Unedited[1] Report of the DKG COST Action Hackathon “Querying Federations of Knowledge Graphs”

https://github.com/MaastrichtU-IDS/federatedQueryKG

Use case Provider 1: E-learning use case

Brief summary of the use case:

In recent years, Learning Management Systems (LMSs) have become increasingly important in online education due to their flexible integration platforms for organizing vast learning resources and establishing effective communication channels between teachers and learners. These online platforms are attracting a growing number of users who continually access, download/upload resources and interact with each other during their teaching/learning processes. The outbreak of COVID-19 has accelerated this trend.
In this context, academic institutions are generating large volumes of learning-related data that can be analyzed to support teachers in lesson planning, course and faculty degree planning, and university strategic-level administration. However, managing such a significant amount of data, often from multiple heterogeneous sources and with attributes that sometimes reflect semantic inconsistencies, poses an emerging challenge.
The primary aim of this use case is to provide artificial intelligence algorithms with the ability to analyze implicit interaction patterns within LMSs registered by a specific e-learning community. To achieve this, we require federated SPARQL queries to obtain data on student interactions and academic performance efficiently, which can be used to feed predictive models and visualizations.
The datasets presented in this use case are Open University datasets (OULAD) and Malaga University Dataset (MUD), which contain anonymized data about student interactions with the Virtual Learning Environment (VLE) and other demographic information. These datasets are stored in five RDF repositories built with two OWL ontologies. Federated SPARQL queries are needed to obtain distributed data for predictive models and visualizations. A machine learning algorithm that predicts student grades requires a dataframe with specific features such as user_id, course_id, diff_days, weight_score, num_submissions, and sum_click.

Observations:

This use case is a real-world application with real data. Datasets contain data about the interaction of students from the Software Engineering degree at the University of Málaga (UMA), Spain. The researcher of the Khaos Research Group implemented the analytic tool

[1] The current version of this document is a collection of texts written by the participants at the end of the hackathon. Creating an edited version is work in progress.

in a research project. UMA data was enriched with Open University data, available in a public repository.
The domain of e-learning is a familiar domain for workshop participants. Terms in the domain are courses, students, assessment, etc.
Only five datasets contain the information to resolve the needed queries, and each query needs two datasets to be resolved. Therefore, finding the dataset for a subset of triples could not be a real challenge. However, the number of triples in the datasets is large (of the order of millions of triples). The following table presents some statistics of our EndPoints during the workshop, such as the number of triples and network usage.

SPARQL Endpoint	Nº of triples	Network usage
https://student-oulad.khaos.uma.es/sparql	20.238.494	12,4 GB
https://module-oulad.khaos.uma.es/sparql	34.279	3 GB
https://assignment-mud.khaos.uma.es/sparql	325.600	25,8 GB
https://user-mud.khaos.uma.es/sparql	59.634	14,6 GB
https://log-mud.khaos.uma.es/sparql	137.133.001	158 GB

Our queries are complex SPARQL queries, including group by, count, functions, etc.
The challenge of this use case was the creation of federated SPARQL queries in the e-learning domain, having only an ontology and a brief description of the dataset content. However, the tools aimed to execute the queries without specifying a SERVICE clause, i.e., federate the queries automatically. Therefore, participants in our use case had two different profiles: Non-expert users aiming to find a solution for the query and tools providers seeking a way to split the triple patterns across the endpoints automatically.
Before the workshop, we had solved the complete use case using the SERVICE clause (indicating the corresponding EndPoint manually) in a Virtuoso EndPoint.

Lesson Learnt and Results:

Regarding scalability, Metaphactory and CostFed can execute all the queries without using the SERVICE clause. Other tools present problems with the size of the datasets because they generate intermediate results and then perform operations in the main memory. As the number of triples in these intermediate results can be huge, operations on these results are memory-intensive.
There needs to be more than the expressivity supported by the tools. More than standard SPARQL is required to solve the queries. In real cases, complex handling of dates is often necessary. The query complexity is not a problem directly related to query federation, but it is essential to consider it when implementing a federated query engine.

Another important issue is that providing only an ontology file is insufficient for non-experts users. Ontologies must be well documented to help non-expert users, and the description of the dataset must include the necessary metadata to understand its content. Ontologies and metadata should be helpful for both humans and tools. In the case of tools, metadata indicating links with other datasets, if known, should be available through the EndPoint. Finally, non-expert users require advanced SPARQL knowledge. Tools providers could provide graphical user interfaces to assist the query creation process.
Finally, for expert users (as we are) familiar with the ontologies and the datasets, solving queries is easier with Metaphactory than with Virtuoso because you can focus on queries, not in the federation. Furthermore, Metaphactory includes a graphical interface that allows one to easily navigate the ontology, finding the classes and properties to use in the queries without paying attention to the knowledge graph that contains the data. We could not test the usability of CostFed.

Conclusion and Next steps:

This use case aims not only to test several federated tools from the point of view of their capabilities to solve federated queries transparently but also to bring to the fore problems related to the scalability of real-size datasets, the complexity of the queries and the necessary metadata to develop analytics application on top of distributed knowledge graphs.

In future work, we will refactor ontologies to make them more understandable for non-expert users, improving documentation and providing more people-oriented names for classes and properties.
In addition, we will study the possibility of including new metadata to facilitate the work of federated tools. For example, we will add to the EndPoints the known mappings between our EndPoints and mappings to external ones.
Finally, we will study how to provide support information for constructing queries by non-expert users, beyond the information in the ontologies and metadata, such as query patterns, basic queries, etc.

Use case Provider 2:
Brief summary of the use case:

The main objective of the FedShop use-case is to evaluate how the performances of federated query engines evolve when the number of members in the federation grows

The FedShop use-case simulates a customer browsing a virtual shop made of many single shops. Browsing is done through SPARQL queries retrieving products according to some criteria. Once products are found, customers aim to find deeper information about products, compare products, find similar products, and find reviews on products. This corresponds to the explore use-case of the Berlin Benchmark (BSBM) but in a federated context.

The FedShop use-case naturally grows by increasing the number of shops in the virtual shop. For the Cost Hackathon, FedShop comes in 2 configurations: 20 endpoints with 10 shops and 10 rating sites, and 100 endpoints composed of 50 shops and 50 rating sites. The 20 endpoints configuration contains almost 6M triples while the 100 endpoints configuration contains almost 28M triples.

The FedShop use-case contains 12 queries simulating a customer navigating the federation of shops as if it was a single shop.

To evaluate the evolution of performances, the Federated query engines have to execute the 12 queries on the configuration of 20 endpoints and report execution time. Next, they re-execute the same 12 queries on the configuration of 100 endpoints and observe execution time again. As the size of the federation has been multiplied by 5, it is possible to estimate if execution times of federated query engines remain stable or increase more than 5 or less than 5.

Observations:

The experiment involved FedX, Comunica, CostFed, Semagrow and Colchain. We detail in figure FS1 below the result of FedX that should be considered as the baseline:

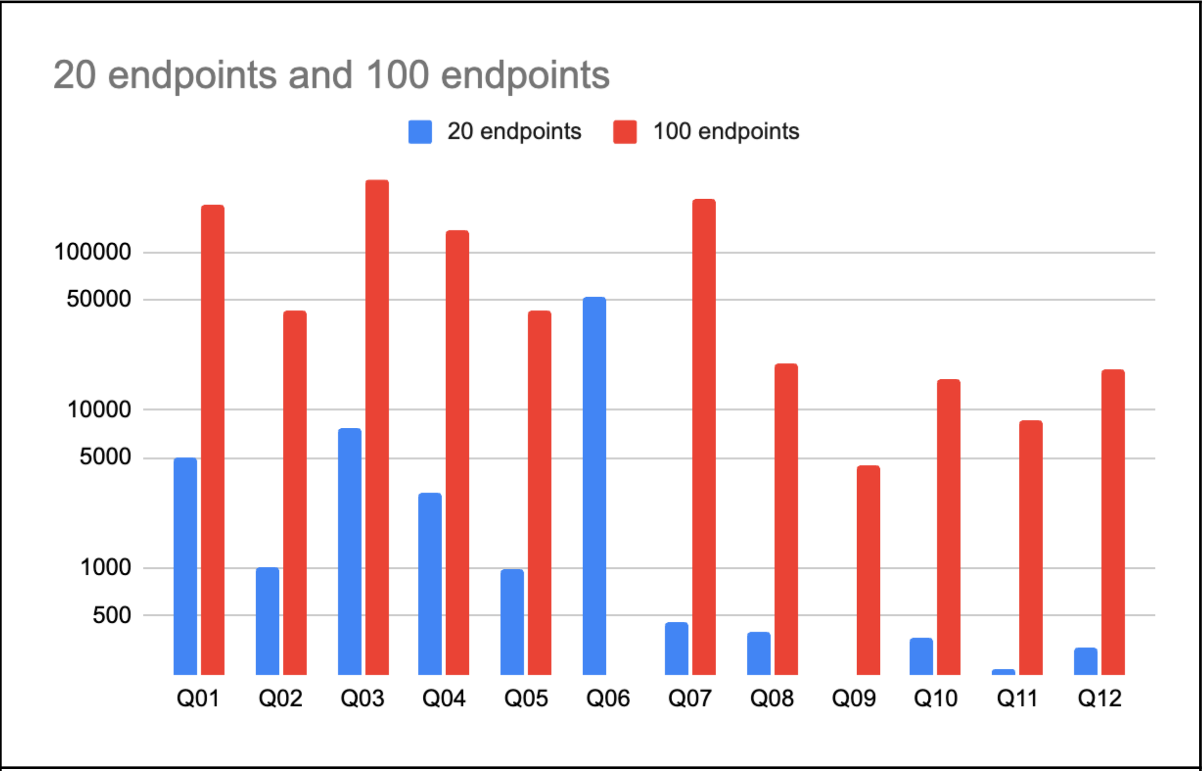


Fig FS1: Execution of the 12 queries by FedX on the 2 configurations 20 and 100 endpoints. Q06 failed in the 100 endpoints configuration.

As we can see, fedX poorly scales with execution times multiplied by 15 on average.

Overall we obtain the average execution time of a query in the workload of 12 queries as described in the following table:

	20 endpoints	**100 endpoints**
Ideal	51 (ms)	81ms
FedX	5 s	125 s
CostFed	86 s	155s
ColChain	6 s	126 s
Comunica	11 s	N/A
Semagrow	N/A	N/A

First, as execution times have been obtained on different computers, it has no sense to compare different engines, only the evolution of execution time between the 2 configurations is meaningful.

The “Ideal” line corresponds to hand-crafted SPARQL 1.1 queries with service clauses where the minimal source selection has been precomputed. Only the source selection is minimal, the execution plan of the SPARQL 1.1 query can be suboptimal. Concerning the evolution of the Ideal execution time, it is less than linear ie. execution time increased by less than a factor 2.

CostFed and Semagrow did not succeed to run the twelve queries in any configuration. This comes from difficulties when creating summaries. CostFed results are coming from experiments conducted before the hackathon by the use-case team.

ColChain runs almost every queries in both configurations, but some queries time-out. We considered a default time-out of 600s. As we can see, the evolution of execution times is far more than linear.

Comunica succeeded in running the queries on the smaller dataset, but required optimisations to execute queries on the larger, full configuration. Measurements were not done in a controlled environment, and only a few queries were executed with the full configuration due to time limitations. Consequently, the results for the full configuration are incomplete and not comparable with the other results.

Lesson Learnt and Results:

The main result of the use-case is that the current federated query engine poorly scales on the number of federation members.

As it is the first time that these engines were exposed to a large number of endpoints, the FedShop use-case revealed new bugs in the implementations of all these engines.

The problem of generation of summaries for CostFed and Semagrow reveals the need for common practices for producing/maintaining summaries for engines that need it. We suggest that the creation of summaries should only rely on the knowledge of the URLs of Sparql endpoints of the federation member as it is the case for FedX. ColChain is not concerned as ColChain is designed as a distributed system.

Conclusion and Next steps:

The FedShop use-case revealed the lack of scalability of tested federated query engines. In their current form, they are not able to run a federated shop although the “ideal” SPARQL 1.1 queries demonstrate that it is feasible.
This means that there exists a large room of improvement for current federated engines to meet requirements of a FedShop use-case.

Use case Provider 3: Life Science Use case
Brief summary of the use case:
In this use case, the main objective is to rewrite and execute queries using multiple data sources in a federated manner. The queries use various SPARQL endpoints including WikiPathways, KG-hub Covid-kg, Wikidata, Bioregistry, and Bio2RDF.

This use case presents two main challenges identified in the life science domain. The first challenge is to efficiently map the same entities described by different IRIs in different sources. Query1 and Query3 were designed to match the same genes and integrate gene information from multiple endpoints. The second challenge involves investigating strategies for splitting large datasets and running federated queries on different endpoints to achieve better load balancing. To address this challenge, we have defined Query 2, which retrieves gene ontology (GO) annotations for known drug targets using three datasets of Bio2RDF, namely HGNC, GOA-human, Drugbank. The queries and additional information can be found here.

Observations:

- The use case helped to identify bugs (#8, #9, #10) in ***HeFQUIN*** pertaining to service result merging and other optimisations. These were fixed and allowed the queries to be correctly executed.
- The use case helped to improve ***SemaGrow*** tool in two aspects, 1. Update Sevod-scraper (tool for creating Semagrow metadata) for support of n-quads file format (https://github.com/semagrow/sevod-scraper/issues/18), 2. Improve Sevod-scraper for automatically finding other prefixes other than authorities.
- SemaGrow is not ideal tool for large datasets such as WikiPathways x BioLink (resources for query 2). As it requires locally store the data dump.
- A limitation of the ***ColChain*** tool was identified in the source selection stage when the query includes a triple with a variable predicate. This is explained by tool feature which lies on definition of the fragments based on predicates.

Lesson Learnt and Results:

- A mapping service (i.e. bioregistry.org/sparql) helps to identify equivalent IRIs for a resource, rather than having to rewrite an IRI into another manually. The SPARQL query is here.
- Taking first steps towards decentralization of Bio2RDF datasets using colchain
- Instructions to run and test ***Colchain*** tool for query 2 examined and documented, it can be find here
- To enhance the execution efficiency of query 2, Federated Query Processor (***Fedx***) can be utilized.
- Requirements and instructions to run ***SemaGrow*** for Bio2RDF use case(query 2),it can be find here
-

Conclusion and Next steps:

The following actions have been proposed for the advancement of ColChain deployment and the optimization of query resolution on Bio2RDF. Firstly, a follow-up discussion is suggested to explore the possibility of deploying ColChain on a larger subset of Bio2RDF. Secondly, the definition of more challenging queries is recommended to further evaluate the capabilities of the current system. Additionally, on-the-fly solutions for identifier matching could be developed to address the issue of undefined sameAs links. This may involve utilizing resources such as http://identifiers.org/ensembl/ENSG00000008710, https://identifiers.org/ensembl/ENSG00000008710, http://bio2rdf.org/ensembl/ENSG00000008710 and the value “ENSG00000008710”^^xsd:string to facilitate more efficient and accurate query resolution.

Federation Tool Provider 1: Comunica

Brief summary of the tool

Comunica is a highly customizable SPARQL meta-query engine. With Comunica, users (developers) can build a SPARQL query engine that fit their needs by selecting components, called actors that handle the different steps of query executions, from the way the data sources are accessed to the join algorithm, to give a polemic example. Developers can also intuitively create their own actors for custom handling of certain parts of the query execution process. The goal of Comunica is to hide the complexity of SPARQL querying, by automatically handling the specificity relative to the different SPARQL endpoint, Comunica will discover the type of endpoint and will automatically adapt its querying strategy based on it. Comunica is free, open source and uses an MIT licence, it is implemented using typescript and offers out of the box three interfaces, a typescript library, a command-line interface and a web client.

Comunica enables federated querying over heterogeneous interfaces, including Linked Data documents, TPF interfaces, and SPARQL endpoints. It follows a federation algorithm where it splits up the query into triple pattern queries, and sends those to sources via a bind-join approach. In order to determine the join orders, it send COUNT queries to SPARQL endpoints for cardinality estimation of triple patterns.

Aim

In this workshop, we mainly focused on two use cases; federated shop and e-learning.

The federated shop queries are quite selective, but are done over a large number of federation members. After resolving several implementation-specific bugs, we were able to execute all of the queries in this use case with a reasonable execution time. Our main difficulty for this use cases was that Comunica was sending so many SPARQL queries to the endpoints, that the endpoint machine became overloaded. This was a consequence of the fact that all 100 endpoints were virtually run on the same machine. We were able to implement a workaround for this. Since such a federation usually does not occur in a real-life setting, we have not implemented this workaround into the production version of Comunica.

The e-learning use case contained just a few SPARQL endpoints, but those queries were highly analytical, and therefore not very selective. As such, the bind-join-based triple pattern-level federation algorithm that Comunica uses resulted in a huge number of intermediary results, which made these queries very slow to execute.

Lesson Learnt and Results

Implementation-specific findings

We have discovered several implementation-specific issues, which we briefly discuss hereafter.

When we first started experimenting with the use cases, we noticed that all queries executed with very bad query plans. This was caused by the COUNT query results that are used for cardinality estimation during query planning not ending up in the right location. As such, the query planner assumed infinite estimates for all triple patterns, which made effective planning not possible.

A second issue was related to internal metadata on whether or not solution sequences could contain undefined bindings was incorrectly set when interacting with SPARQL endpoints. This caused the inefficient nested loop join algorithm to be used for nearly all join operations.

An additional issue was identified with the process of determining cardinalities for triple patterns for joining, when combined with bind join and the SPARQL federation approach of sending individual triple patterns to SPARQL endpoints. The engine uses COUNT queries to determine the cardinality of a triple pattern using an endpoint when sorting join entries, and for bind join it recursively determines the cardinalities of the remaining patterns after having resolved one of them. We made the hypothesis that caching the cardinalities of triple patterns will avoid the execution of multiple redundant COUNT queries and that those queries have a significant impact on the overall query execution time.

Another issue was encountered with the way requests are sent by the query engine, where some earlier updates to the query engine resulted in the removal of the limit on parallel requests being done to network resources. This, together with the method of processing federated SPARQL queries within Comunica, resulted in excessive amounts of simultaneous requests being sent to SPARQL endpoints depending on the query, causing the servers to either reject those requests or crash. Implementing a limit on the number of parallel requests appears to have solved this, allowing the execution of more complex queries on larger amounts of data, at the expense of slower execution for simple queries that would not have resulted in too many requests on their own.

High-level findings

Hereafter, we discuss several findings that are not specific to our implementation, but to the algorithm that we use. Therefore, these findings are generalizable across other implementations.

We hypothesised that sending a large number of COUNT queries to endpoints could overload them, causing the endpoint to have high response times. Hence, we expected some COUNT queries to take a long time to execute. Hence, adding a timeout for COUNT queries could speed up cardinality estimation, and make query planning faster. However, after implementing this, and experimenting with different timeout values, we didn't notice a significant performance difference.

We previously made the hypothesis that a consequential factor in the query execution time is the execution of COUNT query to determine the cardinality of the triple pattern. We decided to experiment with inferring the cardinality of more selective triple patterns based on the cardinality of their less selective parent (eg; <subject> <predicate> ?o is more selective than ?s <predicate> ?o), more precisely we set the cardinality of the more selective triple at a fixed value below the less selective triple pattern.
In our experiment (using this datasource [some help for the setup is available here]), the execution time was (with a sample size of 5) 1s lower compared to our implementation with only the caching which tends to imply that the count queries are not a heavily influential factor in the query execution time. We also calculated the ratio of cardinality calculation effectuated without a count query and obtained a value of 95% with this new approach compared to 86.37% with only the caching system, which tends to indicate that we can reduce the number of requests sent to the endpoint further with this heuristic. Unfortunately, we did not observe a significant performance improvement with this change.

Overall results

While we did not have the time to run all the queries associated with different use cases, the ones we experimented with worked after a number of bug fixes and optimisations in Comunica, despite being slower than dedicated SPARQL federation approaches. The execution time of the queries ranged from half a second for a trivial query in the Federated Shop use case to an estimated well over an hour in the E-Learning use case. Using SERVICE keywords to direct parts of the queries to specific endpoints greatly improved the execution times for some queries, for example from the aforementioned well over an hour to around two and a half minutes. The selection of the join implementation also has a significant impact, as discussed in association with the federation approach taken by Comunica. Thankfully, encountering the various shortcomings of the query engine in a controlled environment enabled efficient development of fixes or enhancements to address them.

Any relevant instructions

Detailed instructions on installing and using Comunica engines can be found on our website: https://comunica.dev/
A browser client is available on http://query.linkeddatafragments.org/
All changes that were implemented during this hackathon are available in https://github.com/comunica/comunica/tree/feature/optimize-sparql-fed

Conclusion and Next steps

As Comunica is being used in production environments, we aim to make the prototypical improvements we developed during this hackathon available in a production-ready manner. As such, we will be working on this in the weeks after this hackathon, and making it available in a next release of Comunica. Furthermore, we foresee several potential improvements for federated queries over SPARQL endpoints in Comunica in the future, which we will list below.

One of the queries we encountered had one selective FILTER expression that was connected to a single triple pattern. Since our algorithm only sends triple patterns to endpoints, and does FILTER operations client-side, this query had bad performance. A simple optimization here would be to push down this filter operation, and send it to the endpoint together with this triple pattern.

Comunica's federation algorithm is based on splitting up the query into triple pattern queries, and using a bind-join-based federation approach. As this could result in a larger number of intermediary results for many non-selective triple patterns, this approach will never perform well for such cases. As such, implementing an exclusive-groups-based approach like FedX would improve performance, where we would send more than just triple pattern queries to endpoints. However, since Comunica aims to focus on federations over heterogeneous interfaces, and not just SPARQL endpoints, this approach is not trivial due to different interfaces having different levels of expressivity, which would require future research.

While our current line of research is not focused on federations over SPARQL endpoints, this is of course still an important topic that could become of relevance to Comunica in the future, if more users start using and requesting this functionality. However, our research is currently focused on cases where federation members are not all known before query execution starts, and need to be discovered dynamically during query execution by the query engine. As such, our research will focus more on federation algorithms that can handle dynamic source discovery.

Federation Tool Provider 2: HeFQUIN

Brief summary of HeFQUIN

HeFQUIN is a query federation engine designed to execute SPARQL queries over *heterogeneous federations* of knowledge graphs. More precisely, the federations considered by HeFQUIN may be heterogeneous in terms of

1. data access interfaces (i.e., going beyond federations of only SPARQL endpoints),
1. vocabularies/ontologies being used in the knowledge graphs of the federation members, and
2. even the graph data models being used for the knowledge graphs (i.e., going beyond federations of only RDF-based knowledge graphs).

Code and License. HeFQUIN is free and open source software under the Apache 2 license. The source code of HeFQUIN is written in Java based on the Apache Jena framework, and it is available in the following github repository: https://github.com/LiUSemWeb/HeFQUIN/

Status. HeFQUIN is still under development. At the point of the hackathon, HeFQUIN supports heterogeneous federations with federation members that provide either a SPARQL endpoint interface, a TPF interface, or a brTPF interface. Additionally, support for GraphQL endpoints and openCypher Property Graphs is already implemented, but not yet properly integrated into the query optimization component of HeFQUIN. Similarly, support for vocabulary mappings is already implemented but also not yet considered properly during query optimization. HeFQUIN supports all features of the SPARQL query language, where the fragment of SPARQL consisting of basic graph patterns (BGPs), FILTER, UNION, and OPTIONAL is supported natively by the query engine itself and the rest of SPARQL is supported through the integration of HeFQUIN with Apache Jena. HeFQUIN does not yet have a proper source selection component, which means that queries given to HeFQUIN need to use SERVICE clauses to assign the various subqueries to the relevant federation members.

Aims for using HeFQUIN at the Hackathon

As a relatively new and not yet very mature engine, our aim during the hackathon was to try to apply HeFQUIN for the queries in each of the three use cases in order to find bugs and to identify important aspects in which HeFQUIN needs to be improved to become a mature option for such use cases. In the end, we managed to try HeFQUIN in only two of the three use cases, namely the life-science use case and the e-learning use case.

Achievements and Observations

In the life-science use case we focused on the first query which consists of two subqueries for two different SPARQL endpoints. The join between the results of the two subqueries is on IRIs of genes, where the main challenge of the query is that these IRIs are `http://` IRIs in the data of one of the endpoints whereas they are `https://` IRI in the data of the other endpoints. As a first approach, the challenge was addressed by extending the query with a

BIND clause that rewrites the IRIs and assigns the rewritten IRIs to a separate query variable which is then used in the second subquery. The so-extended query could be executed successfully by HeFQUIN.

As an alternative solution, we created a version of the query in which a separate mapping service is used as an additional endpoint to be accessed by the query. This mapping service can answer requests with `owl:sameAs` triple patterns and covers the rewriting of the `http://` IRIs as needed by the query. After some initial issues (which turned out to be limitations of the mapping service rather than issues with HeFQUIN), the alternative version of the query could be executed successfully.

Finally, we tried the vocabulary mapping feature of HeFQUIN to create a third solution. This feature requires the user to provide a vocabulary mapping file that HeFQUIN can use to rewrite subqueries (not needed in the given use case) and to rewrite solution mappings obtained from subqueries (needed in the given use case). By using this feature, the query does not need to contain a BIND clause nor does it need to contain a SERVICE clause to invoke the aforementioned mapping service for rewriting the `http://` IRIs. As a consequence, there is also no need for the aforementioned two separate query variables in this query. The third variant of the query could then be executed successfully as well.

In the e-learning use case we focus on queries 1 and 5, both of which consist of two subqueries for two different SPARQL endpoints. The main challenge in this use case is the analytical nature of the queries, for which a lot of the data from both of the relevant datasets need to be brought together. That is, for both of the queries, their respective two subqueries retrieve a high number of solutions from the corresponding two endpoints. Additionally, the join between the subqueries has a low selectivity; more precisely, each of the solutions of the first subquery has several join partners in the result of the second subquery. Due to these properties of these queries, their results consist of millions of solutions. This huge size of the final query result presented a serious issue for the HeFQUIN command line tool because, in this tool, all solutions of the query result produced by the HeFQUIN engine are collected (in main memory) to then create a Jena `ResultSet` object that can be passed to Jena to be serialized (e.g., printed on the command line or written into a result file). As more and more of the millions of result solutions were accumulated, the JVM started swapping memory and the computer on which we ran the test became unresponsive. The only way to successfully test the queries was to use a version in which the result of the first subquery is restricted to a few solutions by means of a VALUES clause.

While working on the use cases, we discovered the following bugs and issues in HeFQUIN and managed to fix most of them right away.

- Error in merging multiple endpoints for filter (issue #6)
  status: fixed by PR HeFQUIN#290
- Error in filter on merging results (issue #7)
  status: fixed by PR HeFQUIN#290
- Error due to property path pattern within SERVICE clause (issue #8)
  status: fixed by PR HeFQUIN#290
- No support for BIND in between SERVICE clauses (issue #9)
  status: PR ready for review (HeFQUIN#291)
- Issue with calling mapping service (issue #10)
  status: identified as an issue of the mapping service and not of HeFQUIN

- Vocabulary mapping based rewriting of query patterns with FILTER fails
  status: fixed (see commit)
- Exceptions thrown within the query processing machinery of Jena into which the HeFQUIN engine is integrated are not caught by the HeFQUIN CLI
  status: fixed (see commit)
- When printing out query plans, query patterns are printed using Jena's algebra representation rather than the SPARQL syntax
  status: fixed (see commit and commit)
- Filter operators are not considered when obtaining the set of variables used in subqueries
  status: fixed (see commit)
- All solutions of the query result are collected (in main memory) to produce a Jena `ResultSet` object that will then be passed to Jena to be serialized, which is a problem for queries with huge results
  status: open (fixing this issue may require some more complex changes to the implementation)

Additionally, while working on the life-science use case, we discovered three issues with the Bioregistry mapping service, which we reported (see bioregistry#802, bioregistry#803, and bioregistry#804).

Lessons Learnt

The main lesson learned was that more work and more testing is needed for HeFQUIN to be able to handle i) queries that have big intermediate results (in any possible join order!) and ii) queries that have a huge final result. Addressing the latter (queries with a huge result) requires rethinking how HeFQUIN uses the Jena machinery for result serialization and, perhaps, to replace this part of HeFQUIN with a new result serialization component.

… TODO (is a federation approach actually suited for analytical use cases such as the e-learning use case?) …

Another lesson that we learned by interacting with the use case providers is that the current documentation of HeFQUIN is absolutely insufficient for users that are not familiar with HeFQUIN. In particular, we need better documentation on how to set up HeFQUIN and how to use its command line tool. Additionally, the RDF vocabulary based on which users have to describe their federations needs to be documented, and the same holds for the way in which vocabulary mappings can be used in HeFQUIN.

Relevant Instructions

This section provides instructions for running the queries that we worked on during the hackathon. All relevant artefacts (JAR packages of HeFQUIN, configuration files, query files, etc.) are available in the Github repository of the hackathon under the `./HeFQUIN/` directory.

For the following commands to run the queries of the **life-science use case**, clone the repository and enter the `./HeFQUIN/LifeScienceUseCase/` directory.

The file Query1.rq contains the version of the query that uses the BIND clause to rewrite the `http://` IRIs into their `https://` counterparts. To run this query execute the following command.

```
java -cp
../jars/HeFQUIN-0.0.1-SNAPSHOT-ServiceClauseBasedSourcePlannerImpl-QueryOptimizer
WithoutOptimization.jar se.liu.ida.hefquin.cli.RunQueryWithoutSrcSel
--federationDescription=Federation.ttl --query=Query1.rq
```

Conclusion and Next steps:
TODO

Federation Tool Provider 3: ColChain

Code and Project: https://github.com/dkw-aau/ColChain-Java
Providers: Christian Aebeloe (caebel@cs.aau.dk) & Katja Hose (katja.hose@tuwien.ac.at)

Brief summary of ColChain

ColChain is not a native federated SPARQL engine, it rather is a Peer-to-Peer system consisting of a number of peers, each fulfilling the role as both data provider and data consumer. It allows answering SPARQL queries over the RDF datasets hosted by the participating peers. To ensure data availability despite crashes, datasets are replicated at multiple peers so that they remain available. On the one hand, this means that each peer hosts local datasets that can be queried by other peers in the system. On the other hand, each peer can issue queries that are answered over its local data as well as the data provided by other peers. To allow peers to update datasets in a consensus-based fashion, peers form communities, each responsible for maintaining and providing access to a set of datasets/graphs. In doing so, communities can vote on updates to a graph and by keeping track of the history of changes, ColChain can also support time-travel queries and answer SPARQL queries over past versions of the data.

ColChain uses *Prefix-Partitioned Bloom Filter* indexes (i.e., summaries of the subjects/objects in a fragment) for source selection: triple patterns are matched to data fragments based on whether or not the non-variable values in the triple patterns are part of the summary. Furthermore, non-matching fragments (and their sources) are pruned by checking the overlap of the summaries for fragments that are relevant for joining triple patterns.

ColChain can be configured to resemble a federated setup: each peer forms its own community with its own datasets (corresponding to endpoints in the traditional sense) and another peer (corresponding to the federation engine) has connections to the other peers and issues SPARQL queries.

Aims of using ColChain at the Hackathon

During the workshop, we worked on use cases 2 (Federated Shop) and 3 (Life Science).

In use case 2 (Federated Shop), our aim was to assess the scalability of ColChain as the number of sources increases. This was an interesting use case since, although we tested with numbers of peers beyond 100, we had not yet tested how ColChain performs with 100 distinct datasets. In addition, the queries provided with the use are quite selective and contain several operators that could be challenging for Colchain. As such, the FedShop use case provided an opportunity to find points of future improvement for ColChain by letting us test the system in a new scenario.

In use case 3 (Life Science), our aim was to assess the performance of ColChain in a situation where queries have very large intermediate results, as for example in the Bio2RDF dataset and the corresponding use case queries. We wanted to assess the impact of such queries on the query performance and network overhead. We were also able to test ColChain in a scenario where the group members set up a local P2P network on their computers, connecting to one another's nodes, each member simulating a data provider and uploading a specific dataset to the network. This was a slightly different scenario than our previous experiments, where we created the experimental setup on a server running 128 interconnected nodes. This provided us with the opportunity to do usability testing and to improve our documentation and UI. Furthermore, one of the queries provided in the use case uses a variable in the predicate position of a triple pattern, which could be a problem for the query optimizer, which we wanted to assess.

Lesson(s) Learnt and Results

General findings and observations

Our findings were similar to our initial expectations before the Hackathon started. In summary (more details provided below), we found that very selective queries (e.g., with constant subject and object values) work very well for ColChain, since the source selection algorithm combined with the indexes built by the system was able to select the specific sources needed for the subqueries. This is possible because the indexes do not only contain high-level metadata and statistics but also capture the URLs of the triples' subjects and objects in bitvectors; their overlap helps identify which combinations of sources can produce join results and, more importantly, which ones cannot so that they can be pruned from consideration.

This was especially beneficial in the FedShop use case, where the URIs have different prefixes depending on which specific vendor or rating site contains the data. This allowed ColChain to efficiently prune most unneeded sources.
On the other hand, we also (expectedly) found that queries with a large number of intermediate results caused a significant network overhead. For instance, one specific query from the FedShop use case caused almost 500,000 requests, and one of the Bio2RDF queries that we executed caused more than 10,000 requests - which represents a significant network overhead. In general, we found that such overheads were either caused by the use of FILTER clauses in the queries, which ColChain does not (yet) optimize for, or very non-selective triple patterns in the queries.

For use case 2 (Federated Shop), we obtained the following results (timeout 30 minutes) running the 20 or 100 (depending on the setup) nodes on a server located on a different network than the Hackathon venue (details on the servers below):

- *20 sources (20 concurrent nodes on the following server: Dell R6415, 256GB RAM, 16 cores (AMD 7281), 240GB SSD, 8TB HDD, 2x25 Gbit):*

Query	**Execution time (ms)**	**Number of transferred bytes**	**Number of requests**
q01	2582	14,153,416	3,021
q02	103	451,379	150
q03	37982	338,840,909	66,594

q04	1004	5,207,508	1,191
q05	*Timeout*	1,527,829,279	496,053
q06	15860	76,463,998	17,362
q07	768	5,046,935	801
q08	49	258,780	90
q09	3	8,653	3
q10	26	343,690	122
q11	25	95,631	38
q12	12	64,215	23

- *100 sources (100 concurrent nodes on the following server: Dell R7425, 2TB (32x64GB 2666 MHz) RAM, 64 cores (2x AMD 7551 2551 MHz), 240GB SSD, 10TB HDD (7200 rpm), 2x25 Gbit):*

Query	**Execution time (ms)**	**Number of transferred bytes**	**Number of requests**
q01	271592	644,201,693	80,741
q02	473	693,225	210
q03	*Timeout*	2,854,042,386	188,122
q04	917683	481,597,951	60,698
q05	*Timeout*	2,373,426,959	763,730
q06	86199	205,736,688	21,177
q07	31363	46,962,970	3,435
q08	256	903,692	313
q09	2	2,302	1
q10	185	832,033	301
q11	54	40,879	19
q12	12	107,442	14

During the work on the use cases, we discussed the following ideas for improvement, and we will have follow-up meetings with the providers of use case 3 (Life Science) to improve the system.

Query optimizer

- Implementing different query processing strategies. For instance, in the Bio2RDF query, iit could be more efficient to download the relevant fragments as a pre-processing step, and then execute the query locally. In fact, during the work on the use case, we tried to process the query while all the data was replicated locally, and it took just a few seconds.
- Dynamic/automated choice of query processing strategies based on statistics, metadata, and indexes incl. cardinality estimations, query patterns, etc.
- Rewriting FILTER and UNION queries to execute such queries more efficiently
- Splitting up query delegations between nodes that have the same relevant data
- Compression of intermediate results

User Interface

- Minor UI fixes to increase usability of the system, such as making sure all fields are visible on any browser (style fixes)
- Creating a "locked" page for node administration, and a separate, open page for executing queries
- Minor UI tweaks, such as showing fewer fragments in the overview in the UI.

Other improvements

- Creating a Docker image for streamlining running ColChain
- Adding support for similar entities describing the same things (e.g., 'http' vs. 'https')

Any relevant instructions

The following instructions were crafted in collaboration with the use case providers.
Instructions for setting up ColChain with the FedShop use case (use case 3):

- **Download the use case data file (batch.nq)** from https://github.com/MaastrichtU-IDS/federatedQueryKG
- **Split up the batch.nq file** into the 20 or 100 sources
- **Convert** the .nq files into .hdt files using the rdf2hdt.sh tool available on the following link: *https://storage.googleapis.com/google-code-archive-downloads/v2/code.google.com/hdt-java/hdt-java-rc2.tgz*
- **Start the configuration** using the following scripts from https://github.com/ColChain/ColChain-Experiments (*scripts* directory)
    - Use the start.sh script to start the nodes
    - Use the setup.sh script to load data and indexes
- **1 node per vendor/rating site**
    - Each node creates one community and uploads one dataset to that community
    - All node observes the other communities

- **Run the queries** using the “/experiments” endpoint on the node that executes the queries (e.g., call the following URI):
    - http://<ip>:3000/experiments?mode=performance&queries=queries&out=results_100&reps=1

Instructions for running ColChain over the Bio2RDF use case:

- **Download the data as a zip file called 'colchain.zip' from the GitHub at https://github.com/MaastrichtU-IDS/federatedQueryKG/tree/main/ColChain**
- **Change the directory**
  cd jetty-base
- **Run the following command**
  java -jar ../jetty-home/start.jar
- **Navigate to the UI** by calling the address ‘http://<ip>:8080’ and **upload the config.json file** included in the ‘colchain.zip’ file above
- **Create a community and upload the dataset (in the UI)**
- To upload dataset you should notice that a compressed version of dataset with .hdt format is needed. for our use case we first downloaded Bio2RDF from Zenodo and then used HDT tool converter as follows: *Bio2RDF dataset at Zenodo: https://zenodo.org/record/3770918#.Y71gq-zMJA1 HDT tools and be downloaded from here: https://storage.googleapis.com/google-code-archive-downloads/v2/code.google.com/hdt-java/hdt-java-rc2.tgz*
- **Command for converting a file into HDT:**
  for Mac/Linux: ./rdf2hdt.sh <file.nq> <out.hdt>
  for Windows: ./rdf2hdt.bat <file.nq> <out.hdt>
- *If you get a memory error, increase the heap memory by changing from -Xmx1024M to a larger number in the file “rdf2hdt.sh”*
- **To connect to other communities**, scroll to the bottom of the front page in the UI and search Community by IP address (for example http://192.168.0.100:8080)
- *The ‘participate’ button in the UI that appears after searching for communities (above step) will add the newly created peer (set up in the previous steps) to a selected community and therefore download the triples and indexes into the local datastore. The ‘observe’ button will add the created peer as an observer to the network and will therefore only download the relevant indexes.*

Conclusion and Next steps

The Hackathon has provided great insight into the operating principles of ColChain; both in those cases where ColChain works well and in those with room for improvement. In particular, we found that very selective queries (specifically in the FedShop use case) are very efficiently processed by ColChain since the source selection algorithm is able to prune most unneeded fragments from the query plan. On the other hand, ColChain tends to struggle with queries that have a large number of intermediate results incl. those with FILTER clauses, where due to a lack of optimizational support excessive network usage is needed to answer such queries. This was especially the case for one of the FedShop queries and the tested Bio2RDF query.

During the work with the use case providers, we have identified several aspects to improve (see the ‘Lesson(s) Learnt and Results’ above). As the next steps, we will implement these ideas and assess whether they have a positive impact on the performance for the specific queries that ColChain had difficulties with.

Federation Tool Provider 4: metaphactory (FedX)

Providers: Andreas Schwarte (metaphacts), Peter Haase (metaphacts)

Brief summary of metaphactory

metaphactory is a Knowledge Graph Platform that supports collaborative knowledge modeling and knowledge generation. It abstracts from the underlying complexity of a Knowledge Graph and allows users to consume data intuitively and in context. A low-code approach allows building custom interfaces and applications that enable end-users to interact with the Knowledge Graph.

For interacting with the Knowledge Graph, metaphactory provides a *Data Access Infrastructure* allowing a unified view on distributed and heterogeneous data sources. This includes access to graph databases, relational databases, REST APIs and machine learning algorithms. Additionally, it integrates technologies for federating over multiple data sources: The FedX federation engine allows for transparent federation over a set of SPARQL 1.1 endpoints, where the data appears as a single virtual endpoint. For hybrid use-cases the Ephedra engine can be used: data sources and processing services are wrapped in “virtual” RDF4J repositories, which transform graph patterns into API calls. Services are declaratively described in a service registry and mappings are used to extract input and output parameters. While the FedX federation engine is available in RDF4J, the Ephedra engine is an extension to RDF4J provided in metaphactory.

Aim of using metaphactory

In the Hackathon we specifically made use of the federation technologies of metaphactory. We focussed on the first use-case around E-Learning and contributed also to the Life Science Use-Case.

Our goal for the E-Learning Use-Case was to show that by declaratively configuring the data sources and the FedX federation in metaphactory, we are able to provide a virtual SPARQL endpoint to train the E-Learning mode through federated queries. With the help of the built-in ontology visualization and metaphactory’s support in query formulation & execution, we wanted to write the required queries for the use-case and expose them as first-order entities.

For the Life-Science Use-Case our goal was to validate the execution of federated queries over Bio2RDF data with regard to correctness and performance. Also here we wanted to show that the FedX federation can be configured declaratively in metaphactory, and moreover that the respective queries can be executed in metaphactory.

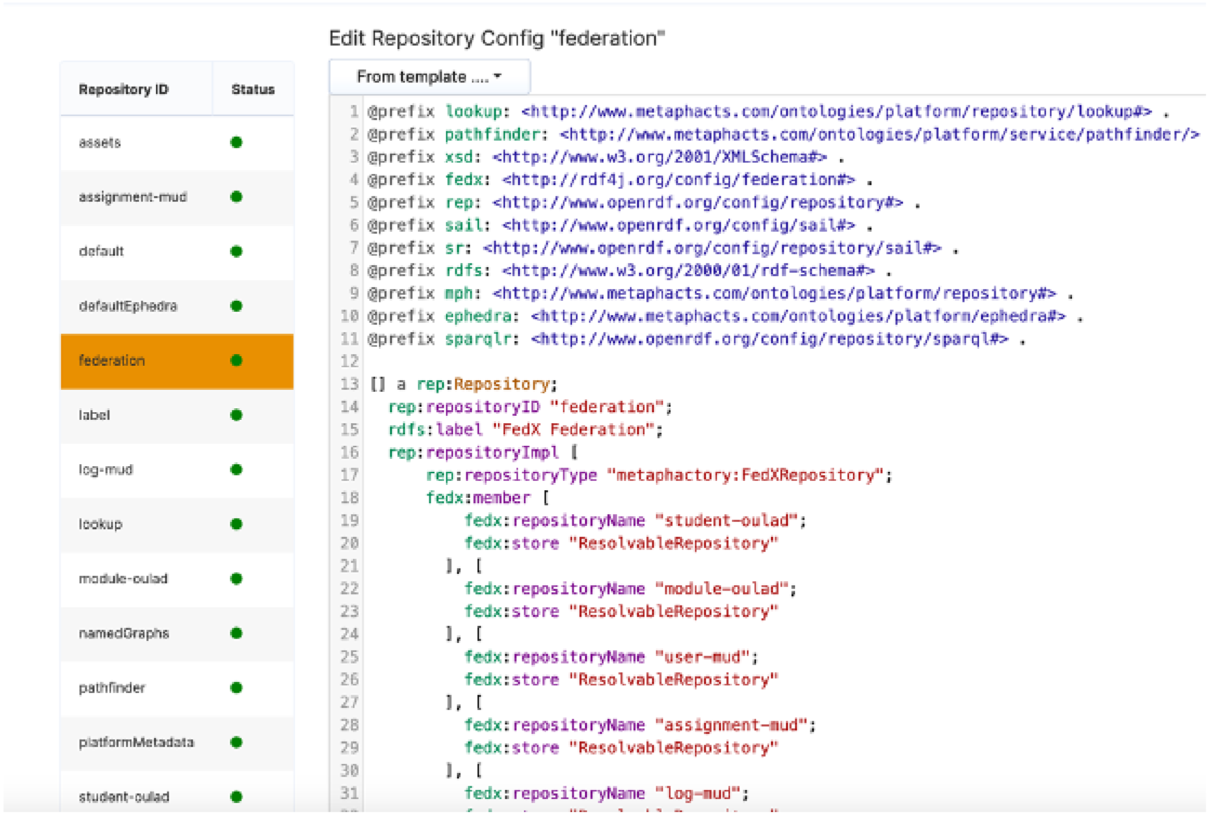
Lessons Learnt and Results

For the Hackathon we have prepared a metaphactory installation in the AWS cloud. We shared access to the machine with other interested parties and collaboratively worked on the use-cases.

Results of the E-Learning Use-Case

As a preparation the respective data sources have been configured as repositories in metaphactory. The use-case providers offered two Virtuoso SPARQL endpoints for the OULAD datasets, and three Virtuoso endpoints for the MUD datasets.

All endpoints have been configured as members to the FedX federation.

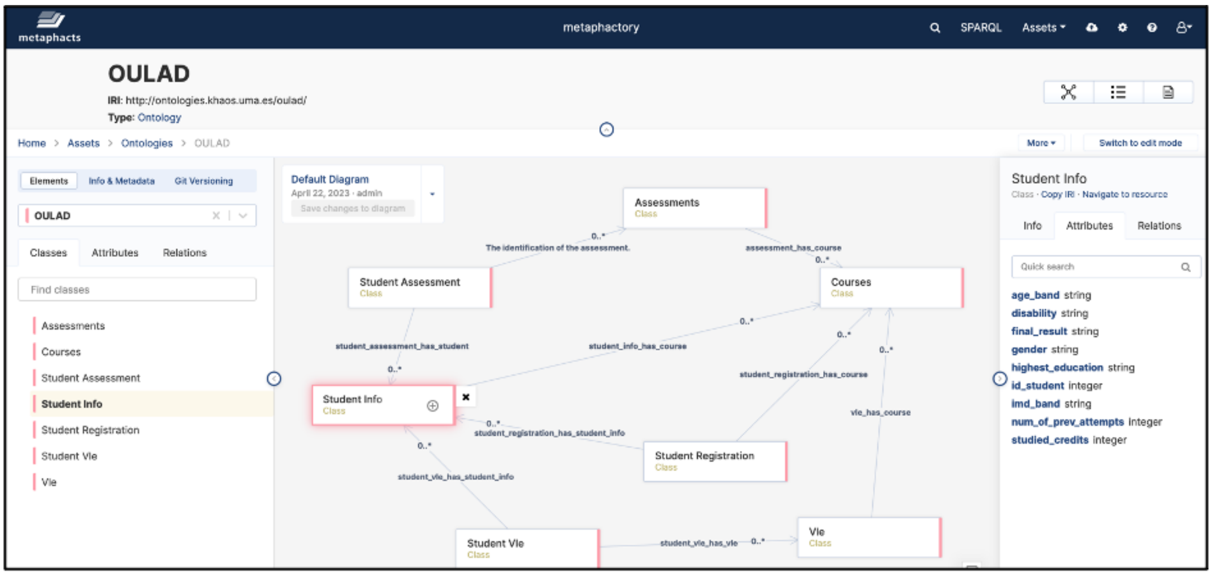


Repository ID	Status
assets	●
assignment-mud	●
default	●
defaultEphedra	●
federation	●
label	●
log-mud	●
lookup	●
module-oulad	●
namedGraphs	●
pathfinder	●
platformMetadata	●
student-oulad	●

Edit Repository Config "federation"

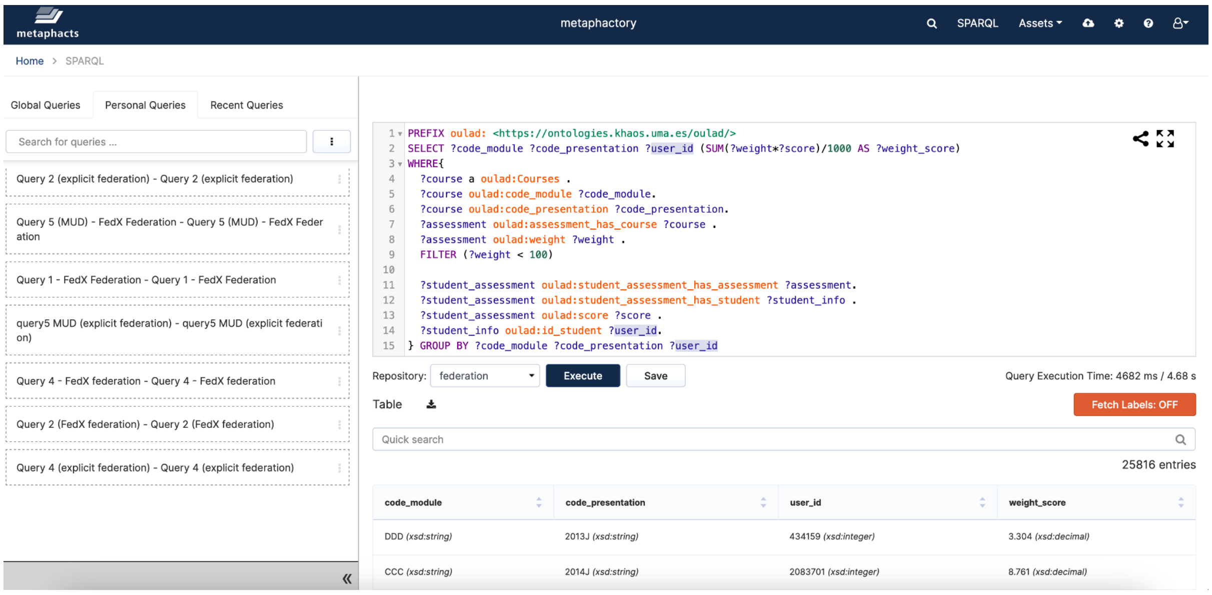
```
@prefix lookup: <http://www.metaphacts.com/ontologies/platform/repository/lookup#> .
@prefix pathfinder: <http://www.metaphacts.com/ontologies/platform/service/pathfinder/>
@prefix xsd: <http://www.w3.org/2001/XMLSchema#> .
@prefix fedx: <http://rdf4j.org/config/federation#> .
@prefix rep: <http://www.openrdf.org/config/repository#> .
@prefix sail: <http://www.openrdf.org/config/sail#> .
@prefix sr: <http://www.openrdf.org/config/repository/sail#> .
@prefix rdfs: <http://www.w3.org/2000/01/rdf-schema#> .
@prefix mph: <http://www.metaphacts.com/ontologies/platform/repository#> .
@prefix ephedra: <http://www.metaphacts.com/ontologies/platform/ephedra#> .
@prefix sparqlr: <http://www.openrdf.org/config/repository/sparql#> .

[] a rep:Repository;
  rep:repositoryID "federation";
  rdfs:label "FedX Federation";
  rep:repositoryImpl [
      rep:repositoryType "metaphactory:FedXRepository";
      fedx:member [
          fedx:repositoryName "student-oulad";
          fedx:store "ResolvableRepository"
        ], [
          fedx:repositoryName "module-oulad";
          fedx:store "ResolvableRepository"
        ], [
          fedx:repositoryName "user-mud";
          fedx:store "ResolvableRepository"
        ], [
          fedx:repositoryName "assignment-mud";
          fedx:store "ResolvableRepository"
        ], [
          fedx:repositoryName "log-mud";
```

To better understand the data, we have also loaded the OULAD and e-lion ontologies into metaphactory and used its visualization capabilities.



The ontology editor captures the classes available in the data, as well as the relations and attributes for each of them. Having a visual depiction of the ontology greatly helped us in understanding the data, and thereby formulating the required queries for training the model.

The use-cases providers have provided the body of the queries, as well as a textual description of the information needs. In metaphactory, we used the SPARQL interface to formulate the respective queries targeting the federation.



Query 2 (explicit federation) - Query 2 (explicit federation)
Query 5 (MUD) - FedX Federation - Query 5 (MUD) - FedX Federation
Query 1 - FedX Federation - Query 1 - FedX Federation
query5 MUD (explicit federation) - query5 MUD (explicit federation)
Query 4 - FedX federation - Query 4 - FedX federation
Query 2 (FedX federation) - Query 2 (FedX federation)
Query 4 (explicit federation) - Query 4 (explicit federation)

```
PREFIX oulad: <https://ontologies.khaos.uma.es/oulad/>
SELECT ?code_module ?code_presentation ?user_id (SUM(?weight*?score)/1000 AS ?weight_score)
WHERE{
  ?course a oulad:Courses .
  ?course oulad:code_module ?code_module.
  ?course oulad:code_presentation ?code_presentation.
  ?assessment oulad:assessment_has_course ?course .
  ?assessment oulad:weight ?weight .
  FILTER (?weight < 100)

  ?student_assessment oulad:student_assessment_has_assessment ?assessment.
  ?student_assessment oulad:student_assessment_has_student ?student_info .
  ?student_assessment oulad:score ?score .
  ?student_info oulad:id_student ?user_id.
} GROUP BY ?code_module ?code_presentation ?user_id
```

Repository: federation | Execute | Save | Query Execution Time: 4682 ms / 4.68 s

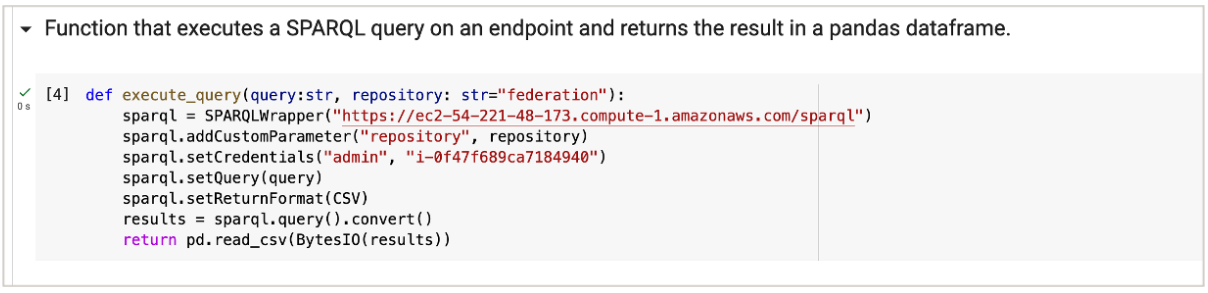
Table | Fetch Labels: OFF

Quick search

25816 entries

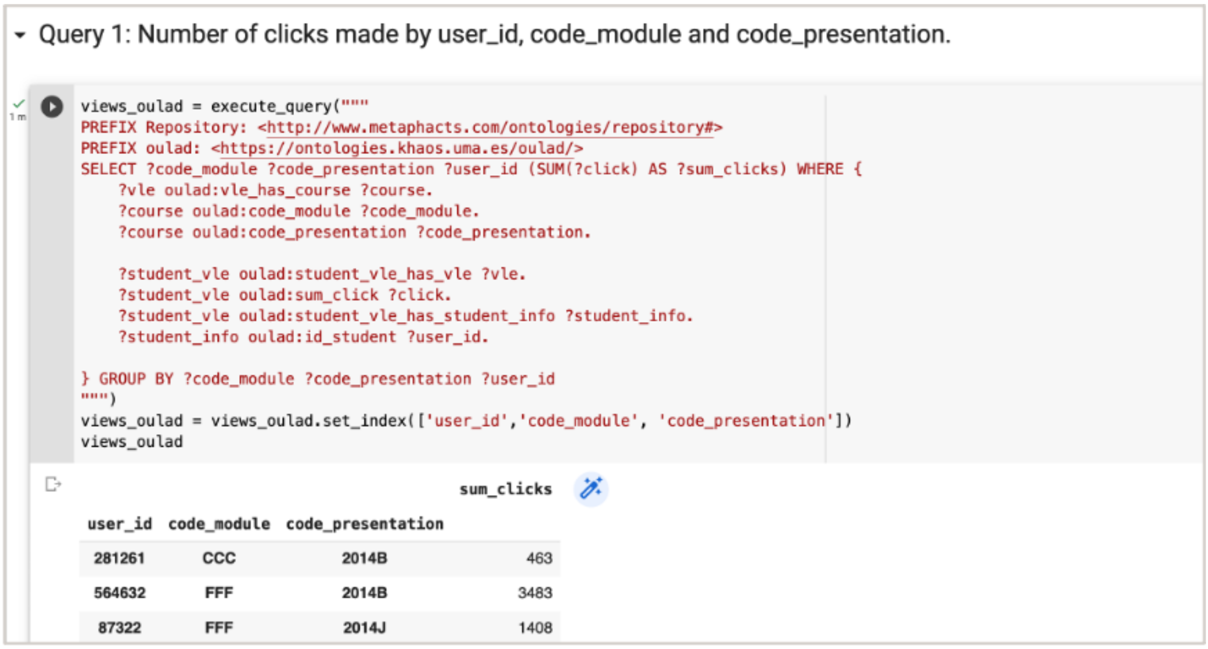
code_module	code_presentation	user_id	weight_score
DDD (xsd:string)	2013J (xsd:string)	434159 (xsd:integer)	3.304 (xsd:decimal)
CCC (xsd:string)	2014J (xsd:string)	2083701 (xsd:integer)	8.761 (xsd:decimal)

Once all queries have been formulated, we entered them into the Python playbook prepared by the use-case providers. Additionally, we adjusted the playbook script to connect to FedX federation exposed through metaphactory SPARQL endpoint.



Function that executes a SPARQL query on an endpoint and returns the result in a pandas dataframe.

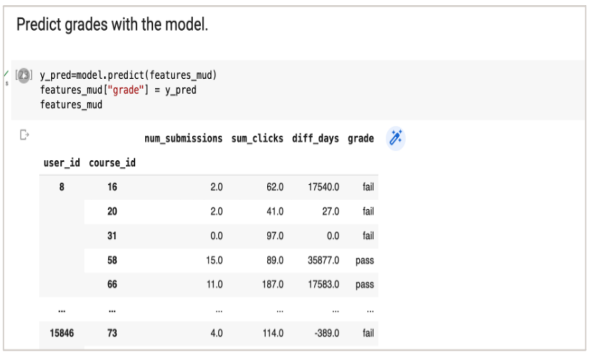
```
[4] def execute_query(query:str, repository: str="federation"):
        sparql = SPARQLWrapper("https://ec2-54-221-48-173.compute-1.amazonaws.com/sparql")
        sparql.addCustomParameter("repository", repository)
        sparql.setCredentials("admin", "i-0f47f689ca7184940")
        sparql.setQuery(query)
        sparql.setReturnFormat(CSV)
        results = sparql.query().convert()
        return pd.read_csv(BytesIO(results))
```



Query 1: Number of clicks made by user_id, code_module and code_presentation.

```
views_oulad = execute_query("""
PREFIX Repository: <http://www.metaphacts.com/ontologies/repository#>
PREFIX oulad: <https://ontologies.khaos.uma.es/oulad/>
SELECT ?code_module ?code_presentation ?user_id (SUM(?click) AS ?sum_clicks) WHERE {
    ?vle oulad:vle_has_course ?course.
    ?course oulad:code_module ?code_module.
    ?course oulad:code_presentation ?code_presentation.

    ?student_vle oulad:student_vle_has_vle ?vle.
    ?student_vle oulad:sum_click ?click.
    ?student_vle oulad:student_vle_has_student_info ?student_info.
    ?student_info oulad:id_student ?user_id.

} GROUP BY ?code_module ?code_presentation ?user_id
""")
views_oulad = views_oulad.set_index(['user_id','code_module', 'code_presentation'])
views_oulad
```

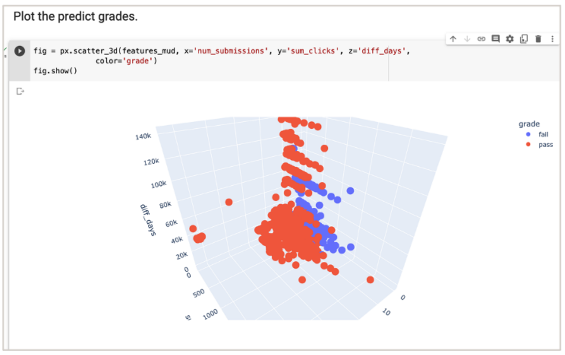
```
                                         sum_clicks
user_id  code_module  code_presentation
 281261      CCC            2014B               463
 564632      FFF            2014B              3483
  87322      FFF            2014J              1408
```



Predict grades with the model.

```
[23] y_pred=model.predict(features_mud)
     features_mud["grade"] = y_pred
     features_mud
```

```
                    num_submissions  sum_clicks  diff_days  grade
user_id  course_id
   8        16                  2.0        62.0    17540.0   fail
            20                  2.0        41.0       27.0   fail
            31                  0.0        97.0        0.0   fail
            58                 15.0        89.0    35877.0   pass
            66                 11.0       187.0    17583.0   pass
  ...       ...                 ...         ...        ...    ...
 15846      73                  4.0       114.0     -389.0   fail
```



Plot the predict grades.

```
fig = px.scatter_3d(features_mud, x='num_submissions', y='sum_clicks', z='diff_days',
              color='grade')
fig.show()
```

In the end we have been successful in running the entire playbook and thereby producing a training model as well as its application in predicting grades.

Notes on the results:

- Except for Query 3 and Query 7 all queries could be successfully executed in the FedX federation. For Query 3 and Query 7 explicit federation using the SPARQL 1.1 SERVICE clauses was required because of the Virtuoso proprietary bif:dateDiff function
- For the setup of metaphactory in the AWS cloud it was required to increase SPARQL query timeouts (from the default of 30s). Additionally, the timeouts of the Nginx

reverse proxy needed to be increased to avoid gateway timeouts. While there was only Query 1 having a duration of more than a minute, we have defined the timeout to 300s to be on the safe side.
- As metaphactory was running in the AWS cloud with self-signed certificates only (i.e. for this setup we have not used a proper domain), we had to tweak the python scripts to accept self-signed certificates.
- Additionally, we have adjusted the SPARQL connection in the Python script to use authentication.
- We had to workaround a number of Virtuoso specialties: as ASK queries are not supported, we use a Virtuoso Wrapper that transforms ASK queries to their SELECT LIMIT 1 variant. Also in our queries we had to deal with Virtuoso's interpretation of literals: "abc" is different from "abc"^^xsd:string

Results of the Life-Science Use-Case

For the Life-Science use-case three Bio2RDF datasets have been prepared:

- Drugbank
- HGNC
- GOA

Our goal was to run a federated query against these three individual Bio2RDF datasets through a FedX federation configured in metaphactory.

As a first step we have downloaded the three datasets from the sources given by the use-case providers.

While attempting to load the datasets into individual local triple-stores, we found that they were using invalid IRIs. Hence parsing, and thereby loading them as-is failed.

The main issue was that IRIs were using a non escaped ":" in their local parts:

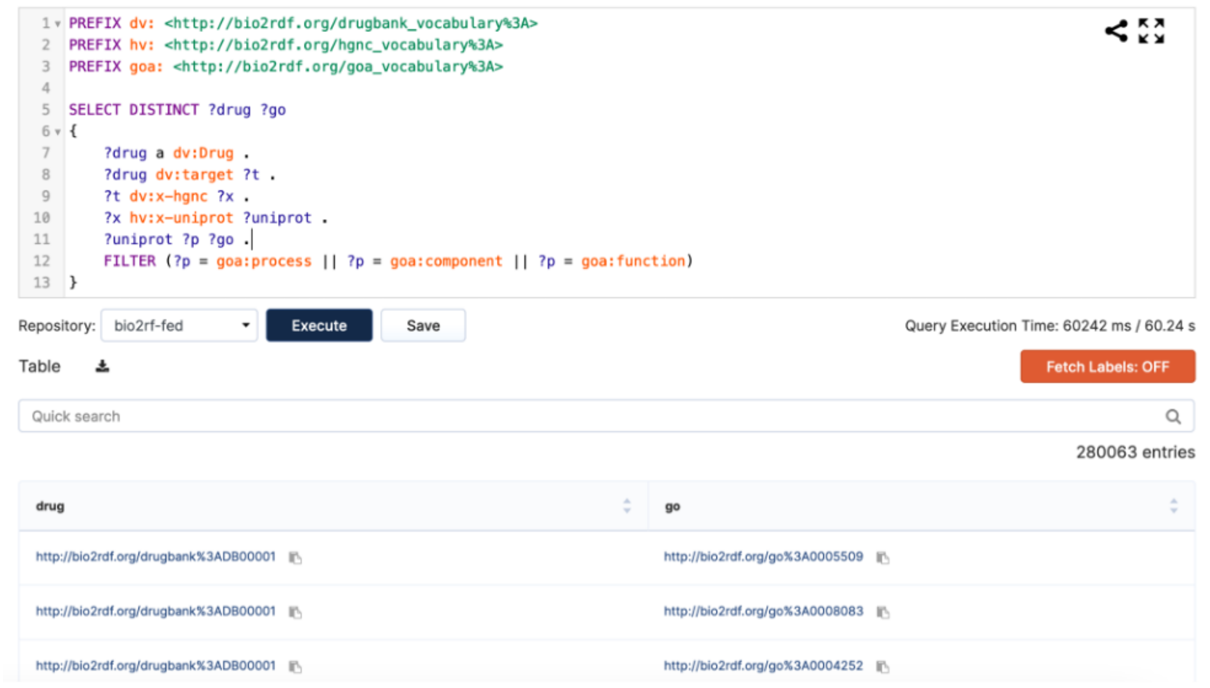
Example: <http://bio2rdf.org/wikipedia:Thallium(I)_chloride>

Our strategy thereafter was to resolve the issues in the data dumps using a small script: in each line we replaced the invalid occurrence of the colon in the local name using the "%3A" representation. We also addressed some other smaller issues in the RDF documents (i.e. non-escaped occurrences of % in the IRIs).

Once we have prepared and cleaned the datasets, we were able to successfully load them into individual local databases.

In metaphactory we then configured individual repositories connecting to the SPARQL endpoints of the data sources, and moreover a transparent FedX federation.

Finally, we successfully executed the query given by the use-case provider:



```
PREFIX dv: <http://bio2rdf.org/drugbank_vocabulary%3A>
PREFIX hv: <http://bio2rdf.org/hgnc_vocabulary%3A>
PREFIX goa: <http://bio2rdf.org/goa_vocabulary%3A>

SELECT DISTINCT ?drug ?go
{
    ?drug a dv:Drug .
    ?drug dv:target ?t .
    ?t dv:x-hgnc ?x .
    ?x hv:x-uniprot ?uniprot .
    ?uniprot ?p ?go .
    FILTER (?p = goa:process || ?p = goa:component || ?p = goa:function)
}
```

Repository: bio2rf-fed

Query Execution Time: 60242 ms / 60.24 s

Quick search

280063 entries

drug	go
http://bio2rdf.org/drugbank%3ADB00001	http://bio2rdf.org/go%3A0005509
http://bio2rdf.org/drugbank%3ADB00001	http://bio2rdf.org/go%3A0008083
http://bio2rdf.org/drugbank%3ADB00001	http://bio2rdf.org/go%3A0004252

Remarks:

- TheQuery was executed successfully despite the unbound triple pattern and using FILTER for the predicates
- Due to the local databases there was no latency in network communication. In a real distributed environment, the latency impacts the query execution time

Instructions for Setup

The use-cases have been worked on in a plain metaphactory installation. For the Hackathon we have prepared a metaphactory installation in the AWS cloud, where we collaboratively could work on the use-caes.

To repeat the setup, an installation of metaphactory needs to be set up. The steps to do so are described on our website: https://metaphacts.com/get-started Additionally, the website offers some demo resources and training material.

As part of the Hackathon we have prepared metaphactory Apps that pre-configure the metaphactory installation (e.g. with the relevant repositories and federation settings). The apps are available as results of the Hackathon in the shared drive.

Installation steps:

- Install the virtuoso-wrapper app (Admin -> Apps)
- Install the hackathon-results app (Admin -> Apps)
- Import the OULAD ontology (Assets -> Ontologies)
- Import the E-Lion ontology (Assets -> Ontologies)

Conclusion

In the Hackathon we have used metaphactory (and specifically the FedX federation engine) to work on the E-Learning and the Life-Science use-cases. We have shown how the repositories and a transparent FedX federation for the respective data sources can be configured in metaphactory. For the E-Learning use-case we have successfully formulated and executed the queries for training the E-Learning model in the transparent federation through the metaphactory SPARQL endpoint. Finally, we have adjusted the Python notebook to successfully use the formulated queries against metaphactory's SPARQL endpoint and thereby train the model. For the Life Science use-case we have shown that the FedX federation engine is capable of evaluating the use-case query successfully.

References

- metaphacts Homepage: https://metaphacts.com/
- Blog Article about Federation with metaphactory: https://blog.metaphacts.com/federation-in-metaphactory
- Access to metaphactory: https://metaphacts.com/get-started
- Metaphactory Federation Documentation: https://help.metaphacts.com/resource/Help:Federation
- Public Wikidata demo system: https://wikidata.metaphacts.com
- RDF4J documentation about FedX: https://rdf4j.org/documentation/programming/federation/

Federation Tool Provider 5 (Semagrow & KOBE)

Brief summary of the tool

Semagrow is a federated query processor that offers a single SPARQL endpoint for serving data from remote data sources and that hides from client applications heterogeneity in both form (federating non-SPARQL endpoints) and meaning (transparently mapping queries and query results between vocabularies). Semagrow uses metadata in order to perform effective source selection and cost-based query optimization, encoded using the Sevod vocabulary (an extension of VoID), and can automatically be extracted from the data using a tool called Sevod-scraper.

KOBE is a benchmarking system that leverages Docker and Kubernetes in order to reproduce experiments of federated query processing in collections of data sources. The KOBE benchmarking engine is a system that aims to provide a generic platform to perform benchmarking and experimentation that can be reproducible in different environments. It allows for benchmark and experiment specifications to be reproduced in different environments and be able to produce comparable and reliable results, and eases the deployment of complex benchmarking experiments by automating the tedious tasks of initialization and execution.

Aim

We worked in the Federated Shop use-case and in the Life-Science use-case. Regarding the Federated Shop, we were interested mostly due to the complex nature of the setup and the queries of the use-case. Regarding the Life-Science use-case, we were mainly interested to see practical queries from the domain since we were involved in the biomedical domain during the past.

Lesson Learnt and Results

Federated Shop Use-Case

The federated shop use-case setup uses virtual endpoints, meaning that all endpoints are loaded in a single Virtuoso endpoint but in different graphs, but since querying from a specific graph from an endpoint isn't supported yet, we focused more on the source selection and planning results for the queries. We used the newly developed functionality of Sevod-scraper for extracting metadata directly by the SPARQL endpoint and we created 2 types of summaries to experiment with: (1) only Void metadata, and (2) Sevod metadata using void predicate descriptions and URI prefixes). The second type of metadata improves the query execution plan, but it is not still effective enough, so this means that our source selector and planner need improving (and this use-case can drive more improvements in this direction). In addition, since FedShop is essentially a benchmark, we have started the integration with KOBE by providing benchmark descriptors of the FedShop benchmark using the formalism of KOBE. Finally, the FedShop team tested Sevod-Scraper, and provided us with valuable feedback on our tool.

Life Science Use-Case

The Life Science team provided us with 3 dump files, which we used to create a setup with 3 endpoints. Since we already had the dumps, we decided to extract Semagrow metadata directly through the dump files. We deployed the 3 federated Virtuoso endpoints in the Kubernetes cluster of NCSR-Demokritos using KOBE, and we deployed the Semagrow federator locally in the laptop here. We tried to execute Query3, but the planner outputted a strange plan, and then we found that it was probably caused by the complex filter of the query (the relative part of the query is the following: ?uniprot ?p ?go . FILTER (?p = goa:process || ?p = goa:component || ?p = goa:function)). By dropping the filter out of the query, we get the following results:

Query Run	Source Selection time (ms)	Query Planning time (ms)	Query Execution Time (ms)	Total query processing time (ms)	Number of sources in the plan	Number of query results
1st run	2441	99	2684	5224	3	16798
2nd run	41	21	2291	2353	3	16798
3rd run	23	16	2302	2341	3	16798

The slow source selection time difference between the first (cold) run and the others can be explained due to the caching of the ASK queries issued by the source selector to the endpoints. Finally, we documented the overall process with the valuable feedback of an end-user from the Life Science team, and we realised that there is missing documentation, especially for setting up the prerequisites of KOBE (since it involves Kuberentes environment)

General remarks

Our participation in the hackathon enabled us to gain a better understanding of the needs and perspectives of use-case providers, end-users, and non-technical users in federated querying scenarios. Overall, it was a positive and interesting experience that provided an opportunity for technical users and use-case providers to engage in meaningful discussions about their approaches.

We were accustomed to having access to dump RDF files for generating summaries and metadata, which is required from our engine in order to provide an effective query execution plan. After discussions during the hackathon, we understood that this fact is actually a non-realistic expectation from a practical point of view. It is important though to have tools that either generate metadata directly from the endpoints or to move to other, adaptive approaches that do not require extracting metadata.

One of the priorities of our team is to have good documentation for our tools so as to be easily deployed and executed by other people. It seems though that we have also to work towards this direction. For instance, regarding the documentation of KOBE, it seems that it is not clear in the readme how to install the prerequisites (Kubernetes cluster, etc).

Any relevant instructions

Instructions to run Semagrow for the Life-Science use case can be found here**.**
The process for Federated Shop is essentially the same.

Conclusion and Next steps

As a future work, we aim to focus on the following aspects:

- Improve Sevod-scraper tool in (extracting metadata from endpoint, creating more refined prefix metadata)
- Make Semagrow able to query specific graph of a SPARQL endpoint
- Further analyse the queries to fix bugs that discovered during the hackathon
- Further integration of FedShop benchmark within KOBE
- Improve the documentation of KOBE to include directions for prerequisites (Kubernetes cluster, etc).

Federation Tool Provider 6:

CostFed: Cost-Based Query Optimization for SPARQL Endpoint Federation

Brief summary of the tool:
CostFed is an index-assisted federation engine for federated SPARQL query processing over multiple endpoints. CostFed makes use of statistical information collected from endpoints to perform efficient source selection and cost-based query planning. In contrast to the state of the art, it relies on a non-linear model for the estimation of the selectivity of joins. Therewith, it is able to generate better plans than the state-of-the-art federation engines.

Aim [Which Use Cases addressed]:
We have addressed two use-cases namely use-case 1 E-learning and use-case 2 Federated Shop. Use-case 1 provided 5 public SPARQL endpoints. A total of 5 federated queries need to be executed. The first query was already provided by the use-case provider, while the other 4 queries need to be constructed by using the schema of the source datasets. For the Federated shop use case, a single SPARQL endpoint was provided with two different settings: 1) there were 20 named graphs where a single named graph contains data for one of the federated shops. That means a total of 20 federated shops need to be queried. 2) There were 100 named graphs where a single named graph contains data for one of the federated shops. That means a total of 100 federated shops need to be queried. For this use case, we need to run 12 federated queries. The queries were already provided by the use-case provider.

Lesson Learnt and Results:
In this section, first we report the evaluation result for each of the use-case followed by the lesson learnt.
Use-case1 results: Table given below shows the evaluation results for the 4 queries. We Query 3 resulted in a parse error because it was using a non-standard SPARQL function called DateDiff. For all other queries, only two distinct sources were queries and we successfully got complete results for other queries relatively fast.

QUERY	**No. of Distinct Sources**	**Runtime (s)**	**Resultsize**
1	2	194	29228
2	2	7.6	25843
3	NA	Error: QName 'bif:datediff' uses an undefined prefix. No standard SPARQL function	NA
4	2	7.7	25816

Use-case2 results:
We were unable to get results for any of the 12 queries. The reason was the use-case makes use of the virtual SPARQL endpoints instead of the real physical SPARQL endpoints. The use-case actually has only a single SPARQL endpoint with 20 or 100 named graphs.

The use-case provider suggested using the default named graphs as part of the SPARQL endpoint URL. e.g. http://localhost:8891/sparql?default-graph-uri=http://www.ratingsite0.fr/
 The actual endpoint url is http://localhost:8891/sparql with a named graph appended. We have tried using that. Unfortunately, everytime we get the error (exact error message is given on the next page) saying single source query with null relevant sources. It seems that the system does not differentiate between the different virtual SPARQL endpoints. Rather, it identifies only a single available source with an endpoint url http://localhost:8891/sparql. We are hopeful that the CostFed will successfully run queries if the use-case provider creates 20 or 100 physical SPARQL endpoints, each having a distinct endpoint URL.

Lessons Learned:
Overall, it was a wonderful experience to test CostFed on different use-cases. The user manual needs to be updated to clearly indicate the set of steps required to set up the federation engine. The index generation can be tricky on a normal user machine. It is recommended to create an index with a machine having at least 16 GB of RAM. The current Virtuoso does not accept SPARQL ASK queries while querying from OpenRDF or Jena library. To this end, a small change (aforementioned) is required in the CostFed code, before running experiments.

Any relevant instructions:
The source code can be checked out from https://github.com/dice-group/CostFed. Follow the following steps to run the Costfed

- Checkout the source code and import as a new maven project. it will create three sub-projects, i.e, costfed-core, fedx, and semagrow-bench.
- Create Index: Since CostFed is an index-assisted approach, the first step is to generate an index for all the endpoints in hand. The index generation, updation is given costfed/src/main/java/org/aksw/simba/quetsal/util/TBSSSummariesGenerator.java.

  Note for FedBench, LargeRDFBench, the index is already given at costfed/summaries/sum-localhost.n3.

- Configuration File: Set properties in /costfed/costfed.props or run with default
- Query Execution: costfed/src/main/java/org/aksw/simba/start/QueryEvaluation.java. Here you need to specify the URLs of the SPARQL endpoints which you want the given query to be federated with and provide the configuration file, i.e., costfed.props as argument.

We have just also discovered that Virtuoso does not accept SPARQL ASK queries using the client libraries like Jena or OpenRDF4J. Since the original source selection of the CostFed makes use of the SPARQL ASK queries, we were unable to run a single query in the start. However, I need to make a small change in the code in order to make it workable. So, in the package **package** com.fluidops.fedx.evaluation;

We need to go to go **public class** SparqlTripleSource class and change the boolean to false
private boolean useASKQueries = **false**;

For the second use-case, which makes use of the virtual SPARQL endpoints, CostFed is unable to differentiate between the real and virtual endpoints. As such, we were unable to run a single query. CostFed needs 20 different SPARQL endpoints URLs from 20 different virtuoso endpoints and not from a single virtuoso with 20 named Graphs.

Conclusion and Next steps:
Overall, it was really great to meet and discuss in person with experts of the SPARQL federated query processing. The use-cases were interesting and challenging. Certain bugs were revealed during running the use-case queries.

In future, we will work on updating the user-manual for CostFed. Also, we will make the CostFed web version available that will help the non-expert users to easily run federated SPARQL queries and register new SPARQL endpoints. We will try to run the use-case 2 queries at home and compare the results.

Exact Error for Use-Case 2:

```
java.lang.NullPointerException: Cannot invoke "com.fluidops.fedx.structures.Endpoint.getTripleSource()"
because "source" is null
        at
com.fluidops.fedx.evaluation.FederationEvalStrategy.evaluateSingleSourceQuery(FederationEvalStrategy.java:2
97)
        at
com.fluidops.fedx.evaluation.FederationEvalStrategy$EvalVisitor.meet(FederationEvalStrategy.java:185)
        at com.fluidops.fedx.algebra.SingleSourceQuery.visit(SingleSourceQuery.java:110)
```

User: Non-Use Case Provider and Tool Provider

Brief summary of the use case:

We used E-Learning data set as Use-Case. E-Learning platform has academic institutions about lessons and planning, course, student information, etc.

Observations:

We test the use case with four different platforms. As it following:

Colab: First, we implemented Sparql queries on colab. We added SPARQL endpoints as services in Query. However, although we knew the labels in the data model, we had difficulty combining different endpoints.

Metaphactory: Secondly, we tested the SPARQL queries with Metaphactory. However, we first learned how the system works and the tool's features. Metaphactory is a licensed software with an easy-to-use interface, that can present ontologies visually, and allows writing federated queries. However, it is not open-source licensed. Instead, there is a free trial version and they provide different usage opportunities to support academic studies. Metaphactory was able to run fast-typed SPARQL queries. However, since Query3 and Query7 are in the use-case use date functions, we had to add a prefix.

Comunica: Third, we tested the SPARQL queries with Comunica. However, the Comunica tool worked slowly because of the large data size. It took about 1 hour to fetch the results for Query 1.

HeFQUIN: Lastly, we tested the SPARQL queries with HeFQUIN. However, the HeFQUIN tool worked slowly because of the large data size. And also, HeFQUIN should add to limit the data because of using JENA in the background.

Lesson Learnt and Results:

First of all, it is necessary to know the features such as class, type, and cardinalities, which are in the content of the ontology. In short, it is necessary to examine the data model.

Secondly, in SPARQL queries, it is necessary to know how to write a federated query to make an inquiry from different endpoints;

Finally, for Federated queries; The URLs of the defined SPARQL endpoints should be well-defined. HTTP and HTTPS different extensions can cause problems in creating and running a federated query.

**Conclusion
and Next steps:**

We prepared a table about tested queries with Colab and the tools. It is given below:

PlatForm	
Colab	· The SPARQL endpoints need to be added to the query as a service. · The data model should be well-known.

	· If the data is too large and the query result will have more rows; It is necessary to filter the queries.
Metaphactory	· The ontology is loaded into the system. It has a UserInterface where classes, properties, cardinality, and relations in the ontology can be displayed. · It uses cloud infrastructure. · It is necessary to login through its own system and make the necessary sub-arrangements. · It is possible to write the automatic query. However, we did not test it. · It has documentation and training videos available.
Comunica	· There is no user interface. · If the structure of the ontology is known, the SPARQL endpoints do not need to be known. Sparql scans the EndPoints and chooses the appropriate one. However, if the endpoint is known, it can be added. · OpenSource and we can add it as a library. It has sample documentation for it. · Large data cause slow operation. · The results obtained from the queries can be written to the screen one by one.
HeFQUIN	· There is no user interface. · OpenSource but no sample documentation. · It is necessary to define SPARQL endpoints of previously federated parts. So pre-define files are needed. · Large data cause slow operation. In addition, there may be memory shortages due to the use of the JENA infrastructure. · The results obtained from the queries can be written to the screen one by one.

Challenges and future directions:

Recognition of data models, that is, the domain needs to be known to write SPARQL queries. Also, defining mappings between endpoints will be beneficial for end users.

It may take a long time to complete preliminary preparations such as adding indexes. Also, It might be useful for end users and developers if the tool providers prepare the documentation on how we can use the tool.

Large data can be difficult.

User: Non-Use Case Provider and Tool Provider-2 (The user attended only the first day.)

Brief summary of the use case: Life Science

What did we do?

- We set up the ColChain Tool
- Downloaded the required data files (*.nq.gz)
- We transformed the data files to the required format which is .hdt
- We tried both simple and complex queries to test the tool

What were the problems?

- Compiling the source code was not possible for me because of some Java problems. I tried to solve it but I couldn't.
- I am provided with the compiled version of the tool.
- Startup command is understandable for me but novice users may have problems with it.
- Transforming the data files was problematic.
- Queries that require the transfer of massive amounts of data between nodes were problematic.
- Frontend has some bugs which were identified by the team.
- Also, Frontend was not informative enough for the user. It should be more informative. For example, while uploading the data files, it should show the progress of the upload.

What can be done?

- Using container technology like Docker can be a solution for the problems with compiling the source code and transforming the data files.
- Frontend should be more informative and user-friendly. This could be done by adding progress bars where needed, adding more information about the process, adding more information about terms(observe, participate) and etc.

What do I want to learn more about tools?
What is the performance of the tool on low-end machines like Raspberry Pi?