Supporting virtual integration of Linked Data with just-in-time query recompilation

Alessandro Adamou
Knowledge Media Institute, The Open University
alessandro.adamou@open.ac.uk

Mathieu d'Aquin
Insight Centre, NUI Galway
mathieu.daquin@insight-centre.org

Carlo Allocca
Knowledge Media Institute, The Open University

Enrico Motta
Knowledge Media Institute, The Open University
enrico.motta@open.ac.uk

ABSTRACT

Virtual data integration takes place at query execution time and relies on transformations of the original query to many target endpoints, where the data reside. In systems that integrate many data sources, this means maintaining many mappings, queries and query templates, as well as possibly issuing separate queries for linking entities in the datasets and retrieving their data. We propose a practical approach to keeping such complexity under control, which manipulates the translation from one client query to many target queries. The method performs just-in-time recompilation of the client query into elements that are combined with a query template into the target queries for multiple sources. It was validated in a setting with a custom star-shaped query language as client API and SPARQL endpoints as sources. The approach has shown to reduce the number of target queries to issue and of query templates to maintain, using a number of compiler functions that scales with the complexity of the data source, with an overhead that may be neglected where the method is most effective.

KEYWORDS

data integration, Linked Data, global-as-view, just-in-time

ACM Reference format:
Alessandro Adamou, Mathieu d'Aquin, Carlo Allocca, and Enrico Motta. 2017. Supporting virtual integration of Linked Data with just-in-time query recompilation. In *Proceedings of 13th International Conference on Semantic Systems, Amsterdam, The Netherlands, September 11–13 (SEMANTiCS17),* 8 pages.
https://doi.org/10.1145/3132218.3132227

1 INTRODUCTION

In **data integration**, a designated party (the integrator) re-publishes data that originate on other systems, typically assumed to be beyond their control, onto another controlled central system. This involves, in general, a target schema for the data (*global schema*) which is designed by the integrator and, if this differs from the schemas of the data on their origins (*source schemas*), the application of transformation rules from source schemas to the global schema. Data integration methods can be categorised with respect to their way of enforcing global schema compliance (hard constraints vs. *schemaless* data integration) and mapping specification strategies (*"global-as-view"* vs. *"local-as-view"*) [4]. Another distinction is between materialised data integration, or *data warehousing*, and *virtual data integration.* In the former, there is an extract-transform-load (ETL) process that makes all the data available on the central system; in the latter, there is no bulk loading of the transformed data, as they still only (or mostly) reside on their origins [3]. Virtual data integration is also called *mediated*, because it uses a component (the mediator) responsible for fetching from each source the data *that are needed, when they are needed*, transforming them at the time the query is executed and delivering them. Note here a slight difference between virtual data integration and data federation: in database literature, federation is when *any* of the participating nodes can serve as the integrator, because all the nodes are autonomous and equally capable of performing integration [23]: the results of federated SPARQL queries are often handled that way. With mediation, *one* designated node, not necessarily sharing the data model or query language of its sources, adopts this role, which causes it to depend on the others.[1]

Both warehousing and virtual integration have advantages and disadvantages and there is no one-size-fits-all solution: the choice is typically determined by business needs. Although warehousing is generally considered more robust and efficient due to the greater control on the integrator's part, thus seeing greater adoption, there can be several reasons to opt for virtual integration, for example:

- it better fits the economy of projects for whom the costs of maintaining large data warehouses would be unaffordable in comparison to a robust network infrastructure backed by modest storage capacities;
- it addresses by nature the problem of delivering data that are up to date, especially those like sensor data that change frequently;
- it allows a shorter time-to-market of data delivery and accommodates the flexibility of pay-as-you-go integration [15], because basic mappings can be quickly generated to have working support for a new data source, and then refined at relatively little cost based on system or client feedback.

A virtual data integration system (hereinafter VDIS) can be a suitable back-end of a Web API that needs to provide up-to-date

[1] See also Majkic [13] for the distinction between data separation and federation.

data from multiple sources in custom formats, provided it can do so in reasonable times.

1.1 Problem statement

If integrated data are not copied and stored centrally, then they must be retrieved upon request, such as by receipt of a query from a client to an API or user interface on the integration platform. This means that, at the time such a query is about to be executed, the integration system needs to know (a) which data sources to query in its turn; (b) how to identify matching data (entity linking/resolution); and (c) how to retrieve the actual data once identified. Finally, it must map the data received from every queried source to the global schema. Optimising these steps is one of the challenges that virtual data integration faces in order to be a viable solution.

Each of the steps (a,b,c) described above can be implemented by performing lookahead queries, with each query exploring the data source and determining what the next query needs to be. This is what a pure federated system will do, as it typically operates without assuming prior knowledge of the data sources: in that case, query relaxation mechanisms are often in place to generate queries satisfiable by as many data sources as possible. A VDIS instead uses mappings between schemas, which is an application of what prior knowledge one has of the dataset. To use a pay-as-you-go strategy, a VDIS should be able to operate with a relatively small amount of knowledge, so as to refine the mappings in later stages. There is then a problem of keeping the mappings, prototypical queries (used e.g. for exploring unknown data sources) etc. in a manageable amount as the data sources are better studied and understood over time. This is necessary both when formulating the query to issue to each data source, and when transforming the data retrieved from one. This paper concentrates on the query formulation stage.

1.2 Summary of the contribution and outline

We present an approach to realise a VDIS starting with only partial knowledge of the structure of the source dataset. The approach is similar to global-as-view (GAV) mappings, i.e. mappings that generate queries over the schema of the source out of one symbol in the global schema. The mappings are structured as sets of lightweight procedures, each generating a piece of the query in the target language. We call this process *just-in-time recompilation* (JIT), since compared to plain query rewriting it involves elements similar to those found in software compilers, e.g. working across different languages; linking and assembling source objects (compilation units) using intermediate code; optimising the amount of target queries to be sent at execution time (similar to the optimisation of JIT compilers when recompiling parts of the code at runtime).

To evaluate the benefits of the JIT method, we pick an existing query federation system for SPARQL and measure the overhead of adding JIT support to it (therefore giving queries that take knowledge of the dataset into account), against the benefit of making it send fewer queries that, in many cases, return more results than with SPARQL federation alone. The input queries of our experiments are in a simple custom language that accommodates conjunctive queries, which is used in the system built for the project that supports this work. We tested the system with queries adapted from the *FedBench* benchmark to fit this language. For this study, we concentrate on a language for star-shaped queries, which constitute an important subclass that are tough to optimise by their own right, and defer studies on more complex classes to future work.

The remainder of this paper is structured as follows: Section 2 elaborates on existing approaches on virtual data integration and their relation to Linked Data; section 3 describes the concepts and method of our approach, with some notes on the existing implementations in Section 4. In Section 5 we present the experiments and discuss their results, before the closing remarks in Section 6.

2 RELATED WORK

Though not the most frequent integration systems in the Big Data age, VDIS have been gaining consideration as an attractive alternative in the industry, historically more inclined to data warehousing and with a limited range of enterprise data federation solutions. A 2013 white paper introduced the notion of a Virtual Data Machine as a re-deployable set of components able to initiate an integration environment out of simply configuring access to data sources [10]. Industrial stakeholders such as Denodo, SAP and Rocket Software have since followed suit, according to a recent Gartner report [25], which predicts significant adoption of data virtualisation by 2020.

For the above reasons we deem this to be an appropriate time for investigating the practicalities of how Semantic Web research may both benefit from, and contribute to, this strand of research on data lifecycles. Past research has investigated ontology languages as a core element for handling mappings between data [5], though most accounts in virtual data integration use ontologies only to mediate between schemas once the query results are received [1, 2, 21]. Approaches such as that by Ponte Vidal et al. also considered ontological mappings for rewriting queries [16]. More recently, the tendentiously open nature of Linked Data has favoured the rise of techniques and systems that extend SPARQL federation with algorithms that, to a degree, consider the prior knowledge of the datasets. *LAW* calculates query plans based on property link graphs, which contain information of the (inferred) domains and ranges of properties as they are used in each dataset [12]. *SPLENDID* has heuristics that leverage the content of VoID[2] descriptions [7]. While these approaches already provide valid optimisations, it still takes a more general framework to accommodate other kinds of prior knowledge of a datasets, such as the existence of mappings between the URI patterns of entities and their names, something for which the semantics of SPARQL alone can only provide limited support.

Still on applying prior knowledge of the data, we refer to a major inspiration for our work, i.e. a contribution by Howe et al. on schemaless profiling of unfamiliar data sources [9]. Their work elaborates upon the notion of *dataspaces*, which are lightweight environments of information sources with minimum guaranteed capabilities (e.g. data cataloguing) [6] paired with techniques for iteratively assessing the utility and compatibility of the data therein.

Steps towards enlarging the role of data source configurations (or *manifests*) were taken with Networked Graphs [19], which allowed the definition of queries for entity linking, if only at instance level. *DARQ* [17] went in a similar direction, but expanded upon the definition of data source capabilities as part of their manifests. Our investigation does not necessarily aim to replace these techniques:

[2]Vocabulary of Interlinked Datasets, http://rdfs.org/ns/void

in fact, it aims to develop an overarching framework where they can be either accommodated, or fed prepared queries in order to reduce their workload.

"Just-in-time" (JIT) is a term lifted from software compilers. It refers to compiling portions of software code to machine code as it gets executed, rather than altogether before execution. Uses of JIT have occasionally appeared in data integration, referring to the pruning of queries to be executed locally depending on the data requested [8], or to the dynamic selection of query operators based on which ones best support the data source [11]. Though both notions can be considered terminologically valid, the one we adopt is closer to the latter than to the former, yet extended in scope to include the translation of several types of query components, above and beyond operators and identifiers.

3 RECOMPILING QUERIES

Suppose we want to develop a virtual data integration system to support a Web API whose URIs follow a language that can be synthesised as follows:

```
{hostname}{/attribute|attribute.../value}+
        ?{attribute}{&attribute}+
```

For example, the URI

```
http://example.org/api/type/person/id|name/
        clint_eastwood?filmwork
```

reads "give me the filmwork of any person who either *is* Clint Eastwood (who is an actor and director) or *is named like* 'Clint Eastwood'".[3] The VDIS integrates data from *DBpedia*[4] and *LinkedMDB*,[5] two datasets of the benchmark used in the evaluation (cf. Sec. 5). An initial study of these datasets will provide an insight into what property chains and naming conventions exist.[6] Taking advantage of the above and any possible ontology alignments (which would tell us, for example, how to transform the `filmwork` property for DBpedia and LinkedMDB), one could translate the above to one SPARQL query[7] sent to both endpoints in order to *seek* a match:

```
SELECT DISTINCT ?filmwork WHERE {
  { dbr:Clint_Eastwood a foaf:Person
     ; ^(dbo:director|dbo:starring) ?filmwork
  } UNION {
    { ?x owl:sameAs dbr:Clint_Eastwood } UNION
    { ?x (movie:actor_name|movie:director_name) ?s FILTER
         (str(?s) = "Clint Eastwood")
    } . ?x foaf:made|^(movie:director|movie:actor) ?filmwork
  }}
```

However, such a query places unnecessary workload upon the SPARQL engines of both datasets, and LinkedMDB does not even fully support this syntax. Feeding this query to a federated SPARQL engine [18] is not guaranteed to work either: in fact, it has caused several error conditions when we experimented with it, due to either missing support from the federated engine, or refusal of the remote endpoints to process the optimisation/lookahead queries generated by the federated engines (see also Sec. 5). To make data retrieval and integration feasible, we could instead generate distinct queries for each endpoint; namely for DBpedia

```
SELECT DISTINCT ?filmwork ?eq WHERE {
  dbr:Clint_Eastwood a foaf:Person
    ; ^(dbo:director|dbo:starring) ?filmwork
  . ?filmwork owl:sameAs|^owl:sameAs ?eq }
```

and for LinkedMDB (whose URI patterns are very hard to exploit)

```
SELECT DISTINCT ?filmwork ?eq WHERE {
 { ?x owl:sameAs dbr:Clint_Eastwood } UNION {
   ?x movie:actor_name "Clint Eastwood" } UNION {
   ?x movie:director_name "Clint Eastwood" }
 } . {
   { ?x foaf:made ?filmwork } UNION { ?filmwork
        movie:director ?x }
   UNION { ?filmwork movie:actor ?x }
 } . ?filmwork owl:sameAs|^owl:sameAs ?eq
}
```

These queries are not intended to be exactly equivalent to the first one, in that they also project the `owl:sameAs` links to the `eq` variable, in order to facilitate actual data integration. More important is that the core semantics of the query is retained.

We want a VDIS that is able to directly produce the last two queries, with only partial knowledge of the entire schemas of each dataset. Normally a GAV mapping associates a query to each element of the global schema, such as `type`, `name` or `filmwork`, which may produce many mappings. We propose to refine the process by having fewer mappings produce elements of a query, which are used to assemble the final query. We realise this with what we call a just-in-time (JIT) framework.

3.1 Just-in-time framework

We start with an input query q, which is a conjunctive formula whose members are the attribute-value pairs (or disjunctions thereof) as in the example at the beginning of this section.

Definition 1. *Let $\mathbb{W}$ be the set of all the attribute-value pairs and Σ the alphabet of a language; a* ***microcompiler*** *is a function $\phi : \wp^{\mathbb{W}} \to \Sigma^*$ that transforms sets of attribute-value pairs into a sequence of symbols in that language.*

This language could be the target one or an intermediate one. Assuming the former, a function that generates the DBpedia URI for `dbr:Clint_Eastwood` from the {(`id`,`clint_eastwood`),(`name`, `clint_eastwood`)} pair is a simple microcompiler. Unlike GAV mappings, a microcompiler determines the association not of a query, but of a query element (a URI in this case) to one or more symbols of the global schema (`id` and `name`). Because the association can be to more than one symbols, we need a strategy for selecting which functions to execute, and how to assemble their outputs into a query over the source schema. This is what a JIT compiler will do. Let us now define another structure to drive the assembly.

Definition 2. *A* ***query skeleton****, or query template, t is a member of $(\Sigma \cup C)^*$, where C is an alphabet called set of* ***control symbols****.*

[3] We use "name" as a general term for a name or title of things, and "named like" means that the query is not sensitive to case, spaces instead of underscores etc.

[4] DBpedia, http://dbpedia.org

[5] Linked Movie Database, http://data.linkedmdb.org

[6] We know that URIs are generally opaque; however if a naming pattern is known for datasets such as DBpedia, there is no reason not to take advantage of it in heuristics.

[7] All prefixes are assumed to be the top results from the http://prefix.cc service.

Again in the above example, a possible query skeleton is the one used to generate the basic graph pattern (BGP) for `filmwork` in the DBpedia query.

```
<[id]> ^(dbo:director|dbo:starring) ?[ctrl_0x]?
```

In the above, `<[id]>` and `?[ctrl_0x]?` are control symbols: eventually, they will be replaced with members of Σ (URIs, variables, BGPs etc.). Note the similarity of the first one to the `id` attribute in the query: we will return to that later in Section 3.3.

Microcompilers and query skeletons are the constituents of a data source specification that a VDIS will use to generate the query for that particular data source, if it is selected by the source selection strategy. If $\mathcal{L}$ is the target language – in our example SPARQL – with alphabet Σ, then the JIT *compiler* itself can be summarised as the function $\Upsilon : \Phi \times \wp^{(\Sigma \cup C)^*} \times \wp^{\mathbb{W}} \rightarrow \mathcal{L}$ (Φ being the universe of microcompilers) that generates the query. A compiler can have many strategies: a bare-bone compiler simply replaces all the control symbols of a query skeleton with the output of microcompilers. A slightly more sophisticated compiler will add `owl:sameAs` BGPs to the query, like we did in our examples, and project onto those in the SELECT. Multiple calls to the compiler will concur to the generation of a *query plan* for a federated query engine to execute.

In general, the definition and complexity of microcompilers and query skeletons is up to the data integration manager, who knows the global schema and has some knowledge of the source schemas. Is it possible then to set up a VDIS by only knowing part of the source schemas and defining only a few mappings? How can the process be made iterative and sustainable, as a pay-as-you-go scheme that is peculiar to virtual data integration would require?

3.2 Compiler strategies

We now describe how to to put together microcompilers and query skeletons into a framework that keeps linked data integration manageable. The goal is to show how they can be dynamically selected depending on the query and the knowledge of the data sources' ability to handle it. We introduce the final notions here.

Definition 3. *A **manifest** is a pair $M = (F, T)$, where F is a set of microcompilers and T a set of query skeletons.*[8]

A strategy for keeping the configuration of a JIT query compiler under control is to designate a restricted set of attributes for input queries, whose possible values can, under reasonable conditions, be considered finite and enumerable. For example in an entity-centric approach, which is common for Linked Data, we propose to use entity types. In so doing, we distinguish two kinds of manifest:

- **entity type manifests** contain microcompilers F and query skeletons T associated to a specific type of entity, or group of types, as well as a mapping from attribute names to elements of F and T;
- similarly, **data source manifests** contain the same elements associated to a specific type of entity, or group thereof; however, they also include mappings to elements of F and T both from entity types and from attribute *names.*

[8]A manifest may also further specify the dataset, through e.g. a VoID description, ontologies or constraints, but we omit them as they are not used in this contribution.

Organising manifests in this way means including the actual *values* for an attribute called `type` in the global schema, thus allowing them to be assigned microcompilers. One reason for having type-based mappings in data source manifests is that, by knowing the taxonomical relationship between types, one can define inheritance rules in data source manifests, so they can share microcompilers or query skeletons for the same or similar types. Also, if there is such knowledge of a data source as to detect conventions on the way entity URIs are generated from their properties, this knowledge can be implemented in microcompilers specific to that data source.

Definition 4. *A query skeleton $t \in T$ is **satisfiable** for a set of microcompilers F and a language $\mathcal{L}$ if there exist $\phi_i \in F$ such that $\Upsilon(\{\phi_i\}, t, W) = \omega$, the attributes expected by ϕ_i and present in t are in W, and ω is a subsequence of some $q_{\mathcal{L}} \in \mathcal{L}$.*[9]

It follows that a set of query skeletons is satisfiable if their control symbols can be eliminated (reduced). How microcompilers and query skeletons are prioritised, and how this influences the selection of data sources to be queried, belongs to the strategy adopted by the compiler. Algorithm 1 illustrates our proposed strategy.

Algorithm 1: Data source selection and compilation strategy

```
Data: a query $q$; a target query language $\mathcal{L}$; a set of entity
      type manifests $M^T$; a set of data sources $S$, each with a
      manifest containing mappings
Result: a set $R$ of datasource-query pairs
extract attribute-value pairs $P_q = \{(a, v)\}$ and projection
attributes $A$ from $q$.
for each $j \in S$ with manifest $M_j^S = (F_j^S, T_j^S)$ and mapping $m$ do
    $F_j = \emptyset$                /* microcompilers for j */
    for $(a_i, v_i) \in P_q$ do
        if $a_i$ is an entity type attribute then
            if $\exists \phi \in F_j^S, m(v_i) = \phi$ and the attributes in the
            input set expected by $\phi$ are in $P_q \cup A$ then
                $F_j = F_j \cup \{\phi\}$ /* select microcompiler if
                executable */
            else
                look for manifest $M_{v_i}^T$ for type $v_i$ and try to
                select $\phi$ as above, using its spec. to cycle
                through supertypes of $v_i$ if necessary.
        else
            same as type attribute case, but look for $\phi$ so that
            $m(a_i) = \phi$ instead
    $T_j$ = the largest $\{t_s \in T_j^S\}$ that is satisfiable wrt. $F_j$ and $\mathcal{L}$
    $q_j^T = \Upsilon(F_j, T_j, P_q)$     /* compile and add resulting
    query to result set */
    if $q_j^T$ is not empty and is in $\mathcal{L}$ then $R = R \cup \{(j, q_j^T)\}$
```

The algorithm reads as follows: parse the input query into a set of attribute-value pairs[10] and projection attributes (those whose

[9]With a slight abuse of notation, we used the function Υ applied to a single query skeleton to generate, rather than the entire query, a sequence that could be part of it.

[10]Note that there can also be no value, in which case the clause is the existence of some value for that attribute.

values we want returned, e.g. `filmwork` in the example). If one of the attributes is e.g. `type`, look in each dataset manifest for microcompilers mapped to that specific type, if they can be executed by passing them all the attribute-value pairs and the names of the projection attributes; if there is none, fall back to the corresponding (data source-independent) entity type manifest and look for microcompilers there (or in its supertype manifests). Do the same for the *names* of the other attributes, only this time there is no falling back to type manifests. Then, inspect the data source manifest again for query skeletons that are satisfiable with these microcompilers. Finally, execute the compiler, which in turn will run the selected microcompilers, inject their output into the query skeleton and assemble the result. If we obtain a well-formed, non-empty query, add it to the set of queries to be executed. We are assuming here that actual query planning will be executed externally as it falls beyond the remit of this paper, which is about generating the queries.

If M^T is the set of entity type manifests, M^S is the set of data source manifests and A is the set of attribute names supported by the input query languages, the number of microcompilers and query skeletons to create and manage would be up to $(|M^T| + |A|) \cdot |M^S| + |M^T|$ in the worst case. Our experiments, however, have shown (cf. Sec. 5) that a JIT compiler working across several linked open datasets can operate with a number closer to $|A| \cdot |M^S| + |M^T|$, mainly because microcompilers for entity types (typically those that generate graph patterns on the `rdf:type` property) are easier to reuse across data sources.

3.3 Application to SPARQL and RDF

We close the description of the approach by describing how we put it in practice when integrating RDF data by querying via SPARQL.

Whilst the decision to balance between the number and complexity of microcompilers and the number and size of query skeletons is subjective to the data integration manager, we have attempted to concentrate as much target query content as possible into the microcompilers, in expectation that they would yield more query results with an acceptable overhead. In an effort to classify these functions according to the types of SPARQL statements (or parts or groups of statements that they return), we list here the ones we have been using the most:

Type 1. (**URIs**). They generate a single entity URI per data source, as in the earlier example of the function ϕ so that we can obtain $\phi_{\mathtt{id}}^{DBpedia}((\mathtt{id}, \mathtt{clint\_eastwood})) = \mathtt{dbr{:}Clint\_Eastwood}$. These have the advantage that they can exploit URI naming conventions and use heuristics to attempt a match, but can only work for a limited number of datasets and input queries.

Type 2. (**binding set assignments**). Depending on the SPARQL flavour, they generate VALUES clauses or UNIONs. For example, in order to list the places to eat in some data source s, a microcompiler in a type manifest can generate a query expansion so that $\phi^{\mathtt{place\_to\_eat}}(\mathtt{place})$ generates (in SPARQL 1.1):

```
VALUES (?t) {
  ( :Cafe ) ( :Diner ) ( :Restaurant )
} ?place a ?t
```

They are one of the most powerful and efficient ways to expand a query, if with limited support from SPARQL endpoints and federated engines.

Type 3. (**GPs**). They generate graph patterns, e.g. from the running example:

```
[ ^(dbo:director|dbo:starring) ?filmwork ]
```

or, if passed an `id` and a `type` as well, can also be used in a hybrid microcompiler of type 2 and 3 to generate the likes of:

```
VALUES(?gen_film) {
  ( dbr:Gran_Torino ) ( dbr:Gran_Torino_(film) )
} ?x ^(dbo:director|dbo:starring) ?gen_film . ?gen_film a
     dbo:Film
```

Type 4. (**filters**). They generate FILTERed values, e.g. to retrieve all the countries whose names end with 'Italy' (including the old Kingdom of Italy):

```
?x_name FILTER( STENDS(str(?x_name),"Italy") )
```

Here the handling of the `name` attribute would be handled by a query skeleton for Geonames and the type `country`, such as:

```
 ?[placeholder]? geonames:name|geonames:officialName
      <[name]>
   ; geonames:countryCode []
     # no matter what the code is, as long as there is one
```

Type 5. (**federation**). They generate SERVICE clauses and should only be used as a last resort, e.g. to retrieve an ID that cannot be computed programmatically.

4 IMPLEMENTATION

The work on query recompilation is part of *MK:Smart*, a project on Big Data for Smart Cities focussed on the city of Milton Keynes, UK. The project required a minimal Web API for integrated data access, which includes the custom input query language used in the examples in this paper and the evaluation (cf. Sec. 5). This API specification encodes conjunctive queries of type *star* (i.e. where the subject is shared among the members of the conjunction [24]) as URIs, e.g. `[host]/type/ward/name/walton_park` ("give me all the data on the entities of type Ward named *Walton Park*", which in this case matches one entity).[11] The recompilation logic described in this paper translates such an input query into several queries, mostly in SPARQL, to be sent to the appropriate data sources.

The data hub that uses this implementation for MK:Smart has now gone public,[12] and another instance is being used to support the work of the AFEL project for informal learning.[13] The reference implementation of JIT-backed data integration exists as an open source project, which includes the JIT logic, a custom experimental

[11]This project that supports this work actually uses a simplified version e.g. `[host]/ward/walton_park`. The choice of language was driven by the use cases of the project and should not be seen as a constraint or defining feature of the method.
[12]The MK:Smart DataHub, http://datahub.mksmart.org
[13]AFEL Data Platform (for project team only), http://data.afel-project.eu

VDIS and an HTTP API.[14] To minimise the implementation bias, we did not use our entire VDIS in this experiment, instead opting for the tried-and-true solution of FedX. The reference implementation is in Java, whereas the language currently supported for microcompiler functions is JavaScript. This language was chosen because of its familiarity among developers and the availability of interpreters in other languages (our Java implementation uses *Rhino*[15]), but this is by no means prescriptive: we are investigating other languages to support. The reference implementation uses *CouchDB*[16] and its Map-Reduce capabilities in order to atomically retrieve multiple microcompiler functions as needed. We are also working on a library of prepared microcompilers that handle a subset of entity types and property-value combinations for popular linked datasets.[17]

5 EVALUATION

Because data federation systems operate differently from VDIS (cf. Sec. 1), they work with little prior knowledge of the datasets. Data integration systems instead have mappings that are derived from the knowledge of the vocabularies, naming conventions etc. We want to measure the cost of putting this prior knowledge to use in a systematic way with a JIT compiler, which used a restricted set of query skeletons and microcompilers. We consider two aspects:

(1) Because the recompilation approach can be used to generate expanded queries, it is expected that in some cases executing them will return more results that executing the single original SPARQL query in a federation. We note those instances from a benchmark where that is indeed the case.
(2) We measure the overhead caused by recompiling the input query for each data source, and discuss whether it is a fair price to pay for generating (where that is the case) more results than with a federated query approach.

In order to abate uncontrolled variables and reduce the bias of factors and dimensions beyond the scope of what is being evaluated, an existing data federation system was used in lieu of the one we are developing (cf. Sec. 4). *FedX* was chosen because it is a well-known, flexible and programmable engine that federates SPARQL queries without explicitly defining the terms of the federation like SPARQL SERVICE does [14, 18, 22]. When receiving a query, FedX sends it to all the endpoints in the federation, after applying some ad-hoc optimisations to them for each endpoint. One drawback of this approach is that the input query should be generic enough to be satisfied by multiple endpoints, e.g. by using UNION and OPTIONAL clauses, which are generally costly due to how they are implemented with joins. FedX tries to optimise such joins by issuing further queries: while experimenting with the live endpoints, this has caused some of them to throw errors or ban us because of request throttling policies. We assumed, and proved afterwards, that a JIT engine could reduce the likelihood of these errors by preparing specific queries for a few endpoints at a time, and letting the federation engine of FedX handle them subsequently.

To benchmark our implementation we based ourselves upon *FedBench* [20], arguably the best-known extensible benchmark for federated SPARQL. We took the FedBench queries that could be translated to our simple custom star-shaped conjunctive query language (cf. Sec. 3 and 4) and translated them. For example, the FedBench cross-domain[18] query ***CD2*** became the query ***CD2***^{c}:

```
[hostname]/id/barack_obama?party&homepage
```

Some identifiers or property names were replaced so that we could get answers from more of the dataset versions online today. For instance, in query ***LD11*** we changed football team membership into musical band membership, and in ***LD10*** we used another DBpedia category. These customisations are allowed in FedBench. We ran exploratory queries for the dataset structures and encoded what we learned into mappings (21 microcompilers and 16 query skeletons in total): that was the configuration of our JIT framework. For each query, we pipelined the output of our JIT engine into FedX.

The evaluation was conducted under the following conditions:

- The experiments ran on a 2.2 GHz Intel Core i7 with 16 GB of 1600 MHz DDR3 RAM running a 64-bit Java Hotspot 1.8.0.
- Measurements were taken on 30 runs per query, discarding the time taken for parsing the input query and dropping the first few warm-up runs.
- We used the *live* SPARQL endpoints whenever possible, in order to assess the impact on the behaviour of external services with our queries (as in the case described earlier where using plain FedX alone got us banned or throttled). This explains why some results are different than older FedX benchmarks. If the endpoint was down or not responding appropriately (e.g. New York Times and LinkedMDB), we loaded locally the dumps downloaded from the FedBench site.
- The microcompiler functions we used were of linear complexity by construction, and did not attempt to query the data sources by themselves, unless otherwise noted. Also they did not contain hardcoded property values, with the possible exception of some common RDF types.

FedX only underwent marginal modifications because the version in the codebase threw optimisation errors upon using certain SPARQL 1.1 clauses that were not commonly used when FedX was developed. Examples are binding set assignments (VALUES clauses) and triple patterns with anonymous nodes, both largely adopted by the microcompilers of our experiments. These statements were simply passed through without FedX trying to optimise them.

5.1 Results and discussion

We recorded in Table 1 the cases where the expanded queries created by the JIT compiler and passed onto FedX generated more (still correct) results than the number *m* of those generated by the original FedBench query. Other than in special highlighted cases (see notes of table) the query expansion caused by JIT created *one rewritten query per source*, for each original query, regardless of whether multiple data sources reside on the same SPARQL endpoint. We omitted Life Science queries ***LS4,6,7*** because, when executed on the latest instance of the datasets, they returned no results to us, with or without JIT support. We also disregarded the SP2Bench

[14] ECAPI, https://github.com/mk-smart/entity-centric-api (source only).
[15] Rhino, https://developer.mozilla.org/en-US/docs/Mozilla/Projects/Rhino
[16] Apache CouchDB, http://couchdb.apache.org
[17] http://github.com/alexdma/jit-datasource-library, also with evaluation queries.

[18] http://fedbench.fluidops.net/resource/fbenchQuery:cross-domain_2

Query	VDI result set boost	Notes
	FedBench Cross-Domain	
CD1^c	$m * 1.387$	
CD2^c	52 new results	0 results with plain FedX
CD3^c	67 new results	† 0 results with plain FedX
CD4^c	$m * 4480.0$	‡
CD5^c	$m * 1.0$	no increment from JIT
	FedBench Life Sciences	
LS1^c	$m * 1.0$	query could not be expanded
LS2^c	$m * 1.0$	no increment from JIT
LS3^c	70981 results	‡ error with plain FedX
	FedBench Linked Data	
LD5^c	$m * 3.677$	
LD9^c	4 new results	0 results with plain FedX
LD10^c	$m * 17.0$	
LD11^c	$m * 1.65$	

Table 1: Increments in result set size using FedX + JIT, where *m* is the number of results returned by plain FedX. In notes: (†): in VDI case, the resulting compiled query contained one SERVICE clause (type 5). (‡): in VDI case, some microcompiler function had to perform a SPARQL query autonomously (e.g. to generate a VALUES clause).

Query	Time (ms) - Feder.		Time (ms) - VDI	
	plain FedX	*FedX+JIT*	*JIT overhead*	*Query TAT*
	FedBench Cross-Domain			
CD1^c	300 ± 50	420 ± 109	400 ± 20	800 ± 120
CD2^c	175 ± 5	475 ± 55	432 ± 9	1500 ± 123
CD3^c	158 ± 4	446 ± 76	408 ± 16	1067 ± 48
CD4^c	8835 ± 954	420 ± 100	787 ± 165	7480 ± 569
CD5^c	851 ± 319	519 ± 145	448 ± 31	548 ± 61
	FedBench Life Sciences			
LS1^c	421 ± 52	892 ± 43	query could not be expanded	
LS2^c	449 ± 113	420 ± 100	444 ± 61	370 ± 61
LS3^c	!ERROR	6653 ± 861	query could not be expanded	
	FedBench Linked Data			
LD5^c	795 ± 371	801 ± 78	486 ± 17	1028 ± 99
LD9^c	484 ± 166	407 ± 23	390 ± 39	318 ± 61
LD10^c	189 ± 36	440 ± 18	416 ± 17	658 ± 101
LD11^c	387 ± 67	861 ± 57	406 ± 20	762 ± 95

Table 2: Performance analysis comparing the times (in milliseconds) for executing data integration queries with FedX + JIT against running federated queries with FedX alone.

queries, since they are meant for execution on a single dataset each, and ***LD1-4*** to which only SWDF[19] answered.

Table 2 shows various costs associated with the results of Table 1. We measured the average cost of adding JIT compilation to generate an equivalent (not expanded) query sent to FedX (*FedX+JIT*), and the cost of employing more refined microcompilers and query skeletons to expand the queries and enable virtual data integration (VDI) capabilities. We divided these in the time taken to compile (*JIT overhead*) and the turnaround time of the resulting queries (*Query TAT*), i.e. the time between issuing the queries (once they have been parsed) and obtaining the results (before consuming them). We note that, in most cases where we tried to generate a single query for VDI (which had to include several UNION and OPTIONAL clauses), FedX threw an error trying to optimise it for the SPARQL endpoints in the federation.

In many of the cases analysed, we were able to use JIT compilation to obtain queries that expanded upon the original one, and at a manageable cost. For instance, starting from ***CD1***, the corresponding ***CD1***^{c} in our input language was recompiled into expanded queries that returned nearly 39% more results (i.e. SPARQL SELECT bindings) than the original, at the standard cost of under half a second for generating the queries and under a second for executing them *in sequence* (parallelisation will be investigated in the future). A similar reasoning applies to ***LD5,10-11***. These have in common that they were generated using hybrid microcompilers of types 2 and 3. JIT compilation also proved effective for queries that involved literal matching and could employ microcompilers of types 3 and 4 (***CD4, LD9***), retrieving more results in both cases at comparable overall execution times. Queries ***CD2-3*** required several URI substitutions to be effective, as the originals yielded zero results (and were comprehensibly quick in doing so).

In the case of Life Sciences queries, which use four datasets of which three are radically distinct in scope and the fourth is DBpedia, we had very little room for manoeuvre and only once were we able to generate an expanded query, which however yielded no additional results. However, we report that JIT re-enabled one query (***LS3***), if at the price of performing a join by itself, which was causing FedX to abort during optimisation.

We observed that the overhead of recompiling the input query into expanded queries is approximately constant with respect to the size of the results, amounting to < 0.5 seconds in all cases except where the only way to perform query expansion was to issue further lookahead queries (***CD4***). We consider this an acceptable price for queries that return more results (or *some* results where the originals did not or caused an error). Most increases in query turnaround time are justified by the increase in result size. When the expanded queries could be generated, but yielded no additional results, the performance is only slightly above, if not similar to, the plain federated approach. In other words, query recompilation seems to be mostly efficient where it is also effective. We finally observe from the *plain FedX* vs. *FedX+JIT* comparison, that compilation is not a viable replacement for the optimisations of pure federated engines, i.e. when it has to generate single queries across all datasets. Though this was to be expected, we will still look into ways for a JIT compiler to detect such cases quicker and delegate query optimisation to federated engines.

6 CONCLUSION AND FUTURE WORK

We have illustrated an approach for dynamically generating target queries in virtual data integration. It is possible in this way to limit the queries issued per data request and incorporate query expansion, entity linking and more. This was demonstrated in a

[19]Semantic Web Dog Food, http://data.semanticweb.org

Linked Data scenario, with common federation and benchmarking frameworks as our testbed. The resulting framework exploits the query optimisations of a federated system, turning it into a VDIS.

The approach was validated with star-shaped input queries (which may however be compiled into more complex and highly selective expanded queries), as they constitute a significant subset of queries. We have there verified that the application of just-in-time recompilation is mostly efficient when it is also effective in delivering more results than standard federated queries. We reasonably assume these particular results will find some validity when applying the approach to chain-shaped (i.e. with property paths) and hybrid input queries, which we intend to investigate next.

Another aspect of external validity to investigate will be other target languages than SPARQL, as well as templating languages. So far we have restricted the investigation on query recompilation to suit the business needs of our projects, which also contemplate, but do not yet use, other target languages. For the purpose of the experiment in this paper, we have kept the components required to generate the target queries to a manageable level: it is important to demonstrate that in actual data-intensive scenarios the combinatorial explosion of these components can still be avoided, and we will use the Smart Cities use cases of the *MK:Smart* project to that end.

We intend to study the potential and limitations that the compiler paradigm may have in data integration. We are also investigating the ability to define cascaded mappings that are compiled at query time, enabling those mappings to refer to data from other data sources, without requiring knowledge of their content, structure or even their existence. Finally, we are also working on automatic recommendation of SPARQL queries in order to streamline the generation of query skeletons when new data sources are added.

Acknowledgements. This work was supported by the projects MK:Smart (http://mksmart.org, OU Ref. HGCK B4466) and AFEL (http://afel-project.eu, EU H2020).

REFERENCES

[1] I. Ali and H. H. Ghenniwa. Ontology-driven mediated data integration in open environment. In J. Filipe, J. L. G. Dietz, and D. Aveiro, editors, *KEOD 2014 - Proceedings of the International Conference on Knowledge Engineering and Ontology Development, Rome, Italy, 21-24 October, 2014*, pages 230–239. SciTePress, 2014.

[2] A. Bakhtouchi, L. Bellatreche, S. Jean, and Y. A. Ameur. Ontologies as a solution for simultaneously integrating and reconciliating data sources. In C. Rolland, J. Castro, and O. Pastor, editors, *Sixth International Conference on Research Challenges in Information Science, RCIS 2012, Valencia, Spain, May 16-18 2012*, pages 1–12. IEEE, 2012.

[3] L. E. Bertossi and L. Bravo. Consistent query answers in virtual data integration systems. In L. E. Bertossi, A. Hunter, and T. Schaub, editors, *Inconsistency Tolerance [result from a Dagstuhl seminar]*, volume 3300 of *Lecture Notes in Computer Science*, pages 42–83. Springer, 2005.

[4] A. Calì. Reasoning in data integration systems: Why LAV and GAV are siblings. In N. Zhong, Z. W. Ras, S. Tsumoto, and E. Suzuki, editors, *Foundations of Intelligent Systems, 14th International Symposium, ISMIS 2003, Maebashi City, Japan, October 28-31, 2003, Proceedings*, volume 2871 of *Lecture Notes in Computer Science*, pages 562–571. Springer, 2003.

[5] D. Calvanese, G. De Giacomo, D. Lembo, M. Lenzerini, R. Rosati, and M. Ruzzi. Using OWL in data integration. In R. D. Virgilio, F. Giunchiglia, and L. Tanca, editors, *Semantic Web Information Management - A Model-Based Perspective*, pages 397–424. Springer, 2009.

[6] M. J. Franklin, A. Y. Halevy, and D. Maier. From databases to dataspaces: a new abstraction for information management. *SIGMOD Record*, 34(4):27–33, 2005.

[7] O. Görlitz and S. Staab. SPLENDID: SPARQL endpoint federation exploiting VOID descriptions. In O. Hartig, A. Harth, and J. Sequeda, editors, *Proceedings of the Second International Workshop on Consuming Linked Data (COLD2011), Bonn, Germany, October 23, 2011*, volume 782 of *CEUR Workshop Proceedings*. CEUR-WS.org, 2011.

[8] M. Hentschel, L. M. Haas, and R. J. Miller. Just-in-time data integration in action. *PVLDB*, 3(2):1621–1624, 2010.

[9] B. Howe, D. Maier, N. Rayner, and J. Rucker. Quarrying dataspaces: Schemaless profiling of unfamiliar information sources. In *Proceedings of the 24th International Conference on Data Engineering Workshops, ICDE 2008, April 7-12, 2008, Cancún, México*, pages 270–277. IEEE Computer Society, 2008.

[10] Informatica. The Vibe virtual data machine. White paper, Informatica, 2013.

[11] M. Karpathiotakis, I. Alagiannis, T. Heinis, M. Branco, and A. Ailamaki. Just-in-time data virtualization: Lightweight data management with ViDa. In *CIDR 2015, Seventh Biennial Conference on Innovative Data Systems Research, Asilomar, CA, USA, January 4-7, 2015, Online Proceedings*. www.cidrdb.org, 2015.

[12] X. Li, Z. Niu, C. Zhang, and X. Wang. LAW: Link-AWare source selection for virtually integrating linked data. In S. Cheng and M. Day, editors, *Technologies and Applications of Artificial Intelligence, 19th International Conference, TAAI*, volume 8916 of *Lecture Notes in Computer Science*, pages 239–248. Springer, 2014.

[13] Z. Majkic. *Big Data Integration Theory - Theory and Methods of Database Mappings, Programming Languages, and Semantics*. Texts in Computer Science. Springer, 2014.

[14] G. Montoya, M. Vidal, Ó. Corcho, E. Ruckhaus, and C. B. Aranda. Benchmarking federated SPARQL query engines: Are existing testbeds enough? In P. Cudré-Mauroux, J. Heflin, E. Sirin, T. Tudorache, J. Euzenat, M. Hauswirth, J. X. Parreira, J. Hendler, G. Schreiber, A. Bernstein, and E. Blomqvist, editors, *The Semantic Web - ISWC 2012 - 11th International Semantic Web Conference, Boston, MA, USA, November 11-15, 2012, Proceedings, Part II*, volume 7649 of *Lecture Notes in Computer Science*, pages 313–324. Springer, 2012.

[15] N. W. Paton, K. Christodoulou, A. A. A. Fernandes, B. Parsia, and C. Hedeler. Pay-as-you-go data integration for linked data: opportunities, challenges and architectures. In R. D. Virgilio, F. Giunchiglia, and L. Tanca, editors, *Proceedings of the 4th International Workshop on Semantic Web Information Management, SWIM 2012, Scottsdale, AZ, USA, May 20, 2012*, page 3. ACM, 2012.

[16] V. M. Ponte Vidal, J. A. Fernandes de Macêdo, J. C. Pinheiro, M. A. Casanova, and F. Porto. Query processing in a mediator based framework for linked data integration. *IJBDCN*, 7(2):29–47, 2011.

[17] B. Quilitz and U. Leser. Querying distributed RDF data sources with SPARQL. In S. Bechhofer, M. Hauswirth, J. Hoffmann, and M. Koubarakis, editors, *The Semantic Web: Research and Applications, 5th European Semantic Web Conference, ESWC 2008, Tenerife, Canary Islands, Spain, June 1-5, 2008, Proceedings*, volume 5021 of *Lecture Notes in Computer Science*, pages 524–538. Springer, 2008.

[18] M. Saleem, Y. Khan, A. Hasnain, I. Ermilov, and A. Ngonga. Fine-grained evaluation of SPARQL endpoint federation systems. *Sem. Web J.*, 7(5):493–518, 2016.

[19] S. Schenk and S. Staab. Networked graphs: a declarative mechanism for SPARQL rules, SPARQL views and RDF data integration on the web. In J. Huai, R. Chen, H. Hon, Y. Liu, W. Ma, A. Tomkins, and X. Zhang, editors, *Proceedings of the 17th International Conference on World Wide Web, WWW 2008, Beijing, China, April 21-25, 2008*, pages 585–594. ACM, 2008.

[20] M. Schmidt, O. Görlitz, P. Haase, G. Ladwig, A. Schwarte, and T. Tran. FedBench: A benchmark suite for federated semantic data query processing. In L. Aroyo, C. Welty, H. Alani, J. Taylor, A. Bernstein, L. Kagal, N. F. Noy, and E. Blomqvist, editors, *The Semantic Web - ISWC 2011 - 10th International Semantic Web Conference*, volume 7031 of *Lecture Notes in Computer Science*, pages 585–600. Springer, 2011.

[21] D. Schober, R. Choquet, K. Depraetere, F. Enders, P. Daumke, M. Jaulent, D. Teodoro, E. Pasche, C. Lovis, and M. Boeker. Debugit: Ontology-mediated layered data integration for real-time antibiotics resistance surveillance. In A. Paschke, A. Burger, P. Romano, M. S. Marshall, and A. Splendiani, editors, *7th International Workshop on Semantic Web Applications and Tools for Life Sciences*, volume 1320 of *CEUR Workshop Proceedings*. CEUR-WS.org, 2014.

[22] A. Schwarte, P. Haase, K. Hose, R. Schenkel, and M. Schmidt. FedX: A federation layer for distributed query processing on linked open data. In G. Antoniou, M. Grobelnik, E. P. B. Simperl, B. Parsia, D. Plexousakis, P. D. Leenheer, and J. Z. Pan, editors, *The Semantic Web: Research and Applications - 8th Extended Semantic Web Conference, ESWC 2011*, volume 6644 of *Lecture Notes in Computer Science*, pages 481–486. Springer, 2011.

[23] A. P. Sheth and J. A. Larson. Federated database systems for managing distributed, heterogeneous, and autonomous databases. *ACM Comput. Surv.*, 22(3):183–236, 1990.

[24] M. Vidal, E. Ruckhaus, T. Lampo, A. Martínez, J. Sierra, and A. Polleres. Efficiently joining group patterns in SPARQL queries. In L. Aroyo, G. Antoniou, E. Hyvönen, A. ten Teije, H. Stuckenschmidt, L. Cabral, and T. Tudorache, editors, *7th Extended Semantic Web Conference, ESWC 2010*, pages 228–242, 2010.

[25] E. Zaidi, M. A. Beyer, and S. Vashisth. Market guide for data virtualization. Market research report, Gartner, Inc., July 2016.