Ephedra: efficiently combining RDF data and services using SPARQL federation

Andriy Nikolov, Peter Haase, Johannes Trame, Artem Kozlov

metaphacts GmbH, Walldorf, Germany
{an, ph, jt, ak}@metaphacts.com

Abstract. Knowledge graph management use cases often require addressing hybrid information needs that involve multitude of data sources, multitude of data modalities (e.g., structured, keyword, geospatial search), and availability of computation services (e.g., machine learning and graph analytics algorithms). Although SPARQL queries provide a convenient way of expressing data requests over RDF knowledge graphs, the level of support for hybrid information needs is limited: existing query engines usually focus on retrieving RDF data and only support a set of hard-coded built-in services. In this paper we describe representative use cases of *metaphacts* in the cultural heritage and pharmacy domains and the hybrid information needs arising in them. To address these needs, we present Ephedra: a SPARQL federation engine aimed at processing hybrid queries. Ephedra provides a flexible declarative mechanism for including hybrid services into a SPARQL federation and implements a number of static and runtime query optimization techniques for improving the hybrid SPARQL queries performance. We validate Ephedra in the use case scenarios and discuss practical implications of hybrid query processing.

1 Introduction

SPARQL has emerged as a standard formalism for expressing information requests in Semantic Web applications where the goal is to retrieve the data stored as RDF. However, in many practical knowledge graph management use cases there is a need to address *hybrid information needs*. Such needs can be characterized by the following dimensions:

– *Variety of data sources*. There is often a need to integrate data stored in several physical repositories. These repositories can include both native RDF triple stores as well as datasets in other formats presented as RDF (e.g., a relational database exposed using R2RML mappings).
– *Variety of data modalities*. Graph data in RDF often needs to be combined with other data modalities, e.g., textual, temporal or geospatial data. A SPARQL query then needs to support corresponding extensions for full-text, spatial, and other types of search.
– *Variety of data processing techniques*. Retrieved data often has to be further processed using dedicated domain-specific services: e.g., graph analytics (finding the shortest path or interconnected graph cliques), statistical analysis and machine learning (applying a machine learning classifier, finding similar entities using a vector space model), etc.

The main motivation for this work comes from our experience with the *metaphactory* knowledge graph management platform[1], which is used in a variety of application domains (e.g., cultural heritage, life sciences, pharmaceutics, and IoT infrastructure). Typical application scenarios often require dealing with a multitude of the above-listed aspects simultaneously: e.g., an example request like "give me the artists who collaborated with Rembrandt and others similar to them" involves (a) keyword search for an RDF resource based on the keyword "rembrandt", (b) structured search over the RDF graph for collaborators, and (c) applying an external model (vector space similarity) to find other similar entities.

To handle such use case scenarios involving hybrid information needs, we developed Ephedra: a federated SPARQL query processing engine targeted at processing hybrid queries. SPARQL 1.1 with its SERVICE clauses provides a convenient data retrieval formalism: a complex information request over several data sources can be expressed using a single query. However, the existing level of tools support for expressing and processing hybrid information needs using SPARQL is often limited. SPARQL federation implementations usually focus on the first dimension: they assume that federation members are data stores containing RDF triples. Some triple stores also contain built-in implementations of alternative search modalities: e.g., "magic" predicates or even custom language constructs to handle keyword search. With Ephedra we overcome these limitations: while adopting the SPARQL 1.1 federation mechanism, we broaden its usage to include custom services as data sources and optimize such hybrid queries to be executed efficiently.

Serving hybrid information requests using SPARQL raises challenges both at the level of configuring the federation and executing the query: (a) how can we use SPARQL queries to combine retrieval of RDF data with invoking additional hybrid services and (b) how can we execute hybrid SPARQL query services in an efficient way. Expressing a complex hybrid information request using a SPARQL query can be non-trivial due to the variety of potential types of services and the limitations of the SPARQL syntax: e.g., a service can take as input a single set of parameters or a list of arbitrary length; it can return as output one value, several values or a table of multiple records, etc.

Processing such service calls in the same way as genuine SPARQL endpoints would result in sub-optimal query runtimes or even failing queries. This is caused by the specific features of the service calls such as:

- *Input and output parameters*. Triple patterns that express a service call do not refer to the actual RDF data structures, but merely denote the values that should be passed as service inputs and the variables to which the service outputs should be bound. It has serious implications for the query evaluation, as they do not follow SPARQL semantics: e.g., join operands cannot be re-ordered, because a service call must only be scheduled after all its input variables are bound.
- *Data cardinalities*. Estimating the cardinality of a tuple expression is important for performing static query optimization and ordering the join operands. While for genuine SPARQL endpoints the selectivity of graph patterns depends on the distribution of data in the underlying repository, for service calls this often depends on the

[1] http://www.metaphactory.com/

service itself: e.g., a keyword search always takes one input and returns a limited set of search results as output.

With Ephedra we take these factors into account and use them to support practical hybrid federation use cases of the *metaphactory* platform and address hybrid information needs in an efficient way. In this paper, we are making the following contributions:

- We describe two representative use case scenarios from different domains (cultural heritage and pharmaceutics) which present hybrid query processing challenges.
- We propose a reusable architecture in which hybrid services can be easily plugged in, described in a declarative way, and invoked using federated SPARQL queries.
- Based on the explicit descriptions of services, we propose a number of static SPARQL query optimization techniques to construct a valid and efficient query plan.
- We utilize dynamic query optimization to execute hybrid queries with SERVICE clauses in minimal time, improving the performance by up to an order of magnitude.
- We validate our approach in two use cases and discuss the practical implications.

The rest of this paper is structured as follows. Section 2 presents two representative use cases and the requirements for our work. Section 3 describes the general architecture of Ephedra and the meta-level descriptions of services available via SPARQL. Section 4 discusses the query algebra extensions for hybrid SPARQL queries and the relevant static query optimization techniques aimed at generating an optimal query execution plan. Section 5 outlines the techniques Ephedra applies at runtime to modify the execution plan on-the-fly based on the actual execution progress. Section 6 reports the experiments we performed in order to validate our approach. In section 7, we provide an overview of existing solutions for processing federated hybrid SPARQL queries. Finally, section 8 concludes the paper and discusses directions for future work.

2 Use cases and challenges

The *metaphactory* platform is used in production in a variety of scenarios involving knowledge graph management in different application domains. Retrieval of stored RDF data using structured SPARQL queries is often insufficient to satisfy the application requirements without invoking additional services. In the following we consider two practical use cases from the cultural heritage and pharmaceutics domains. For reproducibility, they are based on publicly accessible data sets and resources, they are however representative examples akin to production use cases based on closed, proprietary data.

2.1 Use case: Cultural heritage

The CIDOC-CRM ontology[2] became a popular standard for exposing cultural heritage information as linked data. The *metaphactory* platform is utilized in the context of the ResearchSpace project[3] to manage the British Museum knowledge graph and help the researchers explore meta-data about museum artifacts: historical context, associations

[2] http://www.cidoc-crm.org/
[3] http://www.researchspace.org/

with geographical locations, creators, discoverers and past owners, etc. A crucial piece of functionality is *structured search*, where the user can construct a query request like "give me all bronze artifacts created in Egypt between 2500BC and 2000BC" that gets translated into SPARQL and answered using backend data. However, beyond querying the British Museum collections, the use case requirements also involve addressing hybrid information needs:

- *Variety of data sources*: Datasets of other museum collections structured using CIDOC-CRM as well as linked public RDF data sources (e.g., Wikidata [4]).
- *Variety of data modalities*: Some relevant data sources are also associated with keyword search services (e.g., Wikidata search API or an external Solr index) and geospatial search indices. The Wikidata search API, for example, provides a higher quality of retrieved results than a built-in triple store index, but is only available as a REST web service not directly accessible via SPARQL.
- *Variety of data processing techniques*: A custom semantic similarity search service based on a word2vec vector space model.

This scenario includes a specialized data processing service, which applies a trained machine learning model to find entities similar to a given set of other entities. This service utilizes the word2vec vector space model [1] trained on the English Wikipedia corpus. Each Wikidata entity is represented as an *embedding vector* of length 50. Similarity defined as a distance in the embeddings vector space serves as a means to indicate relatedness between entities and complements the explicit relations stored in the RDF triple store. An example request that can be run over such a federation can look like: "Give me other artists similar to the ones who collaborated with Ibaya Kyubei".

```
# Example query (Q1_CH) from the cultural heritage use case
SELECT ?collabWikidataIRI ?label ?artist WHERE {
  SERVICE metaphacts:wikidataSearch {
      ?wikidataIRI wikidata:search "ibaya" .
  }
  SERVICE <http://public.researchspace.org/sparql> {
    ?bmIRI skos:exactMatch ?wikidataIRI .
    ?collabBMIRI rs:Actor_created_Thing ?artifact .
    ?bmIRI rs:Actor_created_Thing ?artifact .
    ?collabBMIRI skos:exactMatch ?collabWikidataIRI .
    FILTER (?bmIRI != ?collabBMIRI)
  }
  ?collabWikidataIRI rdfs:label ?label .
  SERVICE metaphacts:wikidataWord2Vec {
     ?collabWikidataIRI word2vec:similarTo ?artist .
  }
}
```

Such a query requires the query engine to federate over two different data repositories (ResearchSpace and Wikidata) as well as include information from two external services: Wikidata keyword search API and word2vec semantic similarity search.

2.2 Use case: Pharmaceutics

Another scenario in which the *metaphactory* platform is employed is related to the pharmaceutics domain. The internal knowledge graph of the customer contains interlinked

[4] http://www.wikidata.org

data about genes, proteins, and associated diseases. This information, however, has to be augmented with other sources which makes this use case another example of hybrid information needs:

- *Variety of data sources*: Additional relevant data sources include an Oracle relational database exposed as a SPARQL endpoint using R2RML/Ontop as well as relevant public RDF data sources (Wikidata and Nextprot [5]).
- *Variety of data modalities*: Full-text indices enable custom keyword search in addition to the SPARQL structured search.
- *Variety of data processing techniques*. Domain-specific services include the trained machine learning models realized in the KNIME [6] data science platform as well as the BLAST [2] web service to find similar entities based on the genome sequence data.

An example hybrid search request can look like "Give me the gene encoding the reelin protein and other genes having the most similar sequences". Such a simple request would involve all three hybrid search dimensions: keyword search service to retrieve the id of the "reelin" protein, query over the RDF graph to find an associated gene and the ID of the sequence, and a call to the BLAST service to find similar sequences.

```
# Example query (Q1_PH) from the pharmaceutics use case
SELECT * WHERE {
  SERVICE metaphacts:wikidataSearch {
      ?uri wikidata:search "reelin" .
  }
  ?uri wdt:P702 ?gene .
  ?gene wdt:P639 ?refseqID .
  SERVICE ncbi:BLAST {
      ?refseqID blast:hasSimilarSequence ?y .
  }
}
```

These common challenges arising in diverse use cases necessitate the use of a generic approach for hybrid query processing. Extending the SPARQL 1.1 federation mechanism for that has important advantages:

- A hybrid information need can be expressed in a fully declarative way without the need for use case-specific custom implementations.
- The use of the standard SPARQL 1.1 syntax makes the approach compatible with third-party tools (e.g., client-side SPARQL processing libraries).

The solution we present focuses on keeping these advantages while maintaining the system reusable in different scenarios and processing the hybrid queries in the most efficient way.

3 Hybrid SPARQL federation

To support the use cases described above, we developed the Ephedra query processing engine as a part of the *metaphactory* platform. A crucial requirement for enabling the hybrid querying functionality is the ability to plug in additional services with minimal effort and reference them from SPARQL.

[5] https://www.nextprot.org/

[6] https://www.knime.org/

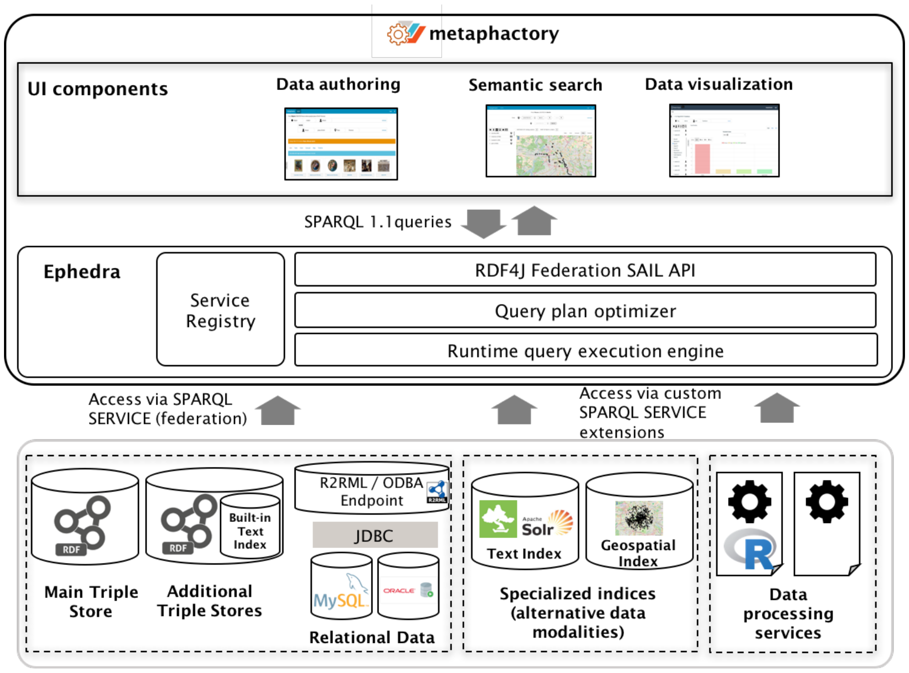

Fig. 1. Ephedra in the *metaphactory* platform architecture.

3.1 Ephedra architecture

Figure 1 shows the generic architecture of the *metaphactory* platform. Ephedra is used as a hybrid query federation layer to access the data repositories and services. In the course of the DIESEL project [3], we developed a set of structured search components which allow the user's hybrid information need to be captured interactively: the user can define search clauses, explore partial results, incrementally add new clauses, while the system provides relevant suggestions. These interactions generate information requests that are expressed as SPARQL 1.1 queries by the UI components and given to Ephedra to process them.

As the basis for Ephedra implementation, we used the RDF4J Federation SAIL API[7] reusing the common functions such as query parsing and accessing remote SPARQL endpoints. However, Ephedra extends the RDF4J object model and overrides the static optimization and query execution strategies to deal with hybrid queries. The Ephedra query evaluation strategy sends the sub-clauses of the query to the corresponding data sources and invokes the relevant processing services, then gathers the partial results, combines them using the union and join operations, and produces the final result set. In this way, processing becomes transparent: hybrid information needs are processed in the same way as ordinary SPARQL queries to an RDF triple store without the need to integrate related processing services at the UI level.

[7] http://docs.rdf4j.org/sail/

3.2 Describing and configuring hybrid services

In order to configure the services as federation members, the system requires relevant information about the *service type* as well as *service instances*.

Ephedra includes two types of hybrid services: *extension services* and *aggregate services*. Extension services take as input a partial query solution (binding set) and extend it with additional variable bindings. Extension services are called in the query via a SPARQL SERVICE clause. On the contrary, aggregate services operate over a set of multiple query solutions as the SPARQL aggregate functions (e.g., AVG, MIN, MAX) do: they take as input a list of records and produce one or more resulting binding sets. As with the SPARQL aggregates, aggregate services are referenced as function calls in the SELECT clause.

Relevant meta-level information about the hybrid service types is summarized using the service descriptors structured according to the *service description ontology*. The ontology expands the well-known SPIN[8] ontology for SPARQL query engines to capture the relevant parameters of services.

A service descriptor contains the following information:

- Input parameters and their expected datatypes. An input parameter is described using the SPIN ontology vocabulary as a *spl:Argument* resource.
- Output parameters and their expected datatypes. An output parameter is described as a *spin:Column* resource in the SPIN ontology.
- Expected graph pattern. The special triple patterns expected by the service are expressed using the SPIN SPARQL syntax[9]. The placeholders for input/output parameters are expressed as resources which are referenced from the input/output parameter descriptors.
- Input and output cardinalities of a service call (optional).

```
:WikidataTextSearch a eph:Service ;
  rdfs:label "A wrapper for the Wikidata test search." ;
  eph:hasSPARQLPattern (
      [ sp:subject :_uri ;
          sp:predicate wikidata:search ;
          sp:object :_token ] ) ;
  spin:constraint
  [ a spl:Argument ;
    rdfs:comment "Input token" ;
    spl:predicate :_token ;
    spl:valueType xsd:string ] ;
  spin:column
  [ a spin:Column ;
    rdfs:comment "URI of the Wikidata resource" ;
    spl:predicate :_uri ;
    spl:valueType rdf:Resource ] .
```

A descriptor for an aggregation service declares the input and output parameters in a similar way, but instead of the list of triple patterns it defines a custom aggregate function which will be referenced by its URI.

[8] http://spinrdf.org/

[9] http://spinrdf.org/sp.html#sp-TriplePattern

```
:word2vec a eph:AggregateService ;
  rdfs:label "A wrapper for the word2vec similarity aggregate service ." ;
  eph:hasAggregateFunction
  [ a sp:Aggregate ;
    sp:expression :_uri ;
    sp:as :_similar ; ] ;
  spin:constraint
  [ a spl:Argument ;
    rdfs:comment "Entity URI" ;
    spl:predicate :_uri ;
    spl:valueType rdfs:Resource ] ;
  spin:column
  [ a spin:Column ;
    rdfs:comment "URI of the similar entity" ;
    spl:predicate :_similar ;
    spl:valueType rdf:Resource ] .
```

3.3 Implementing service extensions

To simplify the integration of new hybrid services into the framework, the architecture provides a generic API to wrap arbitrary services and include them as SPARQL federation members. To this end, Ephedra reuses and extends the RDF4J SAIL API. A service is represented as a SAIL module which is responsible for extracting the values of input parameters from a given SPARQL tuple expression, executing the actual service call, and returning the results by binding resulting values to the output variables. Ephedra provides abstract implementations for a generic service SAIL as well as a specific wrapper for REST services. The common routines, such as extracting the input values and output variables and wrapping the results as binding sets do not depend on the actual service and are performed in a generic way using the declarative service descriptor.

A service instance is thus configured in the same way as a standard RDF4J repository: its descriptor contains a pointer to the service type as well as the specific parameters of the service installation (e.g., the URL by which the REST service can be accessed).

4 Adapting SPARQL algebra for hybrid queries

On receiving a federated query, Ephedra has to create a suitable query plan for its execution. Although Ephedra adheres to the SPARQL 1.1 syntax, processing of hybrid queries requires introducing special algebra elements to handle the extension and aggregate services. Based on these, Ephedra is able to construct a suitable query plan that would avoid failing queries and reduce the subsequent execution time.

4.1 Basic definitions

In a SPARQL query, the WHERE clause defines a *graph pattern* to be evaluated on an RDF graph G. An atomic graph pattern is a *triple pattern* defined as a tuple P from $(I \cup V) \times (I \cup V) \times (I \cup L \cup V)$, where I, L, and V correspond to the sets of IRIs, literals, and variables respectively. Triple patterns are combined by means of JOIN, UNION, FILTER, and OPTIONAL operators to construct arbitrary graph patterns. A *mapping* is defined as a partial function $\mu : V \rightarrow (I \cup L \cup B)$ (B is a set of blank nodes) [4], and the domain of the mapping $dom(\mu)$ expresses a subset of V on which the mapping is defined. Then, the semantics of SPARQL queries is expressed by means of a function $[\![P]\!]_G$, which takes as input a graph pattern P and produces a set of mappings from the set of variables $var(P)$ mentioned in P to elements of the graph G. The binding of the variable $?x$ according to the mapping μ is denoted as $\mu(?x)$. The basic query algebra then defines the standard operations (Selection σ, Join $\bowtie$, Union $\cup$, Difference $\setminus$, and Left Join $⟕$) over the sets of mappings, and query evaluation involves translating the query into a *query tree* composed of these operations. For simplicity, in this paper we use the notation $P_1 \bowtie P_2$ to refer to the join operation over sets of mappings produced by the patterns P_1 and P_2. In order to allow hybrid queries to be optimized, we need to introduce hybrid service calls at the level of SPARQL algebra.

4.2 Service clauses in SPARQL algebra

To handle extension services, Ephedra introduces the notion of a *service call pattern* as a special graph pattern in the query tree.

Definition 1*: A service call pattern Σ^S is a tuple $(id, P^S, f_e^S, m^i, m^o)$ identified by an IRI id and characterized by the following parameters:*

- *Graph pattern P^S: a SERVICE clause by which the service call is expressed in the SPARQL query.*
- *Function $f_e^S : D^i \rightarrow D^o$: the function implemented by the service, which takes a list of input parameters $D^i = \{d^i\}$ and produces a list of output results $D^o = \{d^o\}$*
- *Input parameter mappings m^i: set of mappings from the elements of P^S to elements of D^i, which extract the values of service input parameters from the graph pattern P^S.*
- *Output variable mappings m^o: set of mappings from the elements of $D^o \cup d_{rank}$ to the elements of $var(P^S)$. One special case of a service output is d_{rank}: the rank of the returned result, which can be added implicitly to each result returned by f_e^S.*

Aggregate services represent a different case: since they can be applied to multisets of partial results, they are expressed at the syntax level in the same way as standard SPARQL aggregate operations such as COUNT, AVG, or SUM. To include the aggregate service expressions, we extend the aggregate algebra construct defined in [5]:

Definition 2*: A service aggregate A^S is a construct of the form $Aggregate(\mathbf{F}, f_a^S(\mathbf{D^i}), \Gamma)$, where*

- *$\Gamma = Group(\mathbf{E}, P)$ is a GROUP operator over the graph pattern P using a list of expressions $\mathbf{E}$.*
- *$f_a^S(\mathbf{D^i})$ is an aggregation function implemented by the service, which takes as input a multiset of parameter assignments $\mathbf{D^i} = \{D^i\}$, where $D^i = \{d_1^i \dots d_n^i\}$, and produces as output a set of output values $\mathbf{d^o}$.*
- *$\mathbf{F}$ is a list of expressions which are applied to the results of Γ to produce the inputs of f_a^S.*

In Ephedra we extended the standard SPARQL 1.1 semantics to incorporate aggregate services which can produce as a result a set of values as well as a single value. One example of such services is the word2vec service we use in the cultural heritage use case: it can take a set of several entities as input and produce a list of additional entities that are similar to the whole set.

4.3 Building a query plan

After processing the parsed hybrid query and replacing the default SERVICE clauses with service call patterns Σ^S and service aggregates A^S, the Ephedra query engine tries to build an optimal execution plan for the query. Ephedra focuses on two types of improvements for the query:

– Join order optimization
– Assigning appropriate executors for JOIN and UNION operators

Determining the order in which join operators have to be processed as well as choosing appropriate execution algorithms (e.g., nested loop join vs hash join) may have very significant impact on the execution performance and so have been in the focus of research when designing the federation query engines. Traditionally, sorting the join order operands takes into account the estimated selectivity of the join parameters.

In case of hybrid queries, this step carries additional importance: an unoptimized query may not be executable at all. The query engine must ensure that all input parameters of a service call group Σ^S are bound before the service is called. Moreover, some join operators can be inappropriate for use with hybrid service clauses: e.g., a service clause which has unbound input parameters must be executed as a second argument in a nested loop join and cannot be processed by a hash join.

When processing an n-ary join operation, Ephedra groups together the join operands which can be executed at the same source. After that, it uses the following criteria (starting from the most important) to determine whether an operand tuple expression P can be added to the pipeline:

1. Join operands which contain service call patterns Σ^S are added immediately when all their input dependencies become satisfied.
2. A join operand P_x which binds a variable v_{xi} cannot be added if there exists a join operand $P_y = \Sigma^S_y(P^S, f^S_e, m^i, m^o)$ such that $v_{xi} \in m^o(P^S)$: if the same variable can be bound in a service call pattern and in an ordinary graph pattern, the service call is executed first.
3. A join operand P_x is preferred over P_y if $var(P_x)$ contains a variable bound earlier in the pipeline and $var(P_y)$ does not.
4. Finally, the estimated selectivities are taken into account. Selectivity is estimated based on the number of free variables (for the ordinary graph patterns) or based on service descriptors (for hybrid service calls).

The second technique used by Ephedra involves processing of the *top-k* queries and handling of the results' ranking. To this end, Ephedra uses the $\mathcal{S}$PARQL-$\mathcal{R}$ANK algebra [6].

5 Optimizing hybrid queries execution

While static query optimization helps to reduce query execution time, the resulting query performance still strongly depends on the way the operators are processed. The power of static optimization in a hybrid federation is particularly limited, because precise selectivity estimation is impossible. Reversing the order of operands in a nested loop join or replacing it with a hash join can significantly improve the performance. To further minimize the query execution performance, Ephedra uses two techniques: *synchronizing loop join requests* and *adaptive processing of n-ary joins*.

5.1 Synchronizing loop join requests

Let us consider our example query Q1_CH from section 2.1. The top-level N-ary join of the query contains 4 operands: Σ_1^S (Wikidata text search), Σ_2 (British Museum), Σ_3 (Wikidata) and Σ_4^S (word2vec). When executing a nested loop join, the engine first retrieves the answers μ_i for Σ_1^S. Iterating over μ_i, it will probe Σ_2 to receive bindings μ_{ij} from $(\Sigma_1^S \bowtie \Sigma_2)$ and continue doing this until receiving the complete answers from $(\Sigma_1^S \bowtie \Sigma_2 \bowtie \Sigma_3 \bowtie \Sigma_4^S)$. When iterating over partial result sets μ_i, there are several possible strategies which can be chosen for sending the probing queries q_{ij} to join the next operand Σ_{i+1}.

1. *Synchronous vs asynchronous*. The engine can parallelize sending of the probing queries q_{ij} so that the answers are added into the resulting queue as soon as they appear. Alternatively, it can synchronize the requests to guarantee that the results produced by the query q_{ij} will appear in the final result set before the results of q_{ik} if $j < k$.
2. *Separate requests vs batch*. The engine can send a separate query request for each input binding set μ_{ij}. Another strategy called *Bind Nested Loop Join (BNLJ)* [7] involves grouping together several mappings $\mu_{ij}, \dots, \mu_{ij+K}$ and sending a single query which would contain all bindings from the group expressed using a VALUES clause.

The choice of the appropriate strategy depends on the type of the query as well as the types of each join operand. The batch processing strategy using the BNLJ operator helps to reduce the number of potentially expensive remote requests and was shown to improve the performance significantly [7]. However, it can only be applied if the right join operand is an RDF repository: by default, a hybrid service can only process one set of input parameters at a time and can even break otherwise. Ephedra uses the service descriptors to choose an appropriate strategy: e.g., in our example query it will use the BNLJ strategy to join the data from the British Museum and Wikidata repositories, but send the requests to the word2vec service separately.

One additional technique used by Ephedra that helps to reduce the number of expensive requests to remote hybrid services involves caching the probing queries. In our example, the last operation in the pipeline involves joining the results of $\Sigma_1^S \bowtie \Sigma_2 \bowtie \Sigma_3$ with the word2vec service Σ_4^S using the join variable *?collabWikidataIRI*. The results of the previous operations in the pipeline contain multiple binding sets which have the same value bound to *?collabWikidataIRI*: the same pair of artists (Ibaya Kyubei and Utagawa Kuniyoshi) has collaborated on many woodblock prints. Processing each binding set separately would result in many requests to the *word2vec* service for the same input value (Utagawa Kuniyoshi). To avoid this, when performing a join between ($\Sigma_i^S \bowtie \Sigma_2$) and Σ_3^S, we only send queries for unique key combinations (variables present in both operands). The remote query results are then joined to each relevant binding μ_{ij} that share the same values for key variables. This is equivalent to applying the REDUCED operation implicitly to the set of key combinations from μ_{ij}.

5.2 Adaptive processing of n-ary joins

Sometimes, an n-ary join contains multiple service call groups that do not have unbound inputs. It means that the evaluation of the join can start from both these groups independently. Let us suppose that our query contains 3 join operands: Σ_1^S, Σ_2, and Σ_3^S, where both Σ_1^S and Σ_3^S have all their input parameters bound. This n-ary join can be executed using different plans, for example $(\Sigma_1^S \bowtie_{BNLJ} \Sigma_2) \bowtie_{HJ} \Sigma_3^S$ or $\Sigma_1^S \bowtie_{HJ} (\Sigma_2 \bowtie_{BNLJ} \Sigma_3^S)$. It is not always known in advance, which of these plans is preferable: an incorrect choice can results in big differences in execution time. The *parallel competing join* strategy originally presented in [8] to handle keyword search queries tries to avoid this by executing the competing query plans in parallel and making the final choice between them at runtime.

Algorithm 1 Parallel competing n-ary join

```
$\mathbf{P^s}$: seed operands
$\mathbf{P^d} \leftarrow \mathbf{P} \backslash \mathbf{P^s}$: other operands
for all $P_i^s \in \mathcal{P}$ do
    $Q_i \leftarrow (\{P_i^s\} \cup \mathbf{P^d})$
    start($Q_i$)
...: wait until $\mathbf{P^d} = \emptyset$
if $\mathcal{P}^{\supsetneqq} = \emptyset$ then
    return HashJoin($\{P_i\}$)
...
procedure PUSHRESULTS($P_{curr}$, $P_{prev}^d$, $[\![P_{curr}]\!]_G$)
    $\mathbf{P^d} \leftarrow \mathbf{P^d} \backslash P_{prev}^d$
    for all $Q_i$ do
        $Q_i \leftarrow Q_i \backslash P_{prev}^d$
    $P_{next}^d \leftarrow Q_{curr}.next$
    if joinInThisPlan($P_{curr}$, $P_{next}^d$) then
        $P_{curr} \leftarrow$NestedLoopJoin($P_{curr}$, $P_{next}^d$)
```

The algorithm starts with selecting the "seed" join operands, which serve as starting points for alternative query plans (Σ_1^S and Σ_3^S in our example). For each of these seeds, an alternative join sequence Q_i is produced and triggered. Whenever any of the competing query plans Q_i completes the join of some operand P_{prev} and produces a partial result set, a re-evaulation takes place. In the default case, it checks if the partial result set is too large and continuing the plan Q_i is likely to be more expensive than an alternative plan. If the check passes, the next operand from Q_i is joined, otherwise the plan stops.

Once all partial query plans are finished and no operand P_i remains unprocessed, Ephedra joins the partial results of all query plans via an n-ary hash join. Alternatively, if the ranking must be preserved, the n-ary Pull/Bound Rank Join algorithm [9] can perform the final join.

6 Evaluation and discussion

In order to validate Ephedra, we used data and queries from our two representative use cases from section 2. In the cultural heritage setup, we used two RDF data repositories (British Museum (BM) and Wikidata), the Wikidata entity lookup REST API (WD-text), and the *word2vec* vector space model similarity REST API. The latter was used both as an extension service (to retrieve instances most similar to a single input one) and as an aggregation service (to retrieve instances similar to a group of input entities). The model was trained on the English Wikipedia corpus using gensim[10]. In the pharmaceutics setup, we used two public RDF repositories (Wikidata and Nextprot), Wikidata entity lookup API (WD-text), and a wrapper around the public BLAST API [11].

The test queries for two domains were selected in such a way that (a) they were representative examples of queries arising in practical use and (b) each one covered at least two hybrid query dimensions. We used four queries from the cultural heritage domain and three queries from the pharmaceutics domain[12]. We compared the average query execution runtimes with disabled and enabled optimization techniques described in sections 4 and 5. Table 1 shows the runtime performance for each query, averaged over five runs. As we can see, query optimization techniques of Ephedra led to improvements

Query	Sources	Baseline		Ephedra	
		Time (sec)	σ	Time (sec)	σ
Q1_CH	Wikidata, WD-text, BM, word2vec	10.20	0.42	1.40	0.19
Q2_CH	Wikidata, WD-text	1.56	0.16	0.73	0.25
Q3_CH	Wikidata, BM	13.17	0.13	4.52	0.90
Q4_CH	Wikidata, WD-text, BM, word2vec (aggregate)	4.40	0.49	1.28	0.03
Q1_PH	Wikidata, WD-text, BLAST	12.38	0.76	2.24	0.28
Q2_PH	Wikidata, WD-text	1.50	0.25	0.70	0.02
Q3_PH	Wikidata, WD-text, Nextprot	3.54	0.06	1.31	0.36
Geom. Mean		4.82		1.43	

Table 1. Average execution time (sec) for test queries taken over 5 query runs.

in the query evaluation runtimes for all test queries, sometimes by an order of magnitude. The factors which contributed the most were:

- Processing of loop join requests, where combining asynchronous processing with the bound join operator resulted in the best performance of n-ary joins.
- Competing nested loop join, which was beneficial for n-ary joins with several potential starting points (Q2_CH and Q2_PH).
- Caching of probing requests, which helped to avoid expensive redundant remote service calls (Q1_CH and Q1_PH).

Our validation experiments have shown that having special optimization techniques that treat hybrid service calls differently from "native" SPARQL endpoints can substantially benefit performance. This enabled Ephedra to fulfill the requirements of the use cases by maintaining acceptable response times of the *metaphactory* platform.

[10] https://radimrehurek.com/gensim/

[11] https://ncbi.github.io/blast-cloud/dev/api.html. In the tests we only measured the time required to register a search request, since complete processing of the request takes ~1 min and varies greatly depending on the public server workload.

[12] The queries are available online on https://github.com/metaphacts/ephedra-eval

Beyond that, some of the lessons we learned concern more pragmatic aspects. One such conclusion is that conformance to the SPARQL standard helps both to improve reusability and reduce the maintenance effort. Sometimes, tools and triple stores introduce syntax-level modifications of SPARQL to realize the hybrid query functionalities: e.g., special syntax for keyword search clauses or graph analytics queries. Given that SPARQL processing must be performed at different layers (client-side, query federation, backend repository) using different libraries, special syntax changes become particularly difficult to handle and severely increase the solution building costs. Instead, introducing special interpretations of standard language concepts (e.g., "magic" predicates, custom functions) without changing the language syntax is preferable.

7 Related Work

There are several approaches that focused on specific dimensions of hybrid query processing. Main triple store implementations, such as Blazegraph[13], Virtuoso[14], GraphDB[15], and Stardog[16], take the SPARQL 1.1 standard as the basis for supporting federated queries. Usually, they share the assumption that federation members represent remote RDF repositories and they do not maintain meta-level information about federation members. Some triple stores (e.g., Blazegraph) also provide interfaces for adding custom service extensions. However, to our knowledge, they are not treated differently from remote SPARQL endpoints. Alternative data modalities (e.g., full-text and geospatial search) are supported using specialized built-in indices and expressed in SPARQL using "magic" predicates or SPARQL syntax modifications (e.g., full-text search in Virtuoso and Stardog). This makes it difficult to develop reusable database-independent solution applications.

Specialized SPARQL federation engines focus on optimal processing of distributed SPARQL queries. They usually maintain meta-level information about federation members which helps them to build an optimal query plan (e.g., SPLENDID [10], ANAPSID [11], or HiBISCuS [12]) as well as use special runtime execution techniques targeting remote service queries (e.g., FedX [7]), but still focus on interacting with RDF repositories rather than services.

In contrast, SCRY [13] and Quetzal-RDF [14] deal with calling data processing services using SPARQL queries. Quetzal-RDF defines custom functions and table functions (generalized aggregation operations) and invokes them from a SPARQL query, but does not follow the SPARQL 1.1 syntax. SCRY conforms to SPARQL 1.1 using special GRAPH targets to wrap service invocations, although it cannot distinguish between multiple input/output parameters. None of these systems, to our knowledge, applies optimizations targeted at reducing the hybrid queries' execution time.

[13] http://www.blazegraph.com
[14] http://virtuoso.openlinksw.com
[15] http://ontotext.com/products/graphdb/
[16] http://www.stardog.com/

8 Conclusion

Our approach to the problems of handling hybrid queries was motivated by the requirements of commercial use case scenarios in two different domains. The design choices of Ephedra were influenced by the need to maintain the platform reusability and minimize the effort needed to develop and deploy a solution for a new use case. In this respect, expanding the intended usage area of SPARQL queries to express hybrid information needs while maintaining the conformance to SPARQL 1.1 helped us to achieve this goal. The main directions for the future work concern further minimizing the adaptation effort needed to deploy *metaphactory* in a new use case. This involves, for example, building a library of reusable data analytics services (e.g., for common machine learning algorithms).

Acknowledgements

This work has been supported by the Eurostars project DIESEL (E!9367) and by the German BMWI Project GEISER (project no. 01MD16014).

References

1. Mikolov, T., Sutskever, I., Chen, K., Corrado, G.S., Dean, J.: Distributed representations of words and phrases and their compositionality. In: Advances in neural information processing systems. (2013) 3111–3119
2. Altschul, S.F., Gish, W., Miller, W., Myers, E.W., Lipman, D.J.: Basic local alignment search tool. Journal of Molecular Biology **215**(3) (1990) 403 – 410
3. Usbeck, R., Röder, M., Haase, P., Kozlov, A., Saleem, M., Ngomo, A.N.: Requirements to modern semantic search engine. In: KESW 2016. (2016) 328–343
4. Pérez, J., Arenas, M., Gutierrez, C.: Semantics and complexity of SPARQL. ACM TODS **34**(3) (2009)
5. Kaminski, M., Kostylev, E.V., Grau, B.C.: Semantics and expressive power of subqueries and aggregates in SPARQL 1.1. In: WWW 2016. (2016) 227–238
6. Magliacane, S., Bozzon, A., Valle, E.D.: Efficient execution of top-k SPARQL queries. In: ISWC 2012, Boston, USA (2012) 344–360
7. Schwarte, A., Haase, P., Hose, K., Schenkel, R., Schmidt, M.: FedX: Optimization techniques for federated query processing on linked data. In: ISWC 2011, Bonn, Germany 601–616
8. Nikolov, A., Schwarte, A., Hütter, C.: FedSearch: Efficiently combining structured queries and full-text search in a SPARQL federation. In: ISWC 2013. (2013) 427–443
9. Schnaitter, K., Polyzotis, N.: Optimal algorithms for evaluating rank joins in database systems. ACM Transactions on Database Systems **35**(1) (2008)
10. Görlitz, O., Staab, S.: Splendid: Sparql endpoint federation exploiting void descriptions. In: COLD2011, at ISWC 2011, Bonn, Germany (2011)
11. Acosta, M., Vidal, M.E., Lampo, T., Castillo, J., Ruckhaus, E.: ANAPSID: An adaptive query processing engine for SPARQL endpoints. In: ISWC 2011
12. Saleem, M., Ngonga Ngomo, A.C.: HiBISCuS: Hypergraph-based source selection for SPARQL endpoint federation. In: ESWC 2014. (2014)
13. Stringer, B., Meroño-Peñuela, A., Abeln, S., van Harmelen, F., Heringa, J.: SCRY: extending SPARQL with custom data processing methods for the life sciences. In: SWAT4LS. (2016)
14. Dolby, J., Fokoue, A., Rodriguez-Muro, M., Srinivas, K., Sun, W.: Extending SPARQL for data analytic tasks. In: ISWC 2016. (2016) 437–452