Comunica: A Modular SPARQL Query Engine for the Web

Ruben Taelman(✉), Joachim Van Herwegen, Miel Vander Sande, and Ruben Verborgh

Ghent University – imec – IDLab, Ghent, Belgium
{ruben.taelman,joachim.herwegen,miel.vandersande,ruben.verborgh}@ugent.be

Abstract. Query evaluation over Linked Data sources has become a complex story, given the multitude of algorithms and techniques for single- and multi-source querying, as well as the heterogeneity of Web interfaces through which data is published online. Today's query processors are insufficiently adaptable to test multiple query engine aspects in combination, such as evaluating the performance of a certain join algorithm over a federation of heterogeneous interfaces. The Semantic Web research community is in need of a flexible query engine that allows plugging in new components such as different algorithms, new or experimental SPARQL features, and support for new Web interfaces. We designed and developed a Web-friendly and modular meta query engine called Comunica that meets these specifications. In this article, we introduce this query engine and explain the architectural choices behind its design. We show how its modular nature makes it an ideal research platform for investigating new kinds of Linked Data interfaces and querying algorithms. Comunica facilitates the development, testing, and evaluation of new query processing capabilities, both in isolation and in combination with others.

1 Introduction

Linked Data on the Web exists in many shapes and forms—and so do the processors we use to query data from one or multiple sources. For instance, engines that query RDF data using the SPARQL language [1] employ different algorithms [2,3] and support different language extensions [4,5]. Furthermore, Linked Data is increasingly published through different Web interfaces, such as data dumps, Linked Data documents [6], SPARQL endpoints [7] and Triple Pattern Fragments (TPF) interfaces [8]. This has led to entirely different query evaluation strategies, such as server-side [7], link-traversal-based [9], shared client-server query processing [8], and client-side (by downloading data dumps and loading them locally).

The resulting variety of implementations suffers from two main problems: a lack of sustainability and a lack of comparability. Alternative query algorithms and features are typically either implemented as forks of existing software packages [10–12] or as independent engines [13]. This

D. Vrandečić et al. (Eds.): ISWC 2018, LNCS 11137, pp. 239–255, 2018.
https://doi.org/10.1007/978-3-030-00668-6_15

practice has limited sustainability: forks are often not merged into the main software distribution and hence become abandoned; independent implementations require a considerable upfront cost and also risk abandonment more than established engines. Comparability is also limited: forks based on older versions of an engine cannot meaningfully be evaluated against newer forks, and evaluating combinations of cross-implementation features—such as different algorithms on different interfaces—is not possible without code adaptation. As a result, many interesting comparisons are never performed because they are too costly to implement and maintain. For example, it is currently unknown how the Linked Data Eddies algorithm [13] performs over a federation [8] of brTPF interfaces [14]. Another example is that the effects of various optimizations and extensions for TPF interfaces [10–17] have only been evaluated in isolation, whereas certain combinations will likely prove complementary.

In order to handle the increasing heterogeneity of Linked Data on the Web, as well as various solutions for querying it, there is a need for a flexible and modular query engine to experiment with all of these techniques—both separately and in combination. In this article, we introduce Comunica to realize this vision. It is a highly modular meta engine for federated SPARQL query evaluation over heterogeneous interfaces, including TPF interfaces, SPARQL endpoints, and data dumps. Comunica aims to serve as a flexible research platform for designing, implementing, and evaluating new and existing Linked Data querying and publication techniques.

Comunica differs from existing query processors on different levels:

1. The **modularity** of the Comunica meta query engine allows for extensions and customization of algorithms and functionality. Users can build and fine-tune a concrete engine by wiring the required modules through an RDF configuration document. By publishing this document, experiments can repeated and adapted by others.
2. Within Comunica, multiple **heterogeneous interfaces** are first-class citizens. This enables federated querying over heterogeneous sources and makes it for example possible to evaluate queries over any combination of SPARQL endpoints, TPF interfaces, datadumps, or other types of interfaces.
3. Comunica is implemented using **Web-based technologies** in JavaScript, which enables usage through browsers, the command line, the SPARQL protocol [7], or any Web or JavaScript application.

Comunica and its default modules are publicly available on GitHub and the npm package manager under the open-source MIT license (canonical citation: https://zenodo.org/record/1202509#.Wq9GZhNuaHo).

This article is structured as follows. In the next section, we discuss the related work, followed by the main features of Comunica in Sect. 3. After that, we introduce the architecture of Comunica in Sect. 4, and its implementation in Sect. 5. Next, we compare the performance of different Comunica configurations with the TPF Client in Sect. 6. Finally, Sect. 7 concludes and discusses future work.

2 Related Work

In this section, we illustrate the many possible degrees of freedom for SPARQL query evaluation, and show that they are hard to combine, which is the problem we aim to solve with Comunica. We first discuss the SPARQL query language, its engines, and algorithms. After that, we discuss alternative Linked Data publishing interfaces, and their connection to querying. Finally, we discuss the software design patterns that are essential in the architecture of Comunica.

2.1 The Different Facets of SPARQL

SPARQL [1] is the W3C-recommended RDF query language. The traditional way to implement a SPARQL query processor is to use it as an interface to an underlying database, resulting in a so-called SPARQL endpoint [7]. This is similar to how an SQL interface provides access to a relation database. The internal storage can either be a native RDF store, e.g., AllegroGraph [18] and Blazegraph [19], or a non-RDF store, e.g., Virtuoso [20] uses a object-relational database management system.

Various algorithms have been proposed for optimized SPARQL query evaluation. Some algorithms for example use the concept of query rewriting [2] based on algebraic equivalent query operations, others have proposed the optimization of Basic Graph Pattern evaluation [3] using selectivity estimation of triple patterns.

In order to evaluate SPARQL queries over datasets of different storage types, SPARQL query frameworks were developed, such as Jena (ARQ) [21], RDFLib [22], rdflib.js [23] and rdfstore-js [24]. Jena is a Java framework, RDFLib is a python package, and rdflib.js and rdfstore-js are JavaScript modules. Jena—or more specifically the ARQ API—and RDFLib are fully SPARQL 1.1 [1] compliant. rdflib.js and rdfstore-js both support a subset of SPARQL 1.1. These SPARQL engines support in-memory models or other sources, such as Jena TDB in the case of ARQ. Most of the query algorithms are tightly coupled to these frameworks, which makes swapping out query algorithms for specific query operators hard or sometimes even impossible. Furthermore, complex things such as federated querying over heterogeneous interfaces are difficult to implement using these frameworks, as they are not supported out-of-the-box. This issue of modularity and heterogeneity are two of the main problems we aim to solve within Comunica. The differences between Comunica and existing frameworks will be explained in more detail in Sect. 3.

The Triple Pattern Fragments client [8] (also known as Client.js or `ldf-client`) is a client-side SPARQL engine that retrieves data over HTTP through Triple Pattern Fragments (TPF) interfaces [8]. Different algorithms [10,16,17] for this client and TPF interface extensions [11,12,14,15] have been proposed to reduce effort of server or client in some way. All of these efforts are however implemented and evaluated in isolation. Furthermore, the implementations are tied to TPF interface, which makes it impossible to use them for other types of datasources and interfaces. With Comunica, we aim to solve this by modularizing query

operation implementations into separate modules, so that they can be plugged in and combined in different ways, on top of different datasources and interfaces.

With Semantic Web technologies providing the capability to integrate data from different sources, federated query processing has been an active area of research. However, most of the existing frameworks require SPARQL endpoints on every source. The TPF Client instead federates over TPF interfaces, and achieves similar performance compared to the state of the art [8] despite its usage of a more lightweight interface. However, no frameworks exist that enable federation over heterogeneous interfaces, such as the federation over any combination of SPARQL endpoints and TPF interfaces. With Comunica, we aim to fill this gap. In addition dataset-centric approaches, alternative methods such as link-traversal-based query evaluation [9] exist to query a web of Linked Data documents.

2.2 Linked Data Fragments

In order to formally capture the heterogeneity of different Web interfaces to publish RDF data, the Linked Data Fragment [8] (LDF) conceptual framework uniformly characterizes responses of Web interfaces to RDF-based knowledge graphs. The simplest type of LDF is a data dump—it is the response of a single HTTP requests for a complete RDF dataset. Other types of LDFs includes responses of SPARQL endpoints, TPF interfaces, and Linked Data documents.

Existing LDF research highlights that, when it comes to publishing datasets on the Web, there is no silver bullet: no single interface works well in all situations, as each one involves trade-offs [8]. As such, data publishers must choose the type of interface that matches their intended use case, target audience and infrastructure. This however complicates client-side engines that need to retrieve data from the resulting heterogeneity of interfaces. As shown by the TPF approach, interfaces can be self-descriptive and expose one or more features [25], to describe their functionality using a common vocabulary [26,27]. This allows clients without prior knowledge of the exact inputs and outputs of an interface to discover its usage at runtime.

A design goal of Comunica is to facilitate interaction with any current and future interface within the LDF framework, both in single-source and federated scenarios.

2.3 Software Design Patterns

In the following, we discuss three software design patterns that are relevant to the modular design of the Comunica engine.

Publish-Subscribe Pattern. The publish-subscribe [28] design pattern involves passing messages between publishers and subscribers. Instead of programming publishers to send messages directly to subscribers, they are programmed to publish messages to certain categories. Subscribers can subscribe to

these categories which will cause them to receive these published messages, without requiring prior knowledge of the publishers. This pattern is useful for decoupling software components from each other, and only requiring prior knowledge of message categories. We use this pattern in Comunica for allowing different implementations of certain tasks to subscribe to task-specific buses.

Actor Model. The actor model [29] was designed as a way to achieve highly parallel systems consisting of many independent agents communicating using messages, similar to the publish–subscribe pattern. An actor is a computational unit that performs a specific task, acts on messages, and can send messages to other actors. The main advantages of the actor model are that actors can be independently made to implement certain specific tasks based on messages, and that these can be handled asynchronously. These characteristics are highly beneficial to the modularity that we want to achieve with Comunica. That is why we use this pattern in combination with the publish-subscribe pattern to let each implementation of a certain task correspond to a separate actor.

Mediator Pattern. The mediator [30] pattern is able to reduce coupling between software components that interact with each other, and to easily change the interaction if needed. This can be achieved by encapsulating the interaction between software components in a mediator component. Instead of the components having to interact with each other directly, they now interact through the mediator. These components therefore do not require prior knowledge of each other, and different implementations of these mediators can lead to different interaction results. In Comunica, we use this pattern to handle actions when multiple actors are able to solve the same task, by for example choosing the best actor for a task, or by combining the solutions of all actors.

3 Requirement Analysis

In this section, we discuss the main requirements and features of the Comunica framework as a research platform for SPARQL query evaluation. Furthermore, we discuss each feature based on the availability in related work. The main feature requirements of Comunica are the following:

SPARQL query evaluation The engine should be able to interpret, process and output results for SPARQL queries.

Modularity Different independent modules should contain the implementation of specific tasks, and they should be combinable in a flexible framework. The configurations should be describable in RDF.

Heterogeneous interfaces Different types of datasource interfaces should be supported, and it should be possible to add new types independently.

Federation The engine should support federated querying over different interfaces.

Web-based The engine should run in Web browsers using native Web technologies.

In Table 1, we summarize the availability of these features in similar works.

Table 1. Comparison of the availability of the main features of Comunica in similar works. (1) A subset of SPARQL 1.1 is implemented. (2) Querying over SPARQL endpoints, other types require implementing an internal storage interface. (3) Downloading of dumps. (4) Federation only over SPARQL endpoints using the SERVICE keyword.

Feature	TPF Client	ARQ	RDFLib	rdflib.js	rdfstore-js	Comunica
SPARQL	✓(1)	✓	✓	✓(1)	✓(1)	✓(1)
Modularity						✓
Heterogeneous interfaces		✓(2,3)	✓(2,3)	✓(3)	✓(3)	✓
Federation	✓	✓(4)	✓(4)			✓
Web-based	✓			✓	✓	✓

3.1 SPARQL Query Evaluation

The recommended way of querying within RDF data, is using the SPARQL query language. All of the discussed frameworks support at least the parsing and execution of SPARQL queries, and reporting of results.

3.2 Modularity

Adding new functionality or changing certain operations in Comunica should require minimal to no changes to existing code. Furthermore, the Comunica environment should be developer-friendly, including well documented APIs and auto-generation of stub code. In order to take full advantage of the Linked Data stack, modules in Comunica must be describable, configurable and wireable in RDF. By registering or excluding modules from a configuration file, the user is free to choose how heavy or lightweight the query engine will be. Comunica's modular architecture will be explained in Sect. 4. ARQ, RDFLib, rdflib.js and rdfstore-js only support customization by implementing a custom query engine programmatically to handle operators. They do not allow plugging in or out certain modules.

3.3 Heterogeneous Interfaces

Due to the existence of different types of Linked Data Fragments for exposing Linked Datasets, Comunica should support heterogeneous interfaces types, including self-descriptive Linked Data interfaces such as TPF. This TPF interface is the only interface that is supported by the TPF Client. Additionally, Comunica should also enable querying over other sources, such as SPARQL endpoints and data dumps in RDF serializations. The existing SPARQL frameworks mostly support querying against SPARQL endpoints, local graphs, and specific storage types using an internal storage adapter.

3.4 Federation

Next to the different type of Linked Data Fragments for exposing Linked Datasets, data on the Web is typically spread over different datasets, at different locations. As mentioned in Sect. 2, federated query processing is a way to query over the combination of such datasets, without having to download the complete datasets and querying over them locally. The TPF client supports federated query evaluation over its single supported interface type, i.e., TPF interfaces. ARQ and RDFLib only support federation over SPARQL endpoints using the SERVICE keyword. Comunica should enable combined federated querying over its supported heterogeneous interfaces.

3.5 Web-Based

Comunica must be built using native Web technologies, such as JavaScript and RDF configuration documents. This allows Comunica to run in different kinds of environments, including Web browsers, local (JavaScript) runtime engines and command-line interfaces, just like the TPF-client, rdflib.js and rdfstore-js. ARQ and RDFLib are able to run in their language's runtime and via a command-line interface, but not from within Web browsers. ARQ would be able to run in browsers using a custom Java applet, which is not a native Web technology.

4 Architecture

In this section, we discuss the design and architecture of the Comunica meta engine, and show how it conforms to the modularity feature requirement. In summary, Comunica is collection of small modules that, when wired together, are able to perform a certain task, such as evaluating SPARQL queries. We first discuss the customizability of Comunica at design-time, followed by the flexibility of Comunica at run-time. Finally, we give an overview of all modules.

4.1 Customizable Wiring at Design-Time Through Dependency Injection

There is no such thing as the Comunica engine, instead, Comunica is a meta engine that can be instantiated into different engines based on different configurations. Comunica achieves this customizability at design-time using the concept of dependency injection [31]. Using a configuration file, which is created before an engine is started, components for an engine can be selected, configured and combined. For this, we use the Components.js [32] JavaScript dependency injection framework, This framework is based on semantic module descriptions and configuration files using the Object-Oriented Components ontology [33].

Description of Individual Software Components. In order to refer to Comunica components from within configuration files, we semantically describe

all Comunica components using the Components.js framework in JSON-LD [34]. Listing 1 shows an example of the semantic description of an RDF parser.

Description of Complex Software Configurations. A specific instance of a Comunica engine can be initialized using Components.js configuration files that describe the wiring between components. For example, Listing 2 shows a configuration file of an engine that is able to parse N3 and JSON-LD-based documents. This example shows that, due to its high degree of modularity, Comunica can be used for other purposes than a query engine, such as building a custom RDF parser.

Since many different configurations can be created, it is important to know which one was used for a specific use case or evaluation. For that purpose, the RDF documents that are used to instantiate a Comunica engine can be published as Linked Data [33]. They can then serve as provenance and as the basis for derived set-ups or evaluations.

```
{
  "@context": [ ... ],
  "@id": "npmd:@comunica/actor-rdf-parse-n3",
  "components": [
    {
      "@id":            "crpn3:Actor/RdfParse/N3",
      "@type":          "Class",
      "extends":        "cbrp:Actor/RdfParse",
      "requireElement": "ActorRdfParseN3",
      "comment":        "An actor that parses Turtle-like RDF",
      "parameters": [
        {
          "@id": "caam:Actor/AbstractMediaTypedFixed/mediaType",
          "default": [ "text/turtle", "application/n-triples" ]
        }
      ]
    }
  ]
}
```

Listing 1: Semantic description of a component that is able to parse N3-based RDF serializations. This component has a single parameter that allows media types to be registered that this parser is able to handle. In this case, the component has four default media types.

```
{
  "@context": [ ... ],
  "@id": "http://example.org/myrdfparser",
  "@type": "Runner",
  "actors": [
```

```
    { "@type": "ActorInitRdfParse",
      "mediatorRdfParse": {
        "@type": "MediatorRace",
        "cc:Mediator/bus": { "@id": "cbrp:Bus/RdfParse" }
      } },
    { "@type": "ActorRdfParseN3",
      "cc:Actor/bus": "cbrp:Actor/RdfParse" },
    { "@type": "ActorRdfParseJsonLd",
      "cc:Actor/bus": "cbrp:Actor/RdfParse" },
  ]
}
```

Listing 2: Comunica configuration of `ActorInitRdfParse` for parsing an RDF document in an unknown serialization. This actor is linked to a mediator with a bus containing two RDF parsers for specific serializations.

4.2 Flexibility at Run-Time Using the Actor-Mediator-Bus Pattern

Once a Comunica engine has been configured and initialized, components can interact with each other in a flexible way using the actor [29], mediator [30], and publish-subscribe [28] patterns. Any number of actor, mediator and bus modules can be created, where each actor interacts with mediators, that in turn invoke other actors that are registered to a certain bus.

Fig. 1 shows an example logic flow between actors through a mediator and a bus. The relation between these components, their phases and the chaining of them will be explained hereafter.

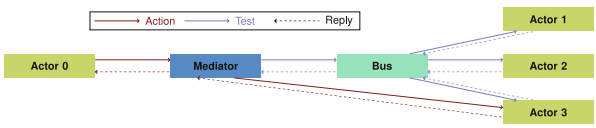

Fig. 1. Example logic flow where Actor 0 requires an action to be performed. This is done by sending the action to the Mediator, which sends a test action to Actors 1, 2 and 3 via the Bus. The Bus then sends all test replies to the Mediator, which chooses the best actor for the action, in this case Actor 3. Finally, the Mediator sends the original action to Actor 3, and returns its response to Actor 0.

Relation Between Actors and Buses. Actors are the main computational units in Comunica, and buses and mediators form the glue that ties them together and makes them interactable. Actors are responsible for being able to accept certain messages via the bus to which they are subscribed, and for responding with an answer. In order to avoid a single high-traffic bus for all message types which could cause performance issues, separate buses exist for different message types. Fig. 2 shows an example of how actors can be registered to buses.

Fig. 2. An example of two different buses each having two subscribed actors. The left bus has different actors for parsing triples in a certain RDF serialization to triple objects. The right bus has actors that join query bindings streams together in a certain way.

Mediators Handle Actor Run and Test Phases. Each mediator is connected to a single bus, and its goal is to determine and invoke the best actor for a certain task. The definition of 'best' depends on the mediator, and different implementations can lead to different choices in different scenarios. A mediator works in two phases: the test phase and the run phase. The test phase is used to check under which conditions the action can be performed in each actor on the bus. This phase must always come before the run phase, and is used to select which actor is best suited to perform a certain task under certain conditions. If such an actor is determined, the run phase of a single actor is initiated. This run phase takes this same type of message, and requires to effectively act on this message, and return the result of this action. Fig. 3 shows an example of a mediator invoking a run and test phase.

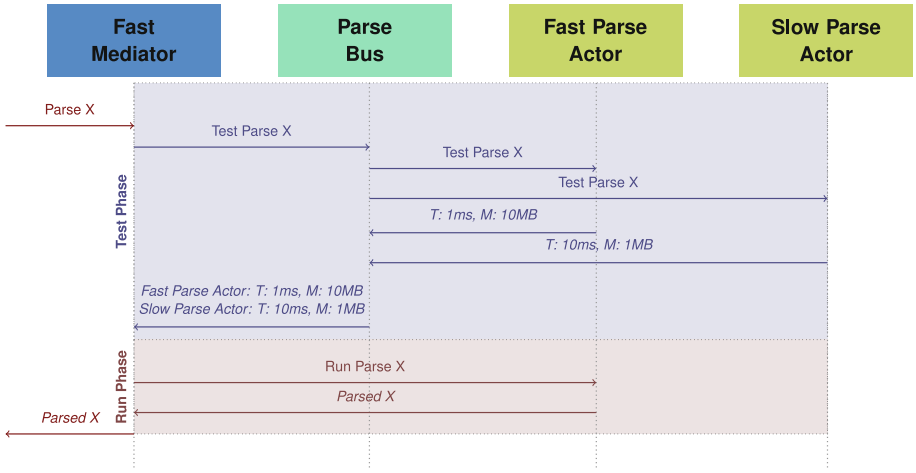

Fig. 3. Example sequence diagram of a mediator that chooses the fastest actor on a parse bus with two subscribed actors. The first parser is very fast but requires a lot of memory, while the second parser is slower, but requires less memory. Which one is best, depends on the use case and is determined by the Mediator. The mediator first calls the tests the actors for the action, and then runs the action using the best actor.

4.3 Modules

At the time of writing, Comunica consists of 79 different modules. This consists of 13 buses, 3 mediator types, 57 actors and 6 other modules. In this section, we will only discuss the most important actors and their interactions.

The main bus in Comunica is the query operation bus, which consists of 19 different actors that provide at least one possible implementation of the typical SPARQL operations such as quad patterns, basic graph patterns (BGPs), unions, projects, ... These actors interact with each other using streams of quad or solution mappings, and act on a query plan expressed in in SPARQL algebra [1].

In order to enable heterogeneous sources to be queried in a federated way, we allow a list of sources, annotated by type, to be passed when a query is initiated. These sources are passed down through the chain of query operation actors, until the quad pattern level is reached. At this level, different actors exist for handling a single source of a certain type, such as TPF interfaces, SPARQL endpoints, local or remote datadumps. In the case of multiple sources, one actor exists that implements a federation algorithm defined for TPF [8], but instead of federating over different TPF interfaces, it federates over different single-source quad pattern actors.

At the end of the pipeline, different actors are available for serializing the results of a query in different ways. For instance, there are actors for serializing the results according to the SPARQL JSON [35] and XML [36] result specifications, but actors with more visual and developer-friendly formats are available as well.

5 Implementation

Comunica is implemented in TypeScript/JavaScript as a collection of Node modules, which are able to run in Web browsers using native Web technologies. Comunica is available under an open license on GitHub and on the NPM package manager. The 79 Comunica modules are tested thoroughly, with more than 1,200 unit tests reaching a test coverage of 100%. In order to be compatible with existing JavaScript RDF libraries, Comunica follows the JavaScript API specification by the RDFJS community group, and will actively be further aligned within this community. In order to encourage collaboration within the community, we extensively use the GitHub issue tracker for planned features, bugs and other issues. Finally, we publish detailed documentation for the usage and development of Comunica.

We provide a default Linked Data-based configuration file with all available actors for evaluating federated SPARQL queries over heterogeneous sources. This allows SPARQL queries to be evaluated using a command-line tool, from a Web service implementing the SPARQL protocol [7], within a JavaScript application, or within the browser. We fully implemented SPARQL 1.0 [37] and a subset of SPARQL 1.1 [1] at the time of writing. In future work, we intend to implement additional actors for supporting SPARQL 1.1 completely.

Comunica currently supports querying over the following types of heterogeneous datasources and interfaces:

- Triple Pattern Fragments interfaces [8]
- Quad Pattern Fragments interfaces (an experimental extension of TPF with a fourth graph element)
- SPARQL endpoints [7]
- Local and remote dataset dumps in RDF serializations.
- HDT datasets [38]
- Versioned OSTRICH datasets [39]

In order to demonstrate Comunica's ability to evaluate federated query evaluation over heterogeneous sources, the following guide shows how you can try this out in Comunica yourself.

Support for new algorithms, query operators and interfaces can be implemented in an external module, without having to create a custom fork of the engine. The module can then be plugged into existing or new engines that are identified by RDF configuration files.

In the future, we will also look into adding support for other interfaces such as brTPF [14] for more efficient join operations and VTPF [15] for queries over versioned datasets.

6 Performance Analysis

One of the goals of Comunica is to replace the TPF Client as a more flexible and modular alternative, with at least the same functionality and similar performance. The fact that Comunica supports multiple heterogeneous interfaces and sources as shown in the previous section validates this flexibility and modularity, as the TPF Client only supports querying over TPF interfaces.

Next to a functional completeness, it is also desired that Comunica achieves similar performance compared to the TPF Client. The higher modularity of Comunica is however expected to cause performance overhead, due to the additional bus and mediator communication, which does not exist in the TPF Client. Hereafter, we compare the performance of the TPF Client and Comunica and discover that Comunica has similar performance to the TPF Client. As the main goal of Comunica is modularity, and not absolute performance, we do not compare with similar frameworks such as ARQ and RDFLib. Instead, relative performance of evaluations using the same engine under different configurations is key for comparisons, which will be demonstrated using Comunica hereafter.

For the setup of this evaluation we used a single machine (Intel Core i5-3230M CPU at 2.60 GHz with 8 GB of RAM), running the Linked Data Fragments server with a HDT-backend [38] and the TPF Client or Comunica, for which the exact versions and configurations will be linked in the following workflow. The main goal of this evaluation is to determine the performance impact of Comunica, while keeping all other variables constant.

In order to illustrate the benefit of modularity within Comunica, we evaluate using two different configurations of Comunica. The first configuration (Comunica-sort) implements a BGP algorithm that is similar to that of the original TPF Client: it sorts triple patterns based on their estimated counts and evaluates and joins them in that order. The second configuration (Comunica-smallest) implements a simplified version of this BGP algorithm that does not sort all triple patterns in a BGP, but merely picks the triple pattern with the smallest estimated count to evaluate on each recursive call, leading to slightly different query plans.

We used the following evaluation workflow:

1. Generate a WatDiv [40] dataset with scale factor=100.
2. Generate the corresponding default WatDiv queries with query-count=5.
3. Install the server software configuration, implementing the TPF specification, with its dependencies.
4. Install the TPF Client software, implementing the SPARQL 1.1 protocol, with its dependencies.
5. Execute the generated WatDiv queries 3 times on the TPF Client, after doing a warmup run, and record the execution times results.
6. Install the Comunica software configuration, implementing the SPARQL 1.1 protocol, with its dependencies, using the Comunica-sort algorithm.
7. Execute the generated WatDiv queries 3 times on the Comunica client, after doing a warmup run, and record the execution times.
8. Update the Comunica installation to use a new configuration supporting the Comunica-smallest algorithm.
9. Execute the generated WatDiv queries 3 times on the Comunica client, after doing a warmup run, and record the execution times.

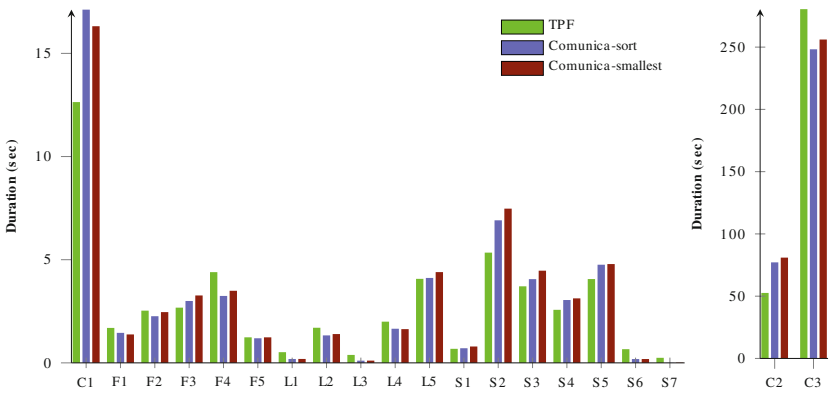

Fig. 4. Average query evaluation times for the TPF Client, Comunica-sort, and Comunica-smallest for all queries (shorter is better). C2 and C3 are shown separately because of their higher evaluation times.

The results from Fig. 4 show that Comunica is able to achieve similar performance compared to the TPF Client. Concretely, both Comunica variants are faster for 11 queries, and slower for 9 queries. However, the difference in evaluation times is in most cases very small, and are caused by implementation details, as the implemented algorithms are equivalent. Contrary to our expectations, the performance overhead of Comunica's modularity is negligible. Comunica therefore improves upon the TPF Client in terms of modularity and functionality, and achieves similar performance.

These results also illustrate the simplicity of comparing different algorithms inside Comunica. In this case, we compared an algorithm that is similar to that of the original TPF Client with a simplified variant. The results show that the performance is very similar, but the original algorithm (Comunica-sort) is faster in most of the cases. It is however not always faster, as illustrated by query C1, where Comunica-sort is almost a second slower than Comunica-smallest. In this case, the heuristic algorithm of the latter was able to come up with a slightly better query plan. Our goal with this result is to show that Comunica can easily be used to compare such different algorithms, where future work can focus on smart mediator algorithms to choose the best BGP actor in each case.

7 Conclusions

In this work, we introduced Comunica as a highly modular meta engine for federated SPARQL query evaluation over heterogeneous interfaces. Comunica is thereby the first system that accomplishes the Linked Data Fragments vision of a client that is able to query over heterogeneous interfaces. Not only can Comunica be used as a client-side SPARQL engine, it can also be customized to become a more lightweight engine and perform more specific tasks, such as for example only evaluating BGPs over Turtle files, evaluating the efficiency of different join operators, or even serve as a complete server-side SPARQL query endpoint that aggregates different datasources. In future work, we will look into supporting supporting alternative (non-semantic) query languages as well, such as GraphQL [41].

If you are a Web researcher, then Comunica is the ideal research platform for investigating new Linked Data publication interfaces, and for experimenting with different query algorithms. New modules can be implemented independently without having to fork existing codebases. The modules can be combined with each other using an RDF-based configuration file that can be instantiated into an actual engine through dependency injection. However, the target audience is broader than just the research community. As Comunica is built on Linked Data and Web technologies, and is extensively documented and has a ready-to-use API, developers of RDF-consuming (Web) applications can also make use of the platform. In the future, we will continue maintaining and developing Comunica and intend to support and collaborate with future researchers on this platform.

The introduction of Comunica will trigger a new generation of Web querying research. Due to its flexibility and modularity, existing areas can be combined

and evaluated in more detail, and new promising areas that remained covered so far will be exposed.

Acknowledgements. The described research activities were funded by Ghent University, imec, Flanders Innovation & Entrepreneurship (AIO), and the European Union. Ruben Verborgh is a postdoctoral fellow of the Research Foundation - Flanders.

References

1. Harris, S., Seaborne, A., Prud'hommeaux, E.: SPARQL 1.1 Query Language. W3C (2013). https://www.w3.org/TR/2013/REC-sparql11-query-20130321/
2. Schmidt, M., Meier, M., Lausen, G.: Foundations of SPARQL query optimization. In: Proceedings of the 13th International Conference on Database Theory, pp. 4–33 (2010)
3. Stocker, M., Seaborne, A., Bernstein, A., Kiefer, C., Reynolds, D.: SPARQL basic graph pattern optimization using selectivity estimation. In: Proceedings of the 17th International Conference on World Wide Web, pp. 595–604 (2008)
4. Erling, O., Mikhailov, I.: RDF support in the virtuoso DBMS. In: Pellegrini, T., Auer, S., Tochtermann, K., Schaffert, S. (eds.) Networked Knowledge - Networked Media: Integrating Knowledge Management, New Media Technologies and Semantic Systems, vol. 221, pp. 7–24. Springer, Heidelberg (2009). https://doi.org/10.1007/978-3-642-02184-8_2
5. Cheng, J., Ma, Z.M., Yan, L.: f-SPARQL: a flexible extension of SPARQL. In: Bringas, P.G., Hameurlain, A., Quirchmayr, G. (eds.) DEXA 2010. LNCS, vol. 6261, pp. 487–494. Springer, Heidelberg (2010). https://doi.org/10.1007/978-3-642-15364-8_41
6. Berners-Lee, T.: Linked Data (2009). https://www.w3.org/DesignIssues/LinkedData.html
7. Feigenbaum, L., Todd Williams, G., Grant Clark, K., Torres, E.: SPARQL 1.1 Protocol. W3C (2013). https://www.w3.org/TR/2013/REC-sparql11-protocol-20130321/
8. Verborgh, R., et al.: Triple pattern fragments: a low-cost knowledge graph interface for the web. J. Web Semantics **37**(38), 184–206 (2016)
9. Hartig, O.: An overview on execution strategies for Linked Data queries. Datenbank-Spektrum **13**, 89–99 (2013)
10. Van Herwegen, J., Verborgh, R., Mannens, E., Van de Walle, R.: Query execution optimization for clients of triple pattern fragments. In: Gandon, F., Sabou, M., Sack, H., d'Amato, C., Cudré-Mauroux, P., Zimmermann, A. (eds.) ESWC 2015. LNCS, vol. 9088, pp. 302–318. Springer, Cham (2015). https://doi.org/10.1007/978-3-319-18818-8_19
11. Vander Sande, M., Verborgh, R., Van Herwegen, J., Mannens, E., Van de Walle, R.: Opportunistic linked data querying through approximate membership metadata. In: Arenas, M., et al. (eds.) ISWC 2015. LNCS, vol. 9366, pp. 92–110. Springer, Cham (2015). https://doi.org/10.1007/978-3-319-25007-6_6
12. Van Herwegen, J., De Vocht, L., Verborgh, R., Mannens, E., Van de Walle, R.: Substring filtering for low-cost linked data interfaces. In: Arenas, M., et al. (eds.) ISWC 2015. LNCS, vol. 9366, pp. 128–143. Springer, Cham (2015). https://doi.org/10.1007/978-3-319-25007-6_8

13. Acosta, M., Vidal, M.-E.: Networks of linked data eddies: an adaptive web query processing engine for RDF data. In: Arenas, M., et al. (eds.) ISWC 2015. LNCS, vol. 9366, pp. 111–127. Springer, Cham (2015). https://doi.org/10.1007/978-3-319-25007-6_7
14. Hartig, O., Buil-Aranda, C.: Bindings-restricted triple pattern fragments. In: Debruyne, C. (ed.) OTM 2016. LNCS, vol. 10033, pp. 762–779. Springer, Cham (2016). https://doi.org/10.1007/978-3-319-48472-3_48
15. Taelman, R., Vander Sande, M., Verborgh, R., Mannens, E.: Versioned triple pattern fragments: a low-cost linked data interface feature for web archives. In: Proceedings of the 3rd Workshop on Managing the Evolution and Preservation of the Data Web (2017)
16. Folz, P., Skaf-Molli, H., Molli, P.: CyCLaDEs: a decentralized cache for triple pattern fragments. In: Sack, H., Blomqvist, E., d'Aquin, M., Ghidini, C., Ponzetto, S.P., Lange, C. (eds.) ESWC 2016. LNCS, vol. 9678, pp. 455–469. Springer, Cham (2016). https://doi.org/10.1007/978-3-319-34129-3_28
17. Taelman, R., Verborgh, R., Colpaert, P., Mannens, E.: Continuous client-side query evaluation over dynamic Linked Data. In: Sack, H., Rizzo, G., Steinmetz, N., Mladenić, D., Auer, S., Lange, C. (eds.) ESWC 2016. LNCS, vol. 9989, pp. 273–289. Springer, Cham (2016). https://doi.org/10.1007/978-3-319-47602-5_44
18. Aasman, J.: AllegroGraph: RDF Triple Database, vol. 17. Oakland Franz Incorporated, Cidade (2006)
19. Thompson, B.B., Personick, M., Cutcher, M.: The Bigdata® RDF graph database. In: Linked Data Management, pp. 193–237 (2014)
20. Erling, O., Mikhailov, I.: Virtuoso: RDF support in a native RDBMS. In: de Virgilio, R., Giunchiglia, F., Tanca, L. (eds.) Semantic Web Information Management, pp. 501–519. Springer, Heidelberg (2010). https://doi.org/10.1007/978-3-642-04329-1_21
21. Apache Jena. https://jena.apache.org/
22. RDFLib. https://rdflib.readthedocs.io/en/stable/
23. rdflib.js. https://github.com/linkeddata/rdflib.js
24. rdfstore-js. https://github.com/antoniogarrote/rdfstore-js
25. Verborgh, R., Dumontier, M.: A Web API ecosystem through feature-based reuse. CoRR. abs/1609.07108 (2016)
26. Lanthaler, M., Gütl, C.: Hydra: a vocabulary for hypermedia-driven web APIs. In: LDOW, vol. 996 (2013)
27. Taelman, R., Verborgh, R.: Declaratively describing responses of hypermedia-driven web APIs. In: Proceedings of the 9th International Conference on Knowledge Capture (2017)
28. Birman, K., Joseph, T.: Exploiting virtual synchrony in distributed systems. ACM (1987)
29. Hewitt, C., Bishop, P., Steiger, R.: Session 8 formalisms for artificial intelligence a universal modular actor formalism for artificial intelligence. In: Advance Papers of the Conference, p. 235. Stanford Research Institute (1973)
30. Gamma, E.: Design Patterns: Elements of Reusable Object-oriented Software. Pearson Education India, Delhi (1995)
31. Fowler, M.: Inversion of Control Containers and the Dependency Injection pattern (2004). https://martinfowler.com/articles/injection.html
32. Taelman, R.: Components.js. http://componentsjs.readthedocs.io/en/latest/
33. Van Herwegen, J., Taelman, R., Capadisli, S., Verborgh, R.: Describing configurations of software experiments as Linked Data. In: Proceedings of the 1st Workshop on Enabling Open Semantic Science (2017)

34. World Wide Web Consortium, et al.: JSON-LD 1.0: a JSON-based serialization for linked data (2014)
35. Grant Clark, K., Feigenbaum, L., Torres, E.: SPARQL 1.1 Query Results JSON Format. W3C (2013). https://www.w3.org/TR/2013/REC-sparql11-results-json-20130321/
36. Hawke, S.: SPARQL Query Results XML Format (Second Edition). W3C (2013). https://www.w3.org/TR/rdf-sparql-XMLres/
37. Prud'hommeaux, E., Seaborne, A.: SPARQL Query Language for RDF. W3C (2008). https://www.w3.org/TR/2008/REC-rdf-sparql-query-20080115/
38. Fernández, J.D., Martínez-Prieto, M.A., Gutiérrez, C., Polleres, A., Arias, M.: Binary RDF representation for publication and exchange (HDT). Web Semant. Sci. Serv. Agents World Wide Web **19**, 22–41 (2013)
39. Taelman, R., Vander Sande, M., Verborgh, R.: OSTRICH: versioned random-access triple store. In: Proceedings of the 27th International Conference Companion on World Wide Web (2018)
40. Aluç, G., Hartig, O., Özsu, M.T., Daudjee, K.: Diversified stress testing of RDF data management systems. In: Mika, P., et al. (eds.) ISWC 2014. LNCS, vol. 8796, pp. 197–212. Springer, Cham (2014). https://doi.org/10.1007/978-3-319-11964-9_13
41. Facebook, Inc.: GraphQL. Working Draft, October 2016. http://facebook.github.io/graphql/October2016/