBIT platform, which resulted in a **first version of our benchmark of faceted browsing**. To our knowledge, this is the first benchmark allowing systems to test performance on the specific choke points collected in item (i).
- We **tested the open source system" tenforce/virtuoso**[1]" as a baseline system.

3 The benchmark in details

3.1 The underlying dataset

In the initial phase, we explored all anticipated datasets provided by project partners (including iMinds transport dataset and USU's datasets from the IT and the production industry), which could potentially play the role of the underlying dataset for the benchmark on faceted browsing. The provided datasets proved not to be suitable for an extensive simulation of a faceted browsing scenario, as the ontologies were rather small and simplistic. As a result, it was necessary to extend one of the data generating algorithms for an enlargement of the underlying ontology. More specifically, the data generator PoDiGG[2] for the transport dataset of iMinds (containing train connections between stations on an artificially created map) was extended to include the possibility of delays that the trains may experience during their trajectory. The simulation of delays includes not only a time value, but also a reason for the delay (Figure 1).

For the integration of delays into the dataset, we used the Transport Disruption Ontology[3], which models possible events that can disrupt the schedule of travel or transport plans. We carefully selected several delay reasons from this ontology that may apply to train delays.

[1] https://github.com/tenforce/docker-virtuoso
[2] https://github.com/PoDiGG/podigg-lc
[3] https://transportdisruption.github.io/

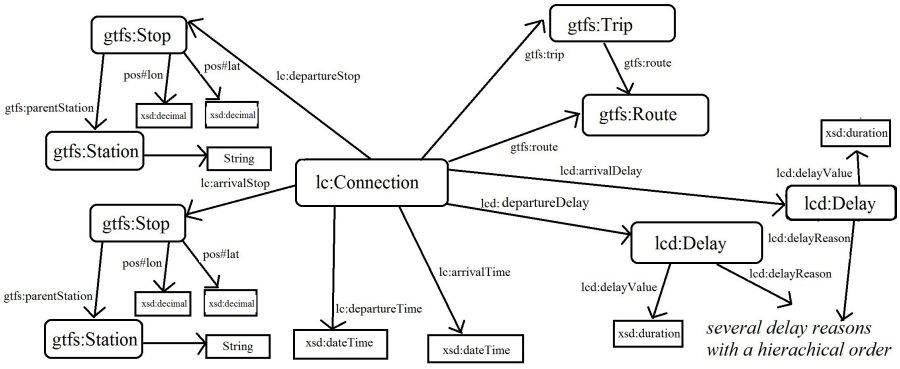


Figure 1: The ontology of the underlying dataset

On the side, the extension of the transport ontology and its use in the setting of Task 6.2 allowed an early testing phase of the involved dataset generator PoDiGG and contributed to the improvement of the generator.

3.2 The choke points

After a careful study of previous publications on faceted browsing (for example and most importantly, Tzitzikas et. al.'s "Faceted Exploration of RDF/S Datasets: A survey"[4]), a list of possible choke points was created. In a browsing scenario it is the effective transition from one state to the next one that determines the user experience. Ideally, a system uses the information of the state of the browsing scenario to return its answer to the SPARQL query that makes up the desired transition, instead of answering the query on basis of the entire dataset in its original form. This could be achieved through an intelligent database structure or through certain precomputations. Therefore, the choke points correspond to certain transitions from one state to the other during the browsing scenario. For example, a transition might consist of a property value of a directly related property to be chosen (which we will call a property value based transition). Or, as another example, a property value behind a property path of length strictly larger than one edge may be chosen (which we will call a property path value based transition) . In the latter case it is, in comparison to the first example, a more challenging task to prepare the system for the upcoming transition, and it requires a more complex choice of the system's database structure. In other cases, choices have to be made whether to better support a certain transition or the other.

Overall, the literature search resulted in a list of 14 transitions that make up the choke points of our benchmark on faceted browsing. This means that systems can be compared on quite a large set of different aspects of performance. The complete list of choke points reads as follows:

1. Property value based transition
   (Find all instances which, additional to satisfying all restrictions defined by the state within the browsing scenario, have a certain property value)

2. Property path based transition
   (Find all instances which additionally realize this property path with any property value)

[4] Tzitzikas, Manolis, Papadakos. J. Intell. Inf. Syst. (2017) 48:329. doi:10.1007/s10844-016-0413-8

3. Property path value based transition
   (Find all instances which additionally have a certain value at the end of a property path)
4. Property class value based transition
   (Find all instances which additionally have a property value lying in a certain class)
5. Transition of a selected property value class to one of its subclasses
   (For a selected class that a property value should belong to, select a subclass)
6. Change of bounds of directly related numerical data
   (Find all instances that additionally have numerical data lying within a certain interval behind a directly related property)
7. Change of numerical data related via a property path of length strictly greater than one edge
   (Similar to 7, but now the numerical data is indirectly related to the instances via a property path)
8. Restrictions of numerical data where multiple dimensions are involved
   (Choke points 7 and 8 under the assumption that bounds have been chosen for more than one dimension of numerical data, here, we count latitude and longitude numerical values together as one dimension)
9. Unbounded intervals involved in numerical data
   (Choke points 7,8,9 when intervals are unbounded and only an upper or lower bound is chosen)
10. Undoing former restrictions to previous state
    (Go back to instances of a previous step)
11. Entity-type switch changing the solution space
    (Change of the solution space while keeping the current filter selections)
12. Complicated property paths or circles
    (Choke points 3 and 4 with advanced property paths involved)
13. Inverse direction of an edge involved in property path based transition
    (Property path value and property value based transitions where the property path involves traversing edges in the inverse direction)
14. Numerical restriction over a property path involving the inverse direction of an edge
    (Additional numerical data restrictions at the end of a property path where the property path involves traversing edges in the inverse direction)

3.3 The KPI's

As Key Performance Indicators, we set the usual suspects: precision, recall, F1-score and, of course, we collect the time between query formulation and receiving of an answer and record it in form of a score measuring queries per second. These four performance values (precision, recall, F1-score and query-per-second score) are registered over all queries of all scenarios combined and additionally for each of the above choke points individually. In the latter case, the procedure is to only include into the calculation of results of those SPARQL queries that make up transitions corresponding to the choke point in question.

In addition we record a list of queries for which no query result was recorded before a specified timeout value, a list of scenarios containing such a failed query and a list of choke points that are linked to a failed query. (Note that a system can still return a result long after the timeout and possibly work on other queries in the meantime. But this breaks the order of transitions from one state to the other. Therefore, the results linked to a failed query should be interpreted with care.)

Whereas the just mentioned KPIs measure only the performance on instance retrievals (i.e. SELECT queries asking to return all instances satisfying or implementing certain relations) there is another type of queries important in faceted browsing: SELECT COUNT queries, which count the number of instances with respect to certain additional restrictions. These counts are important for suggesting possible transitions to the user and to guide in the browsing session. From a given state, only transitions leading to a sensible number of results should be suggested. Therefore, we included six SELECT COUNT queries into each of our 11 scenarios. Performance on these facet counts is also measured in more than one way to give a more complete picture of performance. In the following, we mean by 'error' on a single count query the absolute value of the differences between expected and received count result. For the performance on the count queries we record the following:

- The overall error, equal to the sum of individual errors over all count queries
- The average error which is simply the overall error divided by the number of queries.
- The overall error ratio as the overall error divided by the sum of expected count results
- The average error ratio as the sum of "error divided by expected result" over all queries.

3.4 The benchmark; Scenarios, gold standard and randomness

Subsequently, we created several lists of ordered SPAQRL queries, where each list simulates one browsing scenario (Figure 2). Developing the scenarios took place with the ambition to come up with browsing sessions that make sense in a real-world browsing scenario as well as that cover all types of transitions as specified by the choke points. The overall workload of the benchmark comprises 173 SPARQL queries divided up into 11 scenarios, each simulating a single user browsing through the dataset. Every choke point appears at least a few times throughout all scenarios and one SPARQL query may contribute to the score related to several choke points.

To allow for randomness in the SPARQL queries that make up the benchmark, we wrote additional queries computing several variables within the SPARQL queries. These variables are filled in by either instances or values (of types xsd:dateTime, xsd:duration, xsd:decimal). In the first case, for each such preparatory query, one instance is randomly selected from its result list and plugged into a corresponding placeholder to complete one of the queries belonging to a browsing scenario. In the latter case, the value is either computed from constants that depend on the underlying dataset, or selected from a strategically chosen interval in a random fashion. By choosing the interval we can assure that values will definitely increase, or alternatively definitely decrease instead. Proceeding in this way of precomputing parameters equips us with a method that can easily lead to varying browsing scenarios by simply changing a seed for the generation of random numbers. The seed can be specified when creating a benchmark on the platform by the user of the platform.

Testing the SPARQL queries on different systems eventually resulted in a stable form of queries, which most of the common systems can process correctly. The query structure itself was kept rather simplistic and straightforward (Figure 3) and was not adjusted according to the performance of a single system so that results will not be biased toward a certain query engine such as the one used for the computation of the gold standard.

Scenario 3

- Search space = Instances of connections

(1) Restrict the lat and long for stations 7
(2) Select a station. 3
(3) Select all connections that have a delay 2
(4) Select a delay reason. 4
(5) Deselect the delay reason. 10
(6) Restrict to delay value $> a$ 7,8,9
(7) Restrict to delay value $> b$ where $b < a$ 7,8,9
(8) Select a delay reason with subclasses 4
(9) Select a sub-reason 5
(10) Restrict to delay values $< c$ 7,8
(11) Restrict to delay values $< d$ where $d < c$ 7,8

Scenario 5

- Search space = Instances of connections

(1) Select an interval for long of connections 7
(2) Select an interval of lat 7
(3) Select a route that is going through that region 1
(4) Select an upper bound for the time 6,8,9
(5) Select a lower bound for the time 6,8
(6) Select a delay reason 4
(7) Deselect a delay reason 10
(8) Select a different delay reason 4
(9) Change the bounds for the time 6,8

Figure 2: Two example scenarios annotated with related choke point in blue numbers

```
PREFIX lc: <http://semweb.mmlab.be/ns/linkedconnections#>
PREFIX gtfs: <http://vocab.gtfs.org/terms#>
PREFIX rdfs: <http://www.w3.org/2000/01/rdf-schema#>
PREFIX lcd: <http://semweb.mmlab.be/ns/linked-connections-delay#>
PREFIX td: <http://purl.org/td/transportdisruption#>
PREFIX xsd: <http://www.w3.org/2001/XMLSchema#>

SELECT (COUNT(DISTINCT(?connection)) AS ?count)
WHERE {
    ?connection lc:departureStop ?stop ;
        lcd:departureDelay ?delay .
    ?delay lcd:delayReason ?reason .
    ?reason a ?reasonClass .
    ?reasonClass rdfs:subClassOf* %s .
    ?stop <http://www.w3.org/2003/01/geo/wgs84_pos#lat> ?lat ;
        <http://www.w3.org/2003/01/geo/wgs84_pos#long> ?long .
FILTER(((%s^^xsd:decimal  < ?lat) && (?lat < %s^^xsd:decimal)
    && (%s^^xsd:decimal < ?long) && (?long < %s^^xsd:decimal) ) ) }
```

Figure 3: SPARQL queries of Scenario 3, Query 2

The exploration of possible systems to use for the computation of gold standard results was a task in itself. Initially aiming to use two independent systems to collectively compute the gold standard (equal to the query result both systems agree on) we ultimately settled on using only one single system. Not every tested system was able to return the expected answers to all SPARQL queries, and some got quickly unreliable when increasing the size of the underlying dataset. To mention a concrete example, although being a competitive system for instance retrieval, RDF3X does not support all FILTER expressions or neither SELECT COUNT queries. Both of these features of the SPARQL 1.0 query language need to be supported by systems to benchmark them on our browsing scenarios. According to our tests, virtuoso correctly answered queries in a reliable fashion and therefore proved itself suitable as the system of choice to compute the gold standard.

4 The benchmark

The full source code of the benchmark can be found on the HOBBIT github page[5]. The benchmark is also integrated into the HOBBIT platform[6], where participants can readily test their systems after writing a system adapter following the Common API for the Mighty Storage Challenge (MOCHA)[7] of the ESWC 2017, which the benchmark of Faceted Browsing is part of.

Specific aspects of the workflow of the benchmark will be summarised and commented in the following. For more details on the workflow of a general benchmark on the HOBBIT platform and the interplay of components, see the Overview section of the Wiki to the HOBBIT platform[8].

1. An instance of virtuoso for the computation of the gold standard is initiated and the dataset for the benchmark is added to the database.
2. The virtuoso instance goes through all preparatory queries and assigns values to certain parameters. The seed value, determined as an input variable on the HOBBIT platform when starting the benchmark, is used as a random seed in the assignment of values.
3. The data is sent in bulks to the system adapter over the messaging system of the platform. A signal follows when all data has been sent.
4. After receiving a signal from the system that all data has been inserted successfully, the scenario related SPARQL queries are created, evaluated against the gold standard virtuoso instance and sent to the participating system. Note here that it is crucial for the benchmark that the SPARQL queries receive the participating system in the predefined order. This is certainly necessary as it is the transitions from one state within a browsing session to the next one that make up the choke points. Therefore, a change of the order of queries results in a change of the choke points related to queries. The guaranteed preservation of the order is currently ensured by (the suboptimal solution of) a short time out. In an improved version, the task generator could wait for an acknowledgement signal to send out a next query. As soon as the core-functions of the HOBBIT platform provide such an acknowledgement signal, this feature can be readily implemented. According to our tests, the time out solution never failed. Also, the correct order of queries can be easily and transparently retraced from suitable system logs that monitor a strictly increasing task id.
5. The evaluation module compares the expected results with the received results, and it evaluates them according to the KPI's specified above. If either no result was returned at all or the query execution surpassed a certain time out value, then the time out value is assigned as the result for this specific benchmark.

[5] https://github.com/hobbit-project/faceted-benchmark
[6] http://master.project-hobbit.eu/
[7] https://project-hobbit.eu/challenges/mighty-storage-challenge/
[8] https://github.com/hobbit-project/platform/wiki/Overview

This approach was chosen strategically for two reasons. Firstly, query performance on one or two particularly hard queries should not turn out to be the single factor contributing to the overall results (and to the results of the choke points related to these queries). A maximum for the allowed time value keeps all results in distance to each other. If there was no such maximal value, then one query could take significantly longer than all other queries and hence significantly contribute to the final score, while the impact on the performance score of all other queries would be much lower. A maximal time value avoids this issue that potentially only a few queries determine most of the overall performance, while the queries do not necessarily have to be of comparable difficulty. At the same time, the performance on KPIs measuring correctness still remains unaltered. If the system returns an answer to a query before the end of the benchmark run then, even after the timeout, the results are evaluated.
Secondly, systems that do not return an answer before the end of the run of the whole benchmark should not be given an advantage over systems that take a long time to return a (possibly correct) answer. For the evaluation, it needs to be decided how to treat such queries for which no result was recorded. There are only very few options. If we do assign a value for the query time of unreturned queries, then it should be at least as large as the time of all systems that successfully sent an answer to the query. (The time assigned should not be too large as discussed for the first reason.) If we do not assign a value and take out the unanswered query from the evaluation for the system, then this would lead to an unfair evaluation as the following illustrative example shows: Suppose there are only two queries related to a certain choke points. The first query is easy and the second one is quite hard. Suppose further that System 1 and System 2 do equally well on the first (easy) query. The second query is not answered by System 2 before the end of the benchmark run, but System 1 returns a result after a long time still within the time frame of the benchmark run. Then, taking out the second query for the evaluation of System 2, this system gets assigned a very high query per second score. On the other hand, System 1's query per second score is very low. Therefore, the results suggest that System 2 did so much better, while one would probably agree that System 1 performed better over both queries combined.
Assigning the same maximal value to both, i.e., systems that do not return a query result and systems that take too long, follows a simple rule: If there is no query return before a specified timeout, the system failed the task and gets the timeout value assigned as the timely result.

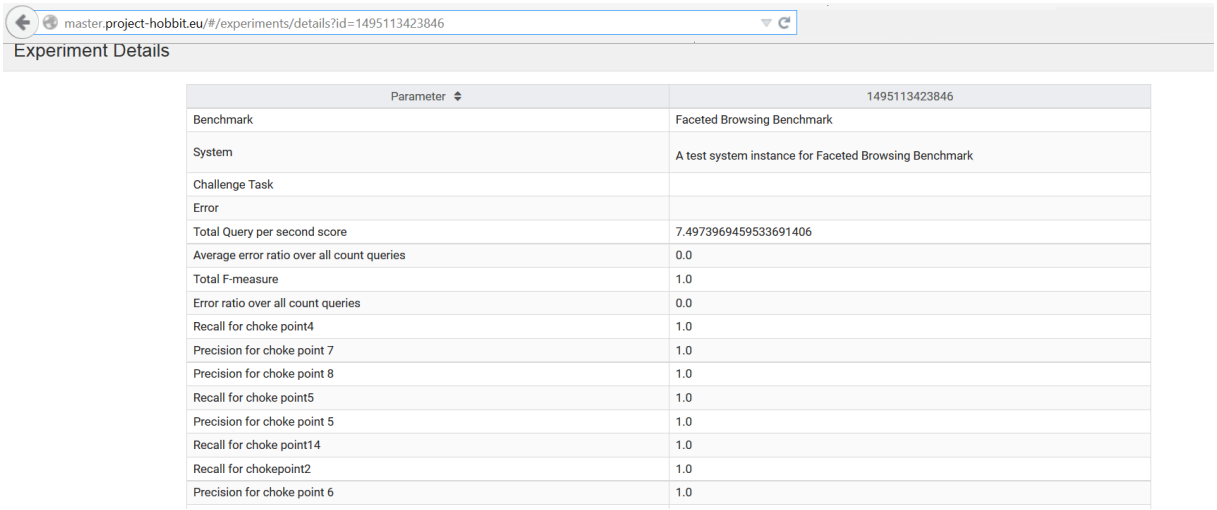
6. Results are collected and saved to the evaluation storage, which ends the run of the benchmark.

5 Experiment

By writing a system adapter for the open source system tenforce/virtuoso[9], we are able to provide results for a baseline system. Figure 4 shows parts of the result on the platform. With no surprise we see that virtuoso scores 1.0 for precision, recall and F1-measure for every choke point, and makes no error on the count queries. This happens not only because of the underlying dataset being rather

[9] https://github.com/tenforce/docker-virtuoso

small (60MB, approx 1 million triples) but, most importantly, because we are comparing the results of our gold standard system to itself.



Parameter	1495113423846
Benchmark	Faceted Browsing Benchmark
System	A test system instance for Faceted Browsing Benchmark
Challenge Task	
Error	
Total Query per second score	7.4973969459533691406
Average error ratio over all count queries	0.0
Total F-measure	1.0
Error ratio over all count queries	0.0
Recall for choke point4	1.0
Precision for choke point 7	1.0
Precision for choke point 8	1.0
Recall for choke point5	1.0
Precision for choke point 5	1.0
Recall for choke point14	1.0
Recall for chokepoint2	1.0
Precision for choke point 6	1.0

Figure 4: Part of the results of our baseline system "tenforce/virtuoso"

Still, the experiment on our baseline system proves functionality of our benchmark on the platform.

6 Preliminary Results

As an outcome of the Mighty Storage Challenge (MOCHA)[10] 2017 at the ESWC 2017 [11] we are able to provide preliminary benchmark results.

The benchmark on Faceted Browsing was Task 4 out of four tasks, are provided on the HOBBIT platform. Unfortunately, only two out of three participating systems were able to complete the benchmark within the given timeframe. Therefore we are left with results for two participating systems. Firstly, we have the implementation of a system adapter for Virtuoso 7.2 Open-Source Edition by OpenLink Software.Because it follows the common API of the MOCHA challenge, this system serves as our MOCHA baseline. The second contestant is the Virtuoso 8.0 Commercial Edition (beta release) by OpenLink Software.

Performance of these two systems were very similar. As one can see in Figure 5, commercial Virtuoso returned an empty result list for a very few queries. On most other queries, the commercial version was slightly faster (Figure 6 and 7b). Furthermore, commercial Virtuoso performed better on facet counts (Figure) 7a.

[10] https://project-hobbit.eu/challenges/mighty-storage-challenge
[11] http://2017.eswc-conferences.org

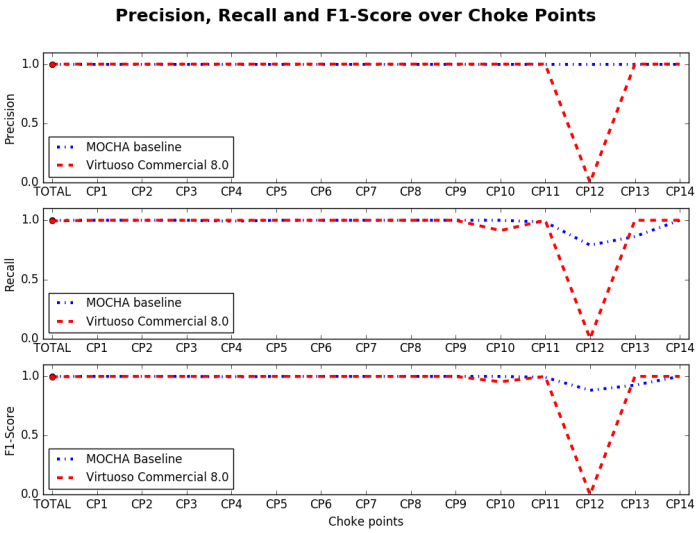


Figure 5: MOCHA results for instance retrieval: correctness

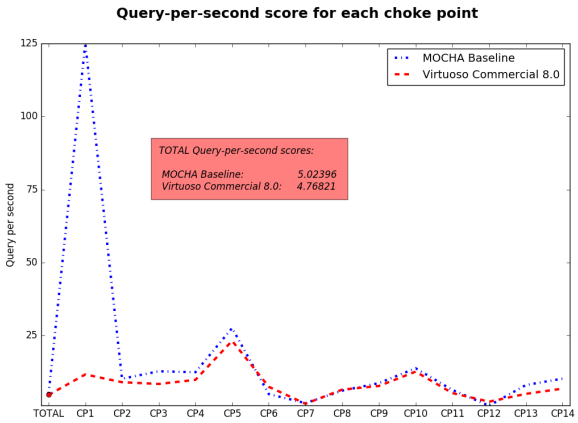


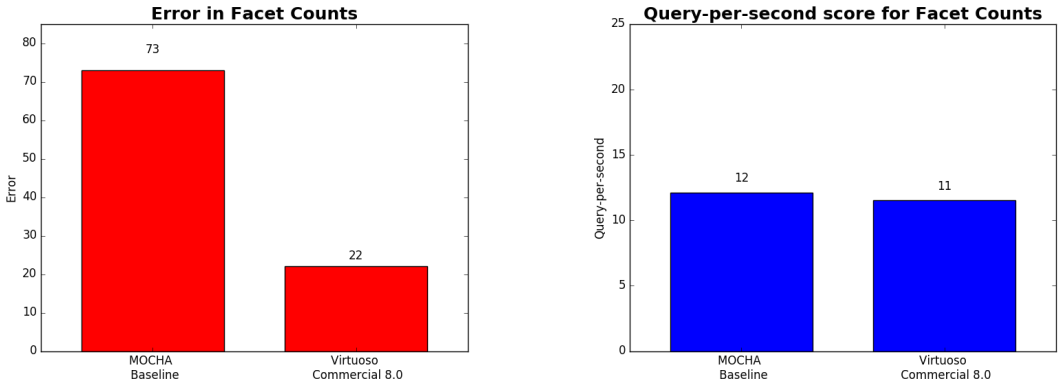
Figure 6: MOCHA results for instance retrieval: speed

7 Future work on the benchmark

Despite the successful building of a first version of a benchmark on faceted browsing, there are a few shortcomings we would like to address. Currently, the underlying dataset used for the benchmark has only the size of 60MB (approx 1 million triples) due to technicalities of the platform that do not allow the use of larger datasets at the time of writing. Since feature extension and improvement of the HOBBIT platform are ongoing projects, these technicalities should be resolved in the very near future. Moreover, since the dataset is generated by a mimicking algorithm, the scaling of the dataset is simply a matter of changing parameters when running the generator.
We are therefore confident that it is just a matter of time (measured in weeks rather than months) until the benchmark is running on a larger dataset.

Another unpleasant state is the lack of meaningful benchmark results on a plenitude of systems. Initially, we were hoping that a detailed specification of choke points would inspire ingenuity in participants to develop targeted strategies to master the difficulties found in faceted browsing. Unfortunately,

Figure 7: MOCHA results for facet counts



(a) Correctness (b) Speed

we cannot assess the effort of participants toward enabling their system to support faceted browsing from the given information. For example, it remains unclear to us how underlying databases are structured to increase performance on specific choke points of faceted browsing. Therefore, and also because of the small underlying datasets used in the benchmark that do not necessitate browsing-specific strategies, it seems more plausible that the complexity of the query itself determines the performance, rather than the different types of transitions from one state to the other that make up the choke points. (By the complexity of the query we mean the subgraph that is determined by the bindings within the SPARQL queries and the SPARQL operators involved in the query.) However, the goal to measure performance with respect to query complexity not the goal of our benchmark and this would require measuring entirely different KPIs related to different choke points. We therefore suggest to sceptically interpret choke point related results as an actual performance measure, at least while the underlying dataset is still relatively small.
Still, our benchmark provides the possibility for systems to test improvements on specific faceted browsing choke points as the outcome of changes to their system's implementation.

As stipulated above, the increase of the size of the underlying dataset would make a natural next step. Datasets of different size have been created and will be tested to be integrated into the benchmark. This integration encompasses more work than simply the exchange of a file. It also needs to be checked and guaranteed that the integrated virtuoso instance computing the gold standard does indeed return faithfully correct results on such larger datasets. A different next step would be a second check for possibilities to include other systems in the computation of the gold standard. It is clearly a suboptimal state to have only one system computing the gold standard. We would like to be able to rely on the gold standard without any feeling of unease, also without additional manual checks after increasing or changing the underlying dataset.

Momentarily, the dataset is generated once by using a generator. Subsequently, it is manually added into the data-generating module of the benchmark (which sends the data to a participating system) and into the docker image of the gold standard virtuoso instance. Hence, the dataset is fixed for a current version of the benchmark. This would ideally be changed to generating the dataset on the fly. The seed (and possibly other parameters like the number of train connections) for the generation of a new dataset could be changed by the user on the platform. This would enable a user of the platform to quickly test the performance of a system on different dataset sizes.

Obviously, we would also like to rejoice seeing the outcome of our work. That is, we would like

to see more results of systems working through our benchmark scenarios. For this to happen, we need to write for each system its system adapter to attach it to our benchmark. Coding additional system adapters for different systems therefore adds another next step of our future work on the benchmark.