The Virtual Knowledge Graph System Ontop

Guohui Xiao[1,2][0000−0002−5115−4769], Davide Lanti[1][0000−0003−1097−2965], Roman Kontchakov[3][0000−0002−9349−9159], Sarah Komla-Ebri[2][0000−0002−6559−2583], Elem Güzel-Kalaycı[4][0000−0001−9916−8062], Linfang Ding[1][0000−0002−3707−5845], Julien Corman[1][0000−0003−4386−5218], Benjamin Cogrel[2][0000−0002−7566−4077], Diego Calvanese[1,2,5][0000−0001−5174−9693], and Elena Botoeva[6][0000−0001−5881−0258]

[1] Free University of Bozen-Bolzano, Italy *lastname*@inf.unibz.it
[2] Ontopic s.r.l., Bolzano, Italy *firstname.lastname*@ontopic.biz
[3] Birkbeck, University of London, UK roman@dcs.bbk.ac.uk
[4] Virtual Vehicle Research GmbH, Graz, Austria elem.guezelkalayci@v2c2.at
[5] Umeå University, Sweden diego.calvanese@umu.se
[6] Imperial College London, UK e.botoeva@imperial.ac.uk

Abstract. *Ontop* is a popular open-source virtual knowledge graph system that can expose heterogeneous data sources as a unified knowledge graph. *Ontop* has been widely used in a variety of research and industrial projects. In this paper, we describe the challenges, design choices, new features of the latest release of *Ontop* v4, summarizing the development efforts of the last 4 years.

1 Introduction

The Virtual Knowledge Graph (VKG) approach, also known in the literature as Ontology-Based Data Access (OBDA) [16,23], has become a popular paradigm for accessing and integrating data sources [24]. In such approach, the data sources, which are normally relational databases, are *virtualized* through a mapping and an ontology, and presented as a unified *knowledge graph*, which can be queried by end-users using a vocabulary they are familiar with. At query time, a VKG system translates user queries over the ontology to SQL queries over the database system. This approach frees end-users from the low-level details of data organization, so that they can concentrate on their high-level tasks. As it is gaining more and more importance, this paradigm has been implemented in several systems [3,4,18,21], and adopted in a large range of use cases. Here, we present the latest major release, *Ontop* v4, of a popular VKG system.

The development of *Ontop* has been a great adventure spanning the past decade. Developing such a system is highly non-trivial. It requires both a theoretical investigation of the semantics, and strong engineering efforts to implement all the required features. *Ontop* started in 2009, only one year after the first version of the SPARQL specification had been standardized, and OWL 2 QL [14] and R2RML [9] appeared 3 years later in 2012. At that time, the VKG research focused on union of conjunctive queries (UCQs) as a query language. With this target, the v1 series of *Ontop* was using Datalog as the core for data representation [20], since it was fitting well a setting based on UCQs. The development of *Ontop* was boosted during the EU FP7 project Optique

(2013–2016). During the project, the compliance with all the relevant W3C recommendations became a priority, and significant progress has been made. The last release of *Ontop* v1 was v1.18 in 2016, which is the result of 4.6K git commits. A full description of *Ontop* v1 is given in [3], which has served as the canonical citation for *Ontop* so far.

A natural requirement that emerged during the Optique project were aggregates introduced in SPARQL 1.1 [12]. The *Ontop* development team spent a major effort, internally called *Ontop* v2, in implementing this query language feature. However, it became more and more clear that the Datalog-based data representation was not well suited for this implementation. Some prototypes of *Ontop* v2 were used in the Optique project for internal purposes, but never reached the level of a public release. We explain this background and the corresponding challenges in Section 2.

To address the challenges posed by aggregation and others that had emerged in the meantime, we started to investigate an alternative core data structure. The outcome has been what we call *intermediate query* (IQ), an algebra-based data structure that unifies both SPARQL and relational algebra. Using IQ, we have rewritten a large fragment of the code base of *Ontop*. After two beta releases in 2017 and 2018, we have released the stable version of *Ontop* v3 in 2019, which contains additional 4.5K commits with respect to *Ontop* v1. After *Ontop* v3, the development focus was to improve compliance and add several major features. In particular, aggregates are supported since *Ontop* v4-beta-1, released late 2019. We have finalized *Ontop* v4 and released it in July 2020, with additional 2.3K git commits. We discuss the design of *Ontop* v4 and highlight some benefits of IQ that VKG practitioners should be aware of in Section 3.

Ontop v4 has greatly improved its compliance with relevant W3C recommendations and provides good performance in query answering. It supports almost all the features of SPARQL 1.1, R2RML, OWL 2 QL and SPARQL entailment regime, and the SPARQL 1.1 HTTP Protocol. Two recent independent evaluations [7,15] of VKG systems have confirmed the robust performance of *Ontop*. When considering all the perspectives, like usability, completeness, and soundness, *Ontop* clearly distinguishes itself among the open source systems. We describe evaluations of *Ontop* in Section 4.

Ontop v4 is the result of an active developer community. The number of git commits now sums up to 11.4K. It has been downloaded more than 30K times from Sourceforge. In addition to the research groups, *Ontop* is also backed by a commercial company, Ontopic s.r.l., born in April 2019. *Ontop* has been adopted in many academic and industrial projects [24]. We discuss the community effort and the adoption of *Ontop* in Section 5.

2 Background and Challenges

A Virtual Knowledge Graph (VKG) system provides access to data (stored, for example, in a relational database) through an *ontology*. The purpose of the ontology is to define a vocabulary (classes and properties), which is convenient and familiar to the user, and to extend the data with background knowledge (e.g., subclass and subproperty axioms, property domain and range axioms, or disjointness between classes). The terms of the ontology vocabulary are connected to data sources by means of a *mapping*, which can be thought of as a collection of database queries that are used to construct class and property assertions of the ontology (RDF dataset). Therefore, a VKG system has the following components: (*a*) queries that describe user information needs, (*b*) an ontology with classes and properties, (*c*) a mapping, and (*d*) a collection of data sources. W3C published recommendations for languages for components *(a)–(c)*: SPARQL, OWL 2 QL, and R2RML, respectively; and SQL is the language for relational DBMSs.

A distinguishing feature of VKG systems is that they retrieve data from data sources only when it is required for a particular user query, rather than extracting all the data and materializing it internally; in other words, the Knowledge Graph (KG) remains *virtual* rather than materialized. An advantage of this approach is that VKG systems expose the actual up-to-date information. This is achieved by delegating query processing to data sources (notably, relational DBMSs): user queries are translated into queries to data sources (whilst taking account of the ontology background knowledge). And, as it has been evident from the early days [4,19,21,22], performance of VKG systems critically depends on the sophisticated query optimization techniques they implement.

Ontop v1. In early VKG systems, the focus was on answering conjunctive queries (CQs), that is, conjunctions of unary and binary atoms (for class and property assertions respectively). As for the ontology language, OWL 2 QL was identified [5,2] as an (almost) maximal fragment of OWL that can be handled by VKG systems (without materializing all assertions that can be derived from the ontology). In this setting, a *query rewriting* algorithm compiles a CQ and an OWL 2 QL ontology into a union of CQs, which, when evaluated over the data sources, has the same answers as the CQ mediated by the OWL 2 QL ontology. Such algorithms lend themselves naturally to an implementation based on non-recursive Datalog: a CQ can be viewed as a clause, and the query rewriting algorithm transforms each CQ (a clause) into a union of CQs (a set of clauses). Next, in the result of rewriting, query atoms can be replaced by their 'definitions' from the mapping. This step, called *unfolding*, can also be naturally represented in the Datalog framework: it corresponds to partial evaluation of non-recursive Datalog programs, provided that the database queries are SELECT-PROJECT-JOIN (SPJ) [16]. So, Datalog was the core data structure in *Ontop* v1 [20], which translated CQs mediated by OWL 2 QL ontologies into SQL queries. The success of *Ontop* v1 heavily relied on the semantic query optimization (SQO) techniques [6] for simplifying non-recursive Datalog programs. One of the most important lessons learnt in that implementation is that rewriting and unfolding, even though they are separate steps from a theoretical point of view, should be considered together in practice: a mapping can be combined with the subclass and subproperty relations of the ontology, and the resulting *saturated mapping* (or *T-mapping*) can be constructed and optimized before any query is processed, thus taking advantage of performing the expensive SQO only once [19].

Ontop Evolution: from Datalog to Algebra As *Ontop* moved towards supporting the W3C recommendations for SPARQL and R2RML, new challenges emerged.

- In SPARQL triple patterns, variables can occur in positions of class and property names, which means that there are effectively only two underlying 'predicates': `triple` for triples in the RDF dataset default graph, and `quad` for named graphs.
- More importantly, SPARQL is based on a rich algebra, which goes beyond expressivity of CQs. Non-monotonic features like OPTIONAL and MINUS, and cardinality-sensitive query modifiers (DISTINCT) and aggregation (GROUP BY with functions such as SUM, AVG, COUNT) are difficult to model even in extensions of Datalog.

– Even without SPARQL aggregation, cardinalities have to be treated carefully: the SQL queries in a mapping produce *bags* (multisets) of tuples, but their induced RDF graphs contain no duplicates and thus are *sets* of triples; however, when a SPARQL query is evaluated, it results in a bag of solutions mappings.

These challenges turned out to be difficult to tackle in the Datalog setting. For example, one has to use three clauses and negation to model OPTIONAL, see e.g., [17,1]. Moreover, using multiple clauses for nested OPTIONALs can result in an exponentially large SQL query, if the related clauses are treated independently. On the other hand, such a group of clauses could and ideally *should* be re-assembled into a single LEFT JOIN when translating into SQL, so that the DBMS can take advantage of the structure [25]. Curiously, the challenge also offers a solution because most SPARQL constructs have natural counterparts in SQL: for instance, OPTIONAL corresponds to LEFT JOIN, GROUP BY to GROUP BY, and so on. Also, both SPARQL and SQL have bag semantics and use 3-valued logic for boolean expressions.

As a consequence of the above observations, when redesigning *Ontop*, we replaced Datalog by a relational-algebra-type representation, discussed in Section 3.

SPARQL vs SQL Despite the apparent similarities between SPARQL and SQL, the two languages have significant differences relevant to any VKG system implementation.

Typing Systems. SQL is *statically typed* in the sense that all values in a given relation column (both in the database and in the result of a query) have the same datatype. In contrast, SPARQL is *dynamically typed*: a variable can have values of different datatypes in different solution mappings. Also, SQL queries with values of unexpected datatypes in certain contexts (e.g., a string as an argument for '+') are simply deemed incorrect. In contrast, SPARQL treats such *type errors* as legitimate and handles them similarly to `NULL`s in SQL. For example, the basic graph pattern `?s ?p ?o FILTER (?o < 4)` retrieves all triples with a numerical object whose value is below 4 (but *ignores* all triples with strings or IRIs, for example). Also, the output datatype of a SPARQL function depends on the types or language tags of its arguments (e.g., if both arguments of '+' are `xsd:integer`, then so is the output, and if both arguments are `xsd:decimal`, then so is the output). In particular, to determine the output datatype of an aggregate function in SPARQL, one has to look at the datatypes of values in the group, which can vary from one group to another.

Order. SPARQL defines a fixed order on IRIs, blank nodes, unbound values, and literals. For multi-typed expressions, this general order needs to be combined with the orders defined for datatypes. In SQL, the situation is significantly simpler due to its static typing: apart from choosing the required order modifier for the datatype, one only needs to specify whether `NULL`s come first or last.

Implicit Joining Conditions. SPARQL uses the notion of solution mapping compatibility to define the semantics of the JOIN and OPTIONAL operators: two solution mappings are *compatible* if both map each shared variable to the same RDF term (`sameTerm`), that is, the two terms have the same type (including the language tag for literals) and the same lexical value. The `sameTerm` predicate is also used for AGGREGATEJOIN. In contrast, equalities in SQL are satisfied when their arguments are *equivalent*, but not necessarily of the same datatype (e.g., `1` in columns of type `INTEGER` and `DECIMAL`), and may even have different lexical values (e.g., timestamps with different timezones). SPARQL has a similar equality, denoted by '`=`', which can occur in FILTER and BIND.

SQL Dialects. Unlike SPARQL with its standard syntax and semantics, SQL is more varied as DBMS vendors do not strictly follow the ANSI/ISO standard. Instead, many use specific datatypes and functions and follow different conventions, for example, for column and table identifiers and query modifiers; even a common function `CONCAT` can behave differently: `NULL`-rejecting in MySQL, but not in PostgreSQL and Oracle. Support for the particular SQL dialect is thus essential for transforming SPARQL into SQL.

3 Ontop v4: New Design

We now explain how we address the challenges in *Ontop* v4. In Section 3.1, we describe a variant of relational algebra for representing queries and mappings. In Section 3.2, we concentrate on translating SPARQL functions into SQL. We discuss query optimization in Section 3.3, and post-processing and dealing with SQL dialects in Section 3.4.

3.1 Intermediate Query

Ontop v4 uses a variant of relational algebra tailored to encode SPARQL queries along the lines of the language described in [25]. The language, called *Intermediate Query*, or IQ, is a uniform representation both for user SPARQL queries and for SQL queries from the mapping. When the query transformation (rewriting and unfolding) is complete, the IQ expression is converted into SQL, and executed by the underlying relational DBMS.

In SPARQL, an *RDF dataset* consists of a default RDF graph (a set of triples of the form *s*-*p*-*o*) and a collection of named graphs (sets of quadruples *s*-*p*-*o*-*g*, where *g* is a graph name). In accordance with this, a ternary relation `triple` and a quaternary relation `quad` model RDF datasets in IQ. We use atomic expressions of the form

$$\texttt{triple}(s, p, o) \qquad \text{and} \qquad \texttt{quad}(s, p, o, g),$$

where s, p, o, and g are either constants or variables—in relational algebra such expressions would need to be built using combinations of SELECT (σ, to deal with constants and matching variables in different positions) and PROJECT (π, for variable names), see, e.g., [8]: for example, triple pattern `:ex :p` $?x$ would normally be encoded as $\pi_{x/o}\sigma_{s=\texttt{":ex"},p=\texttt{":p"}}\texttt{triple}$, where s, p, o are the attributes of `triple`. We chose a more concise representation, which is convenient for encoding SPARQL triple patterns.

We illustrate the other input of the SPARQL to SQL transformation (via IQ) using the following mapping in a simplified syntax:

$$T_1(x, y) \rightsquigarrow \texttt{:b\{}x\texttt{\}}\ \texttt{:p}\ y,\ T_2(x, y) \rightsquigarrow \texttt{:b\{}x\texttt{\}}\ \texttt{:p}\ y,$$
$$T_3(x, y) \rightsquigarrow \texttt{:b\{}x\texttt{\}}\ \texttt{:p}\ y,\ T_4(x, y) \rightsquigarrow \texttt{:b\{}x\texttt{\}}\ \texttt{:q}\ y,$$

where the triples on the right-hand side of $\rightsquigarrow$ represent `subjectMaps` together with `predicateObjectMaps` for properties `:p` and `:q`. In database tables T_1, T_2, T_3, and T_4, the first attribute is the primary key of type `TEXT`, and the second attribute is non-nullable and of type `INTEGER`, `DECIMAL`, `TEXT`, and `INTEGER`, respectively. When we translate the mapping into IQ, the SQL queries are turned into atomic expressions $T_1(x, y)$, $T_2(x, y)$,

$T_3(x, y)$, and $T_4(x, y)$, respectively, where the use of variables again indicates the π operation of relational algebra. The translation of the right-hand side is more elaborate.

Remark 1. IRIs, blank nodes, and literals can be constructed in R2RML using *templates*, where placeholders are replaced by values from the data source. Whether a template function is *injective* (yields different values for different values of parameters) depends on the shape of the template. For IRI templates, one would normally use *safe separators* [9] to ensure injectivity of the function. For literals, however, if a template contains more than one placeholder, then the template function may be non-injective. On the other hand, if we construct literal values of type `xsd:date` from three separate database `INTEGER` attributes (for day, month, and year), then the template function is injective because the separator of the three components, `-`, is 'safe' for numerical values.

Non-constant RDF terms are built in IQ using the binary function `rdf` with a `TEXT` lexical value and the term type as its arguments. In the example above, all triple subjects are IRIs of the same template, and the lexical value is constructed using template function $\texttt{:b\{\}}(x)$, which produces, e.g., the IRI `:b1` when $x = 1$. The triple objects are literals: the `INTEGER` attribute in T_1 and T_4 is mapped to `xsd:integer`, the `DECIMAL` in T_2 is mapped to `xsd:decimal`, and the `TEXT` in T_3 to `xsd:string`. Database values need to be cast into `TEXT` before use as lexical values, which is done by unary functions i2t and d2t for `INTEGER` and `DECIMAL`, respectively. The resulting IQ representation of the mapping assertions is then as follows:

$$\begin{aligned}
T_1(x, y) &\rightsquigarrow \texttt{triple}(\texttt{rdf}(\texttt{:b\{\}}(x), \text{IRI}),\ \texttt{:p},\ \texttt{rdf}(\text{i2t}(y), \texttt{xsd:integer})),\\
T_2(x, y) &\rightsquigarrow \texttt{triple}(\texttt{rdf}(\texttt{:b\{\}}(x), \text{IRI}),\ \texttt{:p},\ \texttt{rdf}(\text{d2t}(y), \texttt{xsd:decimal})),\\
T_3(x, y) &\rightsquigarrow \texttt{triple}(\texttt{rdf}(\texttt{:b\{\}}(x), \text{IRI}),\ \texttt{:p},\ \texttt{rdf}(y, \texttt{xsd:string})),\\
T_4(x, y) &\rightsquigarrow \texttt{triple}(\texttt{rdf}(\texttt{:b\{\}}(x), \text{IRI}),\ \texttt{:q},\ \texttt{rdf}(\text{i2t}(y), \texttt{xsd:integer})).
\end{aligned}$$

To illustrate how we deal with multi-typed functions in SPARQL, we now consider the following query (in the context of the RDF dataset discussed above):

```
SELECT ?s WHERE { ?x :p ?n .  ?x :q ?m .
                  BIND ((?n + ?m) AS ?s)  FILTER (bound(?s)) }
```

It involves an arithmetic sum over two variables, one of which, `?n`, is multi-typed: it can be an `xsd:integer`, `xsd:decimal`, or `xsd:string`. The translation of the SPARQL query into IQ requires the use of most of the algebra operations, which are defined next.

A *term* is a variable, a constant (including `NULL`), or a functional term constructed from variables and constants using SPARQL function symbols such as `numeric-add`, SQL function symbols such as `+`, and our auxiliary function symbols such as `IF`, etc. (`IF` is ternary and such that $\texttt{IF}(\texttt{true}, x, y) = x$ and $\texttt{IF}(\texttt{false}, x, y) = y$). We treat predicates such as `=` and `sameTerm` as function symbols of *boolean* type; boolean connectives $\neg$, $\wedge$, and $\vee$ are also boolean function symbols. Boolean terms are interpreted using the *3-valued logic*, where `NULL` is used for the 'unknown value.' An *aggregate term* is an expression of the form $agg(\tau)$, where *agg* is a SPARQL or SQL aggregate function symbol (e.g., `SPARQL_Sum` or `SUM`) and τ a term. A *substitution* is an expression of the form $x_1/\eta_1, \dots, x_n/\eta_n$, where each x_i is a variable and each η_i either a term (for PROJ) or an aggregate term (for AGG). Then, IQs are defined by the following grammar:

$$\phi \ := \ P(\mathbf{t}) \mid \textsc{Proj}^{\mathbf{x}}_{\tau}\ \phi \mid \textsc{Agg}^{\mathbf{x}}_{\tau}\ \phi \mid \textsc{Distinct}\ \phi \mid \textsc{OrderBy}_{\mathbf{x}}\ \phi \mid \textsc{Slice}_{i,j}\ \phi \mid$$

$$\textsc{Filter}_\beta\ \phi \mid \textsc{Join}_\beta(\phi_1,\dots,\phi_k) \mid \textsc{LeftJoin}_\beta(\phi_1,\phi_2) \mid \textsc{Union}(\phi_1,\dots,\phi_k),$$

where P is a relation name (`triple`, `quad`, or a database table name), $\mathbf{t}$ a tuple of terms, $\mathbf{x}$ a tuple of variables, τ a substitution, $i, j \in \mathbb{N} \cup \{0, +\infty\}$ are values for the offset and limit, respectively, and β is a boolean term. When presenting our examples, we often omit brackets and use indentation instead. The algebraic operators above operate on bags of tuples, which can be thought of as *total* functions from sets of variables to values, in contrast to *partial* functions in SPARQL (such definitions are natural from the SPARQL-to-SQL translation point of view; see [25] for a discussion). Also, JOIN and LEFTJOIN are similar to `NATURAL (LEFT) JOIN` in SQL, in the sense that the tuples are joined (compatible) if their shared variables have the same values. All the algebraic operators are interpreted using the bag semantics, in particular, UNION preserves duplicates (similarly to `UNION ALL` in SQL).

In our running example, the SPARQL query is translated into the following IQ:

$$\begin{array}{l}\textsc{Proj}^{?s}_{?s/\texttt{numeric-add}(?n,?m)}\\ \quad\textsc{Join}_{\neg\mathit{isNull}(\texttt{numeric-add}(?n,?m))}(\texttt{triple}(?x,\ \texttt{:p},\ ?n),\ \texttt{triple}(?x,\ \texttt{:q},\ ?m)),\end{array}$$

where the `bound` filter is replaced by $\neg\mathit{isNull}()$ in the JOIN operation, and the `BIND` clause is reflected in the top-level PROJ. When this IQ is unfolded using the mapping given above, occurrences of `triple` are replaced by unions of appropriate mapping assertions (with matching predicates, for example). Note that, in general, since the RDF dataset is a *set* of triples and quadruples, one needs to insert DISTINCT above the union of mapping assertion SQL queries; in this case, however, the DISTINCT can be omitted because the first attribute is a primary key in the tables and the values of $?n$ are disjoint in the three branches (in terms of `sameTerm`). So, we obtain the following:

$$\begin{array}{l}\textsc{Proj}^{?s}_{?s/\text{numeric-add}(?n,?m)}\ \textsc{Join}_{\neg\mathit{isNull}(\text{numeric-add}(?n,?m))}\\ \quad\textsc{Union}\\ \qquad\textsc{Proj}^{?x,?n}_{?x/\texttt{rdf}(\texttt{:b\{\}}(y_1),\text{IRI}),\ ?n/\texttt{rdf}(\text{i2t}(z_1),\texttt{xsd:integer})}\ T_1(y_1,z_1)\\ \qquad\textsc{Proj}^{?x,?n}_{?x/\texttt{rdf}(\texttt{:b\{\}}(y_2),\text{IRI}),\ ?n/\texttt{rdf}(\text{d2t}(z_2),\texttt{xsd:decimal})}\ T_2(y_2,z_2)\\ \qquad\textsc{Proj}^{?x,?n}_{?x/\texttt{rdf}(\texttt{:b\{\}}(y_3),\text{IRI}),\ ?n/\texttt{rdf}(z_3,\texttt{xsd:string})}\ T_3(y_3,z_3)\\ \quad\textsc{Proj}^{?x,?m}_{?x/\texttt{rdf}(\texttt{:b\{\}}(y_4),\text{IRI}),\ ?m/\texttt{rdf}(\text{i2t}(z_4),\texttt{xsd:integer})}\ T_4(y_4,z_4).\end{array}$$

This query, however, cannot be directly translated into SQL because, for example, it has occurrences of SPARQL functions (`numeric-add`).

3.2 Translating (Multi-Typed) SPARQL Functions into SQL Functions

Recall the two main difficulties in translating SPARQL functions into SQL. First, when a SPARQL function is not applicable to its argument (e.g., `numeric-add` to `xsd:string`), then the result is the *type error*, which, in our example, means that the variable remains unbound in the solution mapping; in SQL, such a query would be deemed invalid (and one would have no results). Second, the type of the result may depend on the types of the arguments: `numeric-add` yields an `xsd:integer` on `xsd:integers` and an `xsd:decimal` on `xsd:decimals`. Using the example above, we illustrate how an IQ with a multi-typed SPARQL operation can be transformed into an IQ with standard SQL operations.

First, the substitutions of PROJ operators are lifted as high in the expression tree as possible. In the process, functional terms may need to be decomposed so that some of their arguments can be lifted, even though other arguments are blocked. For example, the substitution entries for $?n$ differ in the three branches of the UNION, and each needs to be decomposed: e.g., $?n/\texttt{rdf}(\text{i2t}(z_1),\ \texttt{xsd:integer})$ is decomposed into $?n/\texttt{rdf}(v,t)$, $v/\text{i2t}(z_1)$, and $t/\texttt{xsd:integer}$. Variables v and t are re-used in the other branches, and, after the decomposition, all children of the UNION share the same entry $?n/\texttt{rdf}(v,t)$ in their PROJ constructs, and so, this entry can be lifted up to the top. Note, however, that the entries for v and t remain blocked by the UNION. Observe that one child of the UNION can be pruned when propagating the JOIN conditions down: the condition is unsatisfiable as applying `numeric-add` to `xsd:string` results in the SPARQL type error, which is equivalent to false when used as a filter. Thus, we obtain

$$\begin{array}{l}
\textsc{Proj}^{?s}_{?s/\texttt{numeric-add}(\texttt{rdf}(v,t),\ \texttt{rdf}(\text{i2t}(z_4),\ \texttt{xsd:integer}))}\\
\quad \textsc{Join}_{\neg \mathit{isNull}(\texttt{numeric-add}(\texttt{rdf}(v,t),\ \texttt{rdf}(\text{i2t}(z_4),\ \texttt{xsd:integer})))}\\
\qquad \textsc{Union}\big(\textsc{Proj}^{y,v,t}_{v/\text{i2t}(z_1),t/\texttt{xsd:integer}} T_1(y,z_1), \textsc{Proj}^{y,v,t}_{v/\text{d2t}(z_2),t/\texttt{xsd:decimal}} T_2(y,z_2)\big)\\
\qquad T_4(y,z_4).
\end{array}$$

However, the type of the first argument of `numeric-add` is still unknown at this point, which prevents transforming it into a SQL function. So, first, the substitution entries for t are replaced by $t/f(1)$ and $t/f(2)$, respectively, where f is a freshly generated dictionary function that maps 1 and 2 to `xsd:integer` and `xsd:decimal`, respectively. Then, f can be lifted to the JOIN and PROJ by introducing a fresh variable p:

$$\begin{array}{l}
\textsc{Proj}^{?s}_{?s/\texttt{numeric-add}(\texttt{rdf}(v,\boldsymbol{f(p)}),\ \texttt{rdf}(\text{i2t}(z_4),\ \texttt{xsd:integer}))}\\
\quad \textsc{Join}_{\neg \mathit{isNull}(\texttt{numeric-add}(\texttt{rdf}(v,\boldsymbol{f(p)}),\ \texttt{rdf}(\text{i2t}(z_4),\ \texttt{xsd:integer})))}\\
\qquad \textsc{Union}\big(\textsc{Proj}^{y,v,\boldsymbol{p}}_{v/\text{i2t}(z_1),\ \boldsymbol{p/1}}\ T_1(y,z_1),\ \ \textsc{Proj}^{y,v,\boldsymbol{p}}_{v/\text{d2t}(z_2),\ \boldsymbol{p/2}}\ T_2(y,z_2)\big)\\
\qquad T_4(y,z_4),
\end{array}$$

where the changes are emphasized in boldface.

Now, the type of the first argument of `numeric-add` is either `xsd:integer` or `xsd:decimal`, and so, it can be transformed into a complex functional term with SQL `+`: the sum is on `INTEGER`s if p is 1, and on `DOUBLE`s otherwise. Observe that these sums are cast back to `TEXT` to produce RDF term lexical values. Now, the JOIN condition is equivalent to `true` because the `numeric-add` does not produce `NULL` without invalid input types and nullable variables. A similar argument applies to PROJ, and we get

$$\begin{array}{l}
\textsc{Proj}^{?s}_{?s/\texttt{rdf}(\texttt{IF}(p=1,\ \text{i2t}(\text{t2i}(v)+z_4),\ \text{d2t}(\text{t2d}(v)+\text{i2d}(z_4))),\ \texttt{IF}(p=1,\ \texttt{xsd:integer},\ \texttt{xsd:decimal}))}\\
\quad \textsc{Join}\\
\qquad \textsc{Union}\big(\textsc{Proj}^{y,v,p}_{v/\text{i2t}(z_1),\ p/1}\ T_1(y,z_1),\ \ \textsc{Proj}^{y,v,p}_{v/\text{d2t}(z_2),\ p/2}\ T_2(y,z_2)\big)\\
\qquad T_4(y,z_4),
\end{array}$$

which can now be translated into SQL. We would like to emphasize that only the SPARQL variables can be multi-typed in IQs, while the variables for database attributes will always have a unique type, which is determined by the datatype of the attribute.

As a second example, we consider the following aggregation query:

```
SELECT ?x (SUM(?n) AS ?s)  WHERE { ?x :p ?n . } GROUP BY ?x
```

This query uses the same mapping as above, where the values of data property `:p` can belong to `xsd:integer`, `xsd:decimal`, and `xsd:string` from `INTEGER`, `DECIMAL`, and `TEXT` database attributes. The three possible ranges for `:p` require careful handling because of `GROUP BY` and `SUM`: in each group of tuples with the same x, we need to compute (separate) sums of all `INTEGER`s and `DECIMAL`s, as well as indicators of whether there are any `TEXT`s and `DECIMAL`s: the former is needed because any string in a group results in a *type error* and undefined sum; the latter determines the type of the sum if all values in the group are numerical. The following IQ is the final result of the transformations:

$$\begin{array}{l}
\textsc{Proj}^{?x,?s}_{\substack{?x/\texttt{rdf}(\texttt{:b\{\}}(y),\text{IRI}),\\ ?s/\texttt{rdf}(\texttt{IF}(ct>0,\ \texttt{NULL},\ \texttt{IF}(cd=0,\ \text{i2t}(\mathit{coalesce}(si,0)),\ \text{d2t}(sd+\text{i2d}(\mathit{coalesce}(si,0))))),\\ \qquad \texttt{IF}(ct>0,\ \texttt{NULL},\ \texttt{IF}(cd=0,\ \texttt{xsd:integer},\ \texttt{xsd:decimal})))}}\\
\quad \textsc{Agg}^{y}_{cd/\texttt{COUNT}(d),\ ct/\texttt{COUNT}(t),\ si/\texttt{SUM}(i),\ sd/\texttt{SUM}(d)}\\
\qquad \textsc{Union}\big(\textsc{Proj}^{y,i,d,t}_{\substack{d/\texttt{NULL},\\ t/\texttt{NULL}}}\ T_1(y,i),\ \ \textsc{Proj}^{y,i,d,t}_{\substack{i/\texttt{NULL},\\ t/\texttt{NULL}}}\ T_2(y,d),\ \ \textsc{Proj}^{y,i,d,t}_{\substack{i/\texttt{NULL},\\ d/\texttt{NULL}}}\ T_3(y,t)\big).
\end{array}$$

Note that the branches of UNION have the same projected variables, padded by `NULL`.

3.3 Optimization Techniques

Being able to transform SPARQL queries into SQL ones is a must-have requirement, but making sure that they can be efficiently processed by the underlying DBMS is essential for the VKG approach. This topic has been extensively studied during the past decade, and an array of optimization techniques, such as redundant join elimination using primary and foreign keys [6,19,22,18] and pushing down JOINs to the data-level [13], are now well-known and implemented by many systems. In addition to these, *Ontop* v4 exploits several recent techniques, including the ones proposed in [25] for optimizing left joins due to OPTIONALs and MINUSes in the SPARQL queries.

Self-join Elimination for Denormalized Data. We have implemented a novel self-join elimination technique to cover a common case where data is partially denormalized. We illustrate it on the following example with a single database table `loan` with primary key `id` and all non-nullable attributes. For instance, `loan` can contain the following tuples:

id	amount	organisation	branch
10284124	5000	Global Bank	Denver
20242432	7000	Trade Bank	Chicago
30443843	100000	Global Bank	Miami
40587874	40000	Global Bank	Denver

The mapping for data property `:hasAmount` and object properties `:grantedBy` and `:branchOf` constructs, for each tuple in `loan`, three triples to specify the loan amount, the bank branch that granted it, and the head organisation for the bank branch:

$$\begin{array}{rl}
\texttt{loan}(x, a, \_, \_) \rightsquigarrow & \texttt{triple}(\texttt{rdf}(\texttt{:l\{\}}(x), \text{IRI}), \texttt{:hasAmount},\ \texttt{rdf}(\text{i2t}(a), \texttt{xsd:integer})),\\
\texttt{loan}(x, \_, o, b) \rightsquigarrow & \texttt{triple}(\texttt{rdf}(\texttt{:l\{\}}(x), \text{IRI}), \texttt{:grantedBy},\ \texttt{rdf}(\texttt{:b\{\}/\{\}}(o, b), \text{IRI})),\\
\texttt{loan}(\_, \_, o, b) \rightsquigarrow & \texttt{triple}(\texttt{rdf}(\texttt{:b\{\}/\{\}}(o, b), \text{IRI}), \texttt{:branchOf},\ \texttt{rdf}(\texttt{:o\{\}}(o), \text{IRI}))
\end{array}$$

(we use underscores instead of variables for attributes that are not projected). Observe that the last assertion is not 'normalized': the same triple can be extracted from many different tuples (in fact, it yields a copy of the triple for each loan granted by the branch). To guarantee that the RDF graph is a set, these duplicates have to be eliminated.

We now consider the following SPARQL query extracting the number and amount of loans granted by each organisation:

```
SELECT ?o (COUNT(?l) AS ?c) (SUM(?a) AS ?s) WHERE {
  ?l :hasAmount ?a. ?l :grantedBy ?b. ?b :branchOf ?o } GROUP BY ?o
```

After unfolding, we obtain the following IQ:

$$\begin{array}{l}\textsc{Agg}^{?o}_{?c/\texttt{SPARQL\_Count}(l),\ ?s/\texttt{SPARQL\_Sum}(a)}\ \textsc{Join}\\ \qquad \textsc{Proj}^{?l,?a}_{?l/\texttt{rdf}(\texttt{:l\{\}}(x_1),\mathrm{IRI}),\ ?a/\texttt{rdf}(\mathrm{i2t}(a_1),\texttt{xsd:integer})}\ \texttt{loan}(x_1, a_1, \_, \_)\\ \qquad \textsc{Proj}^{?l,?b}_{?l/\texttt{rdf}(\texttt{:l\{\}}(x_2),\mathrm{IRI}),\ ?b/\texttt{rdf}(\texttt{:b\{\}/\{\}}(o_2,b_2),\mathrm{IRI})}\ \texttt{loan}(x_2, \_, o_2, b_2)\\ \qquad \textsc{Distinct}\ \textsc{Proj}^{?b,?o}_{?b/\texttt{rdf}(\texttt{:b\{\}/\{\}}(o_3,b_3),\mathrm{IRI}),\ ?o/\texttt{rdf}(\texttt{:o\{\}}(o_3),\mathrm{IRI})}\ \texttt{loan}(\_, \_, o_3, b_3).\end{array}$$

Note that the DISTINCT in the third child of the JOIN is required to eliminate duplicates (none is needed for the other two since `id` is the primary key of table `loan`).

The first step is lifting the PROJ. For the substitution entries below the DISTINCT, some checks need to be done before (partially) lifting their functional terms. The `rdf` function used by $?o$ and $?b$ is injective by design and can always be lifted. Their first arguments are IRI template functional terms. Both IRI templates, `:o{}` and `:b{}/{}`, are injective (see Remark 1): the former is unary, the latter has a safe separator `/` between its arguments. Consequently, both can be lifted. Note that these checks only concern functional terms, as constants can always be lifted above DISTINCTs. The substitution entry for $?l$ is lifted above the AGG because it is its group-by variable. Other entries are used for substituting the arguments of the aggregation functions. Here, none of the variables is multi-typed. After simplifying the functional terms, we obtain the IQ

$$\begin{array}{l}\textsc{Proj}^{?o,?c,?s}_{?o/\texttt{rdf}(\texttt{:o\{\}}(o_2),\mathrm{IRI}),\ ?c/\texttt{rdf}(\mathrm{i2t}(n),\texttt{xsd:integer}),\ ?s/\texttt{rdf}(\mathrm{i2t}(m),\texttt{xsd:integer})}\\ \qquad \textsc{Agg}^{o_2}_{n/\texttt{COUNT}(x_1),\ m/\texttt{SUM}(a_1)}\\ \qquad\qquad \textsc{Join}\big(\texttt{loan}(x_1, a_1, \_, \_),\ \texttt{loan}(x_1, \_, o_2, b_2),\ \textsc{Distinct}\ \texttt{loan}(\_, \_, o_2, b_2)\big).\end{array}$$

Next, the well-known self-join elimination is applied to the first two children of the JOIN (which is over the primary key). Then, the DISTINCT commutes with the JOIN since the other child of JOIN is also a set (due to the primary key), obtaining the sub-IQ

$$\textsc{Distinct}\ \ \textsc{Join}\big(\texttt{loan}(x_1, a_1, o_2, b_2), \texttt{loan}(\_, \_, o_2, b_2)\big),$$

on which our new self-join elimination technique can be used, as the two necessary conditions are satisfied. First, the JOIN does not need to preserve cardinality due to the DISTINCT above it. Second, all the variables projected by the second child (o_2 and b_2) of the JOIN are also projected by the first child. So, we can eliminate the second child, but have to insert a filter requiring the shared variables o_2 and b_2 to be non-`NULL`:

$$\textsc{Distinct}\ \ \textsc{Filter}_{\neg \mathit{isNull}(o_2) \wedge \neg \mathit{isNull}(b_2)}\ \ \texttt{loan}(x_1, a_1, o_2, b_2).$$

The result can be further optimized by observing that the attributes for o_2 and b_2 are non-nullable and that the DISTINCT has no effect because the remaining data atom produces no duplicates. So, we arrive at

$$\textsc{Proj}^{?o,?c,?s}_{?o/\texttt{rdf}(\texttt{:o\{\}}(o_2),\text{IRI}),\ ?c/\texttt{rdf}(\text{i2t}(n),\texttt{xsd:integer}),\ ?s/\texttt{rdf}(\text{i2t}(m),\texttt{xsd:integer})}$$
$$\textsc{Agg}^{o_2}_{n/\texttt{COUNT}(x_1),\ m/\texttt{SUM}(a_1)}\ \texttt{loan}(x_1, a_1, o_2, \_),$$

where b_2 is replaced by _ because it is not used elsewhere.

3.4 From IQ to SQL

In the VKG approach almost all query processing is delegated to the DBMS. *Ontop* v4 performs only the top-most projection, which typically transforms database values into RDF terms, as illustrated by the last query above. The subquery under this projection must not contain any RDF value nor any SPARQL function. As highlighted above, our IQ guarantees that such a subquery is not multi-typed.

In contrast to SPARQL, the ANSI/ISO SQL standards are very loosely followed by DBMS vendors. There is very little hope for generating reasonably rich SQL that would be interoperable across multiple vendors. Given the diversity of the SQL ecosystem, in *Ontop* v4 we model each supported SQL dialect in a fine-grained manner: in particular, we model *(i)* their datatypes, *(ii)* their conventions in terms of attribute and table identifiers and query modifiers, *(iii)* the semantics of their functions, *(iv)* their restrictions on clauses such as `WHERE` and `ORDER BY`, and *(v)* the structure of their data catalog. *Ontop* v4 directly uses the concrete datatypes and functions of the targeted dialect in IQ by means of Java factories whose dialect-specific implementations are provided through a dependency injection mechanism. Last but not least, *Ontop* v4 allows IQ to contain arbitrary, including user-defined, SQL functions from the queries of the mapping.

4 Evaluation

Compliance of *Ontop* v4 with relevant W3C recommendations is discussed in Section 4.1, and performance and comparison with other systems in Section 4.2.

4.1 Compliance with W3C Recommendations

Since the relevant W3C standards have very rich sets of features, and they also interplay with each other, it is difficult to enumerate all the cases. The different behaviors of DBMSs make the situation even more complex and add another dimension to consider. Nevertheless, we describe our testing infrastructure and do our best to summarize the behavior of *Ontop* with all the different standards.

Testing Infrastructure. To ensure the correct behavior of the system, we developed a rich testing infrastructure. The code base includes a large number of unit test cases. To test against different database systems, we developed a Docker-based infrastructure for creating DB-specific instances for the tests[1]. It uses docker-compose to generate a cluster of DBs including MySQL, PostgreSQL, Oracle, MS SQL Server, and DB2.

SPARQL 1.1 [12]. In Table 1, we present a summary of *Ontop* v4 compliance with SPARQL 1.1, where rows correspond to sections of the WC3 recommendation. Most of the features are supported, but some are unsupported or only partially supported.

[1] `https://github.com/ontop/ontop-dockertests`

Table 1: SPARQL Compliance: unsupported features are ~~crossed out~~.

Section in SPARQL 1.1 [12]	Features	Coverage
5–7. Graph Patterns, etc.	BGP, FILTER, OPTIONAL, UNION	4/4
8. Negation	MINUS, ~~FILTER [NOT] EXISTS~~	1/2
9. Property Paths	~~PredicatePath~~, ~~InversePath~~, ~~ZeroOrMorePath~~, ...	0
10. Assignment	BIND, VALUES	2/2
11. Aggregates	COUNT, SUM, MIN, MAX, AVG, GROUP_CONCAT, SAMPLE	6/6
12. Subqueries	Subqueries	1/1
13. RDF Dataset	GRAPH, ~~FROM [NAMED]~~	1/2
14. Basic Federated Query	~~SERVICE~~	0
15. Solution Seqs. & Mods.	ORDER BY, SELECT, DISTINCT, REDUCED, OFFSET, LIMIT	6/6
16. Query Forms	SELECT, CONSTRUCT, ASK, DESCRIBE	4/4
17.4.1. Functional Forms	BOUND, ~~IF~~, COALESCE, ~~EXISTS~~, ~~NOT EXISTS~~, ||, &&, =, sameTerm, ~~IN~~, ~~NOT IN~~	6/11
17.4.2. Fns. on RDF Terms	isIRI, isBlank, isLiteral, isNumeric, str, lang, datatype, ~~IRI~~, ~~BNODE~~, ~~STRDT~~, ~~STRLANG~~, UUID, STRUUID	9/13
17.4.3. Fns. on Strings	STRLEN, SUBSTR, UCASE, LCASE, STRSTARTS, STRENDS, CONTAINS, STRBEFORE, STRAFTER, ENCODE_FOR_URI, CONCAT, langMatches, REGEX, REPLACE	14/14
17.4.4. Fns. on Numerics	abs, round, ceil, floor, RAND	5/5
17.4.5. Fns. on Dates&Times	now, year, month, day, hours, minutes, seconds, ~~timezone~~, tz	8/9
17.4.6. Hash Functions	MD5, SHA1, SHA256, SHA384, SHA512	5/5
17.5. XPath Constructor Fns.	~~casting~~	0
17.6. Extensible Value Testing	~~user defined functions~~	0

– Property paths are not supported: the ZeroOrMorePath (*) and OneOrMorePath (+) operators require linear recursion, which is not part of IQ yet. An initial investigation of using SQL Common Table Expressions (CTEs) for linear recursion was done in the context of SWRL [26], but a proper implementation would require dedicated optimization techniques.
– [NOT] EXISTS is difficult to handle due to its non-compositional semantics, which is not defined in a bottom-up fashion. Including it in IQ requires further investigation.
– Most of the missing SPARQL functions (Section 17.4) are not so challenging to implement but require a considerable engineering effort to carefully define their translations into SQL. We will continue the process of implementing them gradually and track the progress in a dedicated issue[2].
– The 5 hash functions and functions REPLACE and REGEX for regular expressions have limited support because they heavily depend on the DBMS: not all DBMSs provide

[2] https://github.com/ontop/ontop/issues/346

all hash functions, and many DBMSs have their own regex dialects. Currently, the SPARQL regular expressions of `REPLACE` and `REGEX` are simply sent to the DBMS.

– In the implementation of functions `STRDT`, `STRLANG`, and `langMatches`, the second argument has to a be a constant: allowing variables will have a negative impact on the performance in our framework.

R2RML [9]. *Ontop* is fully compliant with R2RML. In particular, the support of `rr:GraphMap` for RDF datasets and blank nodes has been introduced in *Ontop* v4. The optimization hint `rr:inverseExpression` is ignored in the current version, but this is compliant with the W3C recommendation. In the combination of R2RML with OWL, however, ontology axioms (a TBox in the Description Logic parlance) could also be constructed in a mapping: e.g., $T_1(x, y) \rightsquigarrow$ `:{`x`} rdfs:subClassOf :{`y`}`. Such mappings are not supported in online query answering, but one can materialize the triples offline and then include them in the ontology manually.

OWL 2 QL [14] and SPARQL 1.1 Entailment Regimes [11]. These two W3C recommendations define how to use ontological reasoning in SPARQL. *Ontop* supports them with the exception of querying the TBox, as in `SELECT * WHERE { ?c rdfs:subClassOf :Person. ?x a ?c }`. Although we have investigated this theoretically and implemented a prototype [13], a more serious implementation is needed for IQ, with special attention to achieving good performance. This is on our agenda.

SPARQL 1.1 Protocol [10] and SPARQL Endpoint. We reimplemented the new SPARQL endpoint from scratch and designed a new command-line interface for it. It is stateless and suitable for containers. In particular, we have created a Docker image for the *Ontop* SPARQL endpoint[3], which has greatly simplified deployment. The endpoint is also packed with several new features, like customization of the front page with predefined SPARQL queries, streaming query answers, and result caching.

4.2 Performance and Comparison with Other VKG Systems

Performance evaluation of *Ontop* has been conducted since *Ontop* v1 by ourselves and others in a number of scientific papers. Here we only summarize two recent independent evaluations of *Ontop* v3. Recall that the main focus of *Ontop* v4 compared to v3 has been the extension with new features. Hence, we expect similar results for *Ontop* v4.

Chaloupka and Necasky [7] evaluated four VKG systems, namely, Morph, *Ontop*, SparqlMap, and their own EVI, using the Berlin SPARQL Benchmark (BSBM). D2RQ and Ultrawrap were not evaluated: D2RQ has not been updated for years, and Ultrawrap is not available for research evaluation. Only *Ontop* and EVI were able to load the authors' version of the R2RML mapping for BSBM. EVI supports only SQL Server, while *Ontop* supports multiple DBMSs. In the evaluation, EVI outperformed *Ontop* on small datasets, but both demonstrated similar performance on larger datasets, which can be explained by the fact that *Ontop* performs more sophisticated (and expensive) optimizations during the query transformation step.

Namici and De Giacomo [15] evaluated *Ontop* and Mastro on the NPD and ACI benchmarks, both of which have complex ontologies. Some SPARQL queries had to

[3] `https://hub.docker.com/r/ontop/ontop-endpoint`

be adapted for Mastro because it essentially supports only unions of CQs. In general *Ontop* was faster on NPD, while Mastro was faster on ACI.

Both independent evaluations confirm that although *Ontop* is not always the fastest, its performance is very robust. In the future, we will carry out more evaluations, in particular for the new features of *Ontop* v4.

It is important to stress that when choosing a VKG system, among many different criteria, performance is only one dimension to consider. Indeed, in [7], also the aspects of usability, completeness, and soundness have been evaluated. When considering all of these, *Ontop* is a clear winner. In our recent survey [24], we have also listed the main features of popular VKG systems, including D2RQ, Mastro, Morph, *Ontop*, Oracle Spatial and Graph, and Stardog. Overall, it is fair to claim that *Ontop* is the most mature *open source* VKG system currently available.

5 Community Building and Adoption

Ontop is distributed under the Apache 2 license through several channels. Ready-to-use binary releases, including a command line tool and a Protégé bundle with an *Ontop* plugin, are published on Sourceforge since 2015. There have been **30K+ downloads** in the past 5 years according to Sourceforge[4]. The *Ontop* plugin for Protégé is available also in the plugin repository of Protégé, through which users receive auto-updates. A Docker image of the SPARQL endpoint is available at Docker Hub since the *Ontop* v3 release, and it has been pulled 1.1K times. The documentation, including tutorials, is available at the official website[5].

Ontop is the product of a hard-working developer community active for over a decade. Nowadays, the development of *Ontop* is backed by different research projects (at the local, national, and EU level) at the Free University of Bozen-Bolzano and by Ontopic s.r.l. It also receives regular important contributions from Birkbeck, University of London. As of 13 August 2020, the GitHub repository[6] consists of 11,511 git commits from 25 code contributors, among which 10 have contributed more than 100 commits each. An e-mail list[7] created in August 2013 for discussion currently includes 270 members and 429 topics. In Github, 312 issues have been created and 270 closed.

To make *Ontop* sustainable, it needed to be backed up by a commercial company, because a development project running at a public university cannot provide commercial support to its users, and because not all developments are suitable for a university research group. So, Ontopic s.r.l.[8] was born in April 2019, as the first spin-off of the Free University of Bozen-Bolzano[9]. It provides commercial support for the *Ontop* system and consulting services that rely on it, with the aim to push the VKG technology to industry. Ontopic has now become the major source code contributor of *Ontop*.

Ontop has been adopted in many academic and industrial use cases. Due to its liberal Apache 2 license, it is essentially impossible to obtain a complete picture of all use cases

[4] `https://sourceforge.net/projects/ontop4obda/files/stats/timeline`
[5] `https://ontop-vkg.org/`
[6] `https://github.com/ontop/ontop/`
[7] `https://groups.google.com/forum/#!aboutgroup/ontop4obda`
[8] `http://ontopic.biz/`
[9] `https://www.unibz.it/it/news/132449` (in Italian)

and adoptions. Indeed, apart from the projects in which the research and development team is involved directly, we normally learn about a use case only when the users have some questions or issues with *Ontop*, or when their results have been published in a scientific paper. Nevertheless, a few significant use cases have been summarized in a recent survey paper [24]. Below, we highlight two commercial deployments of *Ontop*, in which Ontopic has been involved.

UNiCS[10] is an open data platform for research and innovation developed by SIRIS Academic in Spain. Using *Ontop*, the UNiCS platform integrates a large variety of data sources for decision and policy makers, including data produced by government bodies, data on the higher education & research sector, as well as companies' proprietary data. For instance, the Toscana Open Research (TOR) portal[11] is one such deployment of UNiCS. It is designed to communicate and enhance the Tuscan regional system of research, innovation, and higher education and to promote increasingly transparent and inclusive governance. Recently, Ontopic has also been offering dedicated training courses for TOR users, so that they can autonomously formulate SPARQL queries to perform analytics, and even create VKGs to integrate additional data sources.

Open Data Hub-Virtual Knowledge Graph is a joint project between NOI Techpark and Ontopic for publishing South Tyrolean tourism data as a Knowledge Graph. Before the project started, the data was accessible through a JSON-based Web API, backed by a PostgreSQL database. We created a VKG over the database and a SPARQL endpoint[12] that is much more flexible and powerful than the old Web API. Also, we created a Web Component[13], which can be embedded into any web page like a standard HTML tag, to visualize SPARQL query results in different ways.

6 Conclusion

Ontop is a popular open-source virtual knowledge graph system. It is the result of an active research and development community and has been adopted in many academic and industrial projects. In this paper, we have presented the challenges, design choices, and new features of the latest release v4 of *Ontop*.

Acknowledgements. This research has been partially supported by the Wallenberg AI, Autonomous Systems and Software Program (WASP) funded by the Knut and Alice Wallenberg Foundation, by the Italian Basic Research (PRIN) project HOPE, by the EU H2020 project INODE, grant agreement 863410, by the CHIST-ERA project PACMEL, by the Free Uni. of Bozen-Bolzano through the projects QUADRO, KGID, and GeoVKG, and by the project IDEE (FESR1133) through the European Regional Development Fund (ERDF) Investment for Growth and Jobs Programme 2014-2020.

References

1. M. Arenas, G. Gottlob, and A. Pieris. Expressive languages for querying the semantic web. *ACM Trans. Database Syst.*, 43(3):13:1–13:45, 2018.

[10] http://unics.cloud/
[11] http://www.toscanaopenresearch.it/
[12] https://sparql.opendatahub.bz.it/
[13] https://github.com/noi-techpark/webcomp-kg

2. A. Artale, D. Calvanese, R. Kontchakov, and M. Zakharyaschev. The DL-Lite family and relations. *J. Artif. Intell. Res.*, 36:1–69, 2009.
3. D. Calvanese, B. Cogrel, S. Komla-Ebri, R. Kontchakov, D. Lanti, M. Rezk, M. Rodriguez-Muro, and G. Xiao. Ontop: answering SPARQL queries over relational databases. *SWJ*, 8(3):471–487, 2017.
4. D. Calvanese, G. De Giacomo, D. Lembo, M. Lenzerini, A. Poggi, M. Rodriguez-Muro, R. Rosati, M. Ruzzi, and D. F. Savo. The MASTRO system for ontology-based data access. *SWJ*, 2(1):43–53, 2011.
5. D. Calvanese, G. De Giacomo, D. Lembo, M. Lenzerini, and R. Rosati. Tractable reasoning and efficient query answering in description logics: The *DL-Lite* family. *JAR*, 2007.
6. U. S. Chakravarthy, J. Grant, and J. Minker. Logic-based approach to semantic query optimization. *ACM TODS*, 15(2):162–207, 1990.
7. M. Chaloupka and M. Necasky. Using Berlin SPARQL benchmark to evaluate relational database virtual SPARQL endpoints. Submitted to SWJ, 2020.
8. A. Chebotko, S. Lu, and F. Fotouhi. Semantics preserving SPARQL-to-SQL translation. *DKE*, 68(10):973–1000, 2009.
9. S. Das, S. Sundara, and R. Cyganiak. R2RML: RDB to RDF mapping language. W3C recommendation, W3C, 2012.
10. L. Feigenbaum, G. T. Williams, K. G. Clark, and E. Torres. SPARQL 1.1 protocol. W3C recommendation, W3C, 2013.
11. B. Glimm and C. Ogbuji. SPARQL 1.1 entailment regimes. W3C recommendation, 2013.
12. S. Harris, A. Seaborne, and E. Prud'hommeaux. SPARQL 1.1 query language. W3C recommendation, W3C, 2013.
13. R. Kontchakov, M. Rezk, M. Rodriguez-Muro, G. Xiao, and M. Zakharyaschev. Answering SPARQL queries over databases under OWL 2 QL entailment regime. In *Proc. ISWC*, 2014.
14. B. Motik, B. Cuenca Grau, I. Horrocks, Z. Wu, A. Fokoue, and C. Lutz. OWL 2 Web Ontology Language: Profiles. W3C Recommendation, W3C, 2012.
15. M. Namici and G. De Giacomo. Comparing query answering in OBDA tools over W3C-compliant specifications. In *Proc. DL*, volume 2211. CEUR-WS.org, 2018.
16. A. Poggi, D. Lembo, D. Calvanese, G. De Giacomo, M. Lenzerini, and R. Rosati. Linking data to ontologies. *J. Data Sem.*, 10:133–173, 2008.
17. A. Polleres. From SPARQL to rules (and back). In *WWW*, pages 787–796. ACM, 2007.
18. F. Priyatna, Ó. Corcho, and J. F. Sequeda. Formalisation and experiences of R2RML-based SPARQL to SQL query translation using morph. In *WWW*, pages 479–490, 2014.
19. M. Rodriguez-Muro, R. Kontchakov, and M. Zakharyaschev. Ontology-based data access: Ontop of databases. In *Proc. ISWC*, pages 558–573. Springer, 2013.
20. M. Rodriguez-Muro and M. Rezk. Efficient SPARQL-to-SQL with R2RML mappings. *J. Web Sem.*, 33:141–169, 2015.
21. J. F. Sequeda, M. Arenas, and D. P. Miranker. Ontology-based data access using views. In *Proc. RR*, volume 7497 of *LNCS*, pages 262–265. Springer, 2012.
22. J. Unbehauen, C. Stadler, and S. Auer. Optimizing SPARQL-to-SQL rewriting. In *Proc. IIWAS*, pages 324–330. ACM, 2013.
23. G. Xiao, D. Calvanese, R. Kontchakov, D. Lembo, A. Poggi, R. Rosati, and M. Zakharyaschev. Ontology-based data access: A survey. In *Proc. IJCAI*, 2018.
24. G. Xiao, L. Ding, B. Cogrel, and D. Calvanese. Virtual knowledge graphs: An overview of systems and use cases. *Data Intelligence*, 1:201–223, 2019.
25. G. Xiao, R. Kontchakov, B. Cogrel, D. Calvanese, and E. Botoeva. Efficient handling of SPARQL OPTIONAL for OBDA. In *Proc. ISWC*, pages 354–373, 2018.
26. G. Xiao, M. Rezk, M. Rodriguez-Muro, and D. Calvanese. Rules and ontology based data access. In *Proc. RR*, LNCS. Springer, 2014.