# RDF Analytics: Lenses over Semantic Graphs

Dario Colazzo
U. Paris Dauphine, France
dario.colazzo@dauphine.fr

François Goasdoué
U. Rennes 1, France
francois.goasdoue@univ-rennes1.fr

Ioana Manolescu
Inria & U. Paris-Sud, France
ioana.manolescu@inria.fr

Alexandra Roatiş
U. Paris-Sud & Inria, France
alexandra.roatis@inria.fr

## ABSTRACT

The development of Semantic Web (RDF) brings new requirements for data analytics tools and methods, going beyond querying to semantics-rich analytics through warehouse-style tools. In this work, we fully redesign, from the bottom up, core data analytics concepts and tools in the context of RDF data, leading to the first complete formal framework for warehouse-style RDF analytics. Notably, we define *i*) *analytical schemas* tailored to heterogeneous, semantics-rich RDF graph, *ii*) *analytical queries* which (beyond relational cubes) allow flexible querying of the data and the schema as well as powerful aggregation and *iii*) *OLAP-style operations*. Experiments on a fully-implemented platform demonstrate the practical interest of our approach.

## Categories and Subject Descriptors

H.2.1 [**Database Management**]: Logical Design;
H.2.7 [**Database Management**]: Database Administration
—*Data warehouse and repository*

## Keywords

RDF; data warehouse; OLAP

## 1. INTRODUCTION

The development of Semantic Web data represented within W3C's Resource Description Framework [33] (or RDF, in short), and the associated standardization of the SPARQL query language now at v1.1 [35] has lead to the emergence of many systems capable of storing, querying, and updating RDF, such as OWLIM [38], RDF-3X [28], Virtuoso [15] etc. However, as more and more RDF datasets are made available, in particular Linked Open Data, application requirements also evolve. In the following scenario, we identify by (*i*)-(*v*) a set of application needs, for further reference.

Alice is a software engineer working for an IT company responsible of developing **user applications based on open (RDF) data** from the region of Grenoble. From a dataset describing the region's restaurants, she must build a clickable map showing for each district of the region, "the number of restaurants and their average rating per type of cuisine".

The data is (*i*) heterogeneous, as information, such as the menu, opening hours or closing days, is available for some restaurants, but not for others. Fortunately, Alice studied data warehousing [22]. She thus designs a *relational data warehouse* (RDW, in short), writes some SPARQL queries to extract tabular data from the restaurant dataset (filled with nulls when data is missing), loads them in the RDW and builds the application using standard RDW tools.

The client is satisfied, and soon Alice is given two more datasets, on shops and museums; she is asked to (*ii*) merge them in the application already developed. Alice has a hard time: she had designed a classical *star schema* [23], centered on restaurants, which cannot accommodate shops. She builds a second RDW for shops and a third for museums.

The application goes online and soon bugs are noticed. When users search for landmarks in an area, they don't find anything, although there are multiple museums. Alice knows this happens because (*iii*) the RDW does not capture the fact that *a museum is a landmark*. With a small redesign of the RDW, Alice corrects this, but she is left with a nagging feeling that there may be many other relationships present in the RDF which she missed in her RDW. Further, the client wants the application to find (*iv*) the relationships between the region and famous people related to it, e.g., Stendhal was born in Grenoble. In Alice's RDWs, relationships between entities are part of the schema and statically fixed at RDW design time. In contrast, useful open datasets such as DBpedia [1], which could be easily linked with the RDF restaurant dataset, may involve many relationships between two classes, e.g., *bornIn, gotMarriedIn, livedIn* etc.

Finally, Alice is required to support (*v*) a new type of aggregation: for each landmark, show how many restaurants are nearby. This is impossible in Alice's RDW designs of a separate star schema for each of restaurants, shops and landmarks, as both restaurants and landmarks are central entities and Alice cannot use one as a measure for the other.

Alice's needs in setting up the application can be summarized as follows: (*i*) support of *heterogeneous* data; (*ii*) *multiple* central concepts, e.g., restaurants *and* landmarks above; (*iii*) support for *RDF semantics* when querying the warehouse, (*iv*) the possibility to *query the relationships between entities* (similar to querying the schema), (*v*) flexible choice of aggregation dimensions.

In this work, we perform a *full redesign, from the bottom up, of the core data analytics concepts and tools*, leading to a *complete formal framework for warehouse-style analytics on RDF data*; in particular, our framework is especially

suited to heterogeneous, semantic-rich corpora of Linked Open Data. Our contributions are:

- We devise a *full-RDF* warehousing approach, where the base data *and* the warehouse extent are RDF graphs. This answers to the needs (*i*), (*iii*) and (*iv*) above.

- We introduce *RDF Analytical Schemas (AnS)*, which are graphs of classes and properties themselves, having nodes (classes) connected by edges (properties) with *no single central concept (node)*. This contrasts with the typical RDW star or snowflake schemas, and caters to requirement (*ii*) above. The core idea behind many-node analytical schemas is to define *each node (resp. edge) by an independent query* over the base data.

- We define *Analytical Queries (AnQ)* over our decentralized analytical schemas. Such queries are highly flexible in the choice of measures and classifiers (requirement (*v*)), while supporting all the classical analytical cubes and operations (slice, dice etc.).

- We fully implemented our approach in an operational prototype and empirically demonstrate its interest and performance.

The remainder of this paper is organized as follows. We recall RDF data and queries in Sections 2 and 3. Section 4 presents our analytical schemas and queries, and Section 5 studies efficient query evaluation methods. Section 6 introduces typical analytical operations (slice, dice etc.) on our RDF analytical cubes. We present our experimental evaluation in Section 7, discuss related work, and then conclude.

## 2. RDF GRAPHS

An *RDF graph* (or *graph*, in short) is a set of *triples* of the form s p o. A triple states that its *subject* s has the *property* p, and the value of that property is the *object* o.

We consider only well-formed RDF triples, as per the RDF specification [33], using uniform resource identifiers (URIs), typed or un-typed literals (constants) and blank nodes (unknown URIs or literals).

**Notation**. We use s, p, o in triples as placeholders. Literals are shown as strings between quotes, e.g., "*string*". Finally, the set of values – URIs ($U$), blank nodes ($B$), literals ($L$) – of an RDF graph G is denoted Val(G).

Figure 1 (top) shows how to use triples to describe resources, that is, to express class (unary relation) and property (binary relation) assertions. The RDF standard [33] provides a set of built-in classes and properties, as part of the rdf: and rdfs: pre-defined namespaces. We use these namespaces exactly for these classes and properties, e.g., rdf:type specifies the class(es) to which a resource belongs.

Below, we formalize the representation of an RDF graph using graph notations. We use $f_{|d}$ to denote the restriction of a function $f$ to its sub-domain $d$. Our formalization follows the RDF standard [33].

DEFINITION 1. (GRAPH NOTATION OF AN RDF GRAPH) *An RDF graph is a labeled directed graph* $G = \langle \mathcal{N}, \mathcal{E}, \lambda \rangle$ *with:*

- $\mathcal{N}$ *is the set of* nodes, *let* $\mathcal{N}^0$ *denote the nodes in* $\mathcal{N}$ *having no outgoing edge, and let* $\mathcal{N}^{>0} = \mathcal{N} \setminus \mathcal{N}^0$;

- $\mathcal{E} \subseteq \mathcal{N}^{>0} \times \mathcal{N}$ *is the set of* directed edges;

- $\lambda : \mathcal{N} \cup \mathcal{E} \rightarrow U \cup B \cup L$ *is a* labeling function *such that* $\lambda_{|\mathcal{N}}$ *is injective, with* $\lambda_{|\mathcal{N}^0} : \mathcal{N}^0 \rightarrow U \cup B \cup L$ *and* $\lambda_{|\mathcal{N}^{>0}} : \mathcal{N}^{>0} \rightarrow U \cup B$, *and* $\lambda_{|\mathcal{E}} : \mathcal{E} \rightarrow U$.

| Assertion | Triple | Relational notation |
|---|---|---|
| Class | s rdf:type o | o(s) |
| Property | s p o | p(s, o) |

| Constraint | Triple | OWA interpretation |
|---|---|---|
| Subclass | s rdfs:subClassOf o | s $\subseteq$ o |
| Subproperty | s rdfs:subPropertyOf o | s $\subseteq$ o |
| Domain typing | s rdfs:domain o | $\Pi_{\mathrm{domain}}(s) \subseteq$ o |
| Range typing | s rdfs:range o | $\Pi_{\mathrm{range}}(s) \subseteq$ o |

**Figure 1: RDF (top) & RDFS (bottom) statements.**

$$G = \begin{cases} \text{user}_1 \text{ hasName "Bill", user}_1 \text{ hasAge "28", user}_1 \text{ friend user}_3, \\ \text{user}_1 \text{ bought product}_1, \text{ product}_1 \text{ rdf:type SmartPhone,} \\ \text{user}_1 \text{ worksWith user}_2, \text{ user}_2 \text{ hasAge "40", } \ldots \end{cases}$$
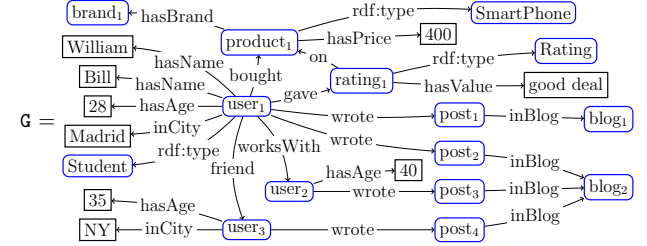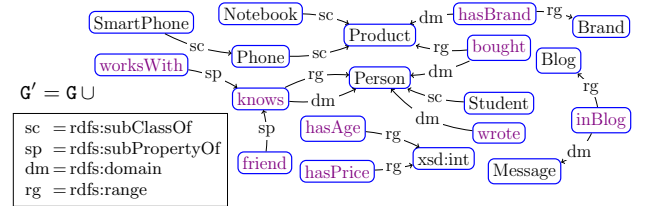


**Figure 2: Running example: RDF graph.**



**Figure 3: Running example: RDF Schema triples.**

EXAMPLE 1. (RDF GRAPH) *We consider an RDF graph comprising information about users and products. Figure 2 shows some of the triples (top) and depicts the whole dataset using its graph notation (bottom). The RDF graph features a resource* user$_1$ *whose name is "Bill" and whose age is "28". Bill works with* user$_2$ *and is a friend of* user$_3$. *He is an active contributor to two blogs, one shared with his co-worker* user$_2$. *Bill also bought a* SmartPhone *and rated it online etc.*

A valuable feature of RDF is RDF Schema (RDFS) that allows enhancing the descriptions in RDF graphs. RDFS triples declare *semantic constraints* between the classes and the properties used in those graphs. Figure 1 (bottom) shows the allowed constraints and how to express them; *domain* and *range* denote respectively the first and second attribute of every property. The RDFS constraints (Figure 1) are interpreted under the open-world assumption (OWA) [7]. For instance, given two relations $R_1, R_2$, the OWA interpretation of the constraint $R_1 \subseteq R_2$ is: any tuple $t$ in the relation $R_1$ *is considered as being also in the relation* $R_2$ (the inclusion constraint *propagates* $t$ *to* $R_2$).

EXAMPLE 2. (RDF SCHEMA) *Consider next to the graph* G *from Figure 2, the schema depicted in Figure 3. This schema expresses semantic (or ontological) constraints like a* Phone *is a* Product, *a* SmartPhone *is a* Phone, *a* Student *is a* Person, *the domain and range of* knows *is* Person, *that working with someone is one way of* knowing *her etc.*

**RDF entailment**. Our discussion on constraint interpretation above illustrated an important RDF feature: *implicit*

468

*triples*, considered part of the RDF graph even though they are not explicitly present in it. An example is product$_1$ rdf:type Phone, which is implicit in the graph G$'$ of Figure 3. W3C names *RDF entailment* the mechanism through which, based on a set of explicit triples and some *entailment rules*, implicit RDF triples are derived. We denote by $\vdash^i_{\text{RDF}}$ *immediate entailment*, i.e., the process of deriving new triples through a single application of an entailment rule. More generally, a triple s p o is entailed by a graph G, denoted G $\vdash_{\text{RDF}}$ s p o, if and only if there is a sequence of applications of immediate entailment rules that leads from G to s p o (where at each step of the entailment sequence, the triples previously entailed are also taken into account).

**Saturation**. The immediate entailment rules allow defining the finite *saturation* (a.k.a. closure) of an RDF graph G, which is the RDF graph G$^\infty$ defined as the fix-point obtained by repeatedly applying $\vdash^i_{\text{RDF}}$ on G.

The saturation of an RDF graph is unique (up to blank node renaming), and does not contain implicit triples (they have all been made explicit by saturation). An obvious connection holds between the triples entailed by a graph G and its saturation: G $\vdash_{\text{RDF}}$ s p o if and only if s p o $\in$ G$^\infty$.

RDF entailment is part of the RDF standard itself; in particular, *the answers of a query posed on G must take into account all triples in* G$^\infty$, since *the semantics of an RDF graph is its saturation*. In Sesame [39], Jena [37], OWLIM [38] etc., RDF entailment is supported through *saturation*.

## 3. BGP QUERIES

We consider the well-known subset of SPARQL consisting of (unions of) *basic graph pattern* (BGP) queries, also known as SPARQL conjunctive queries. A BGP is a set of *triple patterns*, or triples in short. Each triple has a subject, property and object, some of which can be variables.

**Notation**. In the following we will use the conjunctive query notation $q(\bar{x})$:- $t_1, \ldots, t_\alpha$, where $\{t_1, \ldots, t_\alpha\}$ is a BGP; the query head variables $\bar{x}$ are called *distinguished variables*, and are a subset of the variables occurring in $t_1, \ldots, t_\alpha$; for boolean queries $\bar{x}$ is empty. The head of $q$ denoted $\text{head}(q)$ is $q(\bar{x})$, and the body of $q$ denoted $\text{body}(q)$ is $t_1, \ldots, t_\alpha$. We use $x$, $y$, and $z$ (possibly with subscripts) to denote variables in queries. We denote by $\text{VarBl}(q)$ the set of variables *and* blank nodes occurring in the query $q$.

**BGP query graph**. For our purposes, it is useful to view each triple atom in the body of a BGP query as a *generalized RDF triple*, where *variables* may appear in any of the subject, predicate and object positions. This leads to a *graph notation for BGP queries*, which can be seen as a corresponding generalization of our RDF graph representation (Definition 1). For instance, the body of the query:

$$q(x, y, z)\text{:- } x \text{ hasName } y, x\ z \text{ product}_1$$

is represented by the graph:
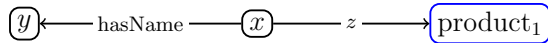


**Query evaluation**. Given a query $q$ and an RDF graph G, the *evaluation of q against* G is:

$$q(\text{G}) = \{\bar{x}_\mu \mid \mu : \text{VarBl}(q) \to \text{Val}(\text{G}) \text{ is a total assignment}$$
$$\text{such that } t_1^\mu \in \text{G}, t_2^\mu \in \text{G}, \ldots, t_\alpha^\mu \in \text{G}\}$$

where we denote by $t^\mu$ the result of replacing every occurrence of a variable or blank node $e \in \text{VarBl}(q)$ in the triple $t$, by the value $\mu(e) \in \text{Val}(\text{G})$.

Notice that evaluation *treats the blank nodes in a query exactly as it treats non-distinguished variables* [8]. Thus, in the sequel, without loss of generality, we consider queries where all blank nodes have been replaced by distinct (new) non-distinguished variable symbols.

**Query answering**. The evaluation of $q$ against G uses only G's explicit triples, thus may lead to an incomplete answer set. The (complete) *answer set* of $q$ against G is obtained by the evaluation of $q$ against G$^\infty$, denoted by $q(\text{G}^\infty)$.

EXAMPLE 3. (BGP QUERY ANSWERING) *The following query on* G$'$ *(Figure 3) asks for the names of those having bought a product related to* Phone*:*

$$q(x) \text{ :- } y_1 \text{ hasName } x, y_1 \text{ bought } y_2, y_2\ y_3 \text{ Phone}$$

*Here,* $q(\text{G}'^\infty) = \{\langle\text{"Bill"}\rangle, \langle\text{"William"}\rangle\}$.
*The answer results from* G$'$ $\vdash_{\text{RDF}}$ product$_1$ rdf:type Phone *and the assignments:*
$\mu_1 = \{y_1 \to \text{user}_1, x \to \text{Bill}, y_2 \to \text{product}_1, y_3 \to \text{rdf:type}\}$ *and*
$\mu_2 = \{y_1 \to \text{user}_1, x \to \text{William}, y_2 \to \text{product}_1, y_3 \to \text{rdf:type}\}$.
*Note that evaluating q against* G$'$ *leads to the incomplete (empty) answer set* $q(\text{G}') = \{\langle\rangle\}$.

**BGP queries for data analysis**. Data analysis typically allows investigating particular sets of facts according to relevant criteria (a.k.a. *dimensions*) and measurable or countable attributes (a.k.a. *measures*) [23]. In this work, *rooted* BGP queries play a central role as they are used to specify the set of facts to analyze, as well as the dimensions and the measures to be used (Section 4.2).

DEFINITION 2. (ROOTED QUERY) *Let $q$ be a BGP query,* G $= \langle \mathcal{N}, \mathcal{E}, \lambda \rangle$ *its graph and $n \in \mathcal{N}$ a node whose label is a variable in $q$. The query $q$ is* rooted *in $n$ iff* G *is a connected graph and any other node $n' \in \mathcal{N}$ is reachable from $n$ following the directed edges in $\mathcal{E}$.*

EXAMPLE 4. (ROOTED QUERY) *The query $q$ described below is a rooted BGP query, with $x_1$ as root node.*

$$q(x_1, x_2, x_3) \text{ :- } x_1 \text{ knows } x_2, x_1 \text{ hasName } y_1,$$
$$x_1 \text{ wrote } y_2, y_2 \text{ inBlog } x_3$$

*The query's graph representation below shows that every node is reachable from the root $x_1$.*
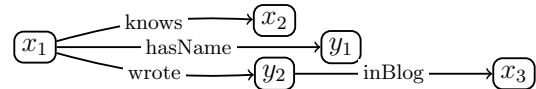


Next, we introduce the concept of *join* query, which joins BGP queries on their *distinguished variables* and projects out some of these variables. Join queries will be used when defining data warehouse analyses.

DEFINITION 3. (JOIN QUERY) *Let $q_1, \ldots, q_n$ be BGP queries whose non-distinguished variables are pairwise disjoint. We say $q(\bar{x})$:- $q_1(\bar{x}_1) \wedge \cdots \wedge q_n(\bar{x}_n)$, where $\bar{x} \subseteq \bar{x}_1 \cup \cdots \cup \bar{x}_n$, is a* join query $q$ *of $q_1, \ldots, q_n$. The answer set to $q(\bar{x})$ is defined to be that of the BGP query $q^{\bowtie}$:*

$$q^{\bowtie}(\bar{x})\text{:- } \text{body}(q_1(\bar{x}_1)), \cdots, \text{body}(q_n(\bar{x}_n))$$

In the above, $q_1, q_2, \ldots, q_n$ do not share *non-distinguished variables* (variables not present in the query head). This assumption is made *without loss of generality*, as one can easily rename non-distinguished variables in $q_1, q_2, \ldots, q_n$ in order to meet the condition. In the sequel, we assume such renaming has been applied in join queries.

EXAMPLE 5. (JOIN QUERY) *Consider the BGP queries $q_1$, asking for the users having bought a product and their age, and $q_2$, asking for users having posted in some blog:*

$$q_1(x_1, x_2) :- \ x_1 \text{ bought } y_1, x_1 \text{ hasAge } x_2$$
$$q_2(x_1, x_3) :- \ x_1 \text{ wrote } y_2, y_2 \text{ inBlog } x_3$$

*The join query $q_{1,2}(x_1, x_2) :- q_1(x_1, x_2) \wedge q_2(x_1, x_3)$ asks for the users and their ages, for all the users having posted in a blog and having bought a product, i.e.,*

$$q_{1,2}^{\bowtie}(x_1, x_2) :- \ x_1 \text{ bought } y_1, x_1 \text{ hasAge } x_2,$$
$$x_1 \text{ wrote } y_2, y_2 \text{ inBlog } x_3$$

*Other join queries can be obtained from $q_1$ and $q_2$ by returning a different subset of the head variables $x_1, x_2, x_3$, and/or by changing their order in the query head etc.*

# 4. RDF GRAPH ANALYSIS

We define here the basic ingredients of our approach for analyzing RDF graphs. An *analytical schema* is the lens through which we analyze an RDF graph, as we explain in Section 4.1. An analytical schema *instance* is analyzed with *analytical queries*, introduced in Section 4.2, modeling the chosen criteria (a.k.a. dimensions) and measurable or countable attributes (a.k.a. measures) of the analysis.

## 4.1 Analytical schema and instance

We model a schema for RDF graph analysis, called *analytical schema*, as a labeled directed graph.

From a classical data warehouse analytics perspective, *each node of our analytical schema represents a set of facts* that may be analyzed. Moreover, *the facts represented by an analytical schema node can be analyzed using (as either dimensions or measures) the schema nodes reachable from that node.* This makes our analytical schema model much more general than the traditional DW setting where facts (at the center of a star or snowflake schema) are analyzed according to a fixed set of dimensions and of measures.

From a Semantic Web perspective, an analytical schema node corresponds to an RDF class, while an analytical schema edge connecting two nodes corresponds to an RDF property. The instances of these classes and properties, modeling the DW contents to be further analyzed, are *intensionally* defined in the schema, following the well-know "Global As View" (GAV) approach for data integration [19].

DEFINITION 4. (ANALYTICAL SCHEMA) *An analytical schema (AnS) is a labeled directed graph $\mathcal{S} = \langle \mathcal{N}, \mathcal{E}, \lambda, \delta \rangle$ in which:*

- $\mathcal{N}$ *is the set of nodes;*
- $\mathcal{E} \subseteq \mathcal{N} \times \mathcal{N}$ *is the set of directed edges;*
- $\lambda : \mathcal{N} \cup \mathcal{E} \to U$ *is an injective labeling function, mapping nodes and edges to URIs;*
- $\delta : \mathcal{N} \cup \mathcal{E} \to \mathcal{Q}$ *is a function assigning to each node $n \in \mathcal{N}$ a unary BGP query $\delta(n) = q(x)$, and to every edge $e \in \mathcal{E}$ a binary BGP query $\delta(e) = q(x, y)$.*

**Notation**. We use $n$ and $e$ respectively (possibly with subscripts) to denote *AnS* nodes and edges. To emphasize that an edge connects two particular nodes we will place the nodes in subscript, e.g., $e_{n_1 \to n_2}$.

For simplicity, we assume that through $\lambda$, each node in the *AnS* defines a *new* class (not present in the original graph G), while each edge defines a new property[1]. Observe that

---

[1] In practice, nothing prevents $\lambda$ from returning URIs of class/properties from G and/or the RDF model, e.g., rdf:type etc.
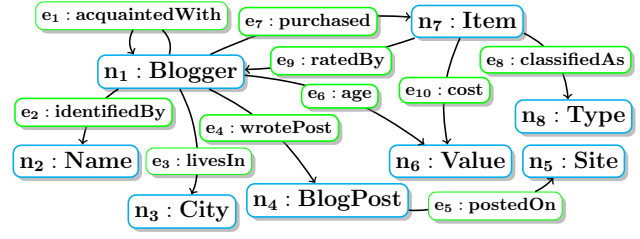
**Figure 4: Sample Analytical Schema ($AnS$).**

| node | $\lambda(n)$ | $\delta(n)$ |
|------|------|------|
| $n_1$ | Blogger | $q(x):- x$ rdf:type Person, $x$ wrote $y, y$ inBlog $z$ |
| $n_2$ | Name | $q(x):- y$ hasName $x$ |
| $n_3$ | City | $q(x):- y$ inCity $x$ |
| $n_4$ | BlogPost | $q(x):- x$ rdf:type Message, $x$ inBlog $z, z$ rdf:type Blog |
| $n_5$ | Site | $q(x):- y$ inBlog $x, x$ rdf:type Blog |
| $n_6$ | Value | $q(x):- z$ rdfs:range xsd:int, $y$ $z$ $x$ |
| $n_7$ | Item | $q(x):- x$ rdf:type $y, y$ rdfs:subClassOf Product |
| $n_8$ | Type | $q(x):- x$ rdfs:subClassOf Product |

| edge | $\lambda(e)$ | $\delta(e)$ |
|------|------|------|
| $e_1$ | acquaintedWith | $q(x, y):- z$ rdfs:subPropertyOf knows, $x$ $z$ $y$ |
| $e_2$ | identifiedBy | $q(x, y):- x$ hasName $y$ |
| $e_3$ | livesIn | $q(x, y):- x$ hasCity $y$ |
| $e_4$ | wrotePost | $q(x, y):- x$ wrote $y, y$ rdf:type Message |
| $e_5$ | postedOn | $q(x, y):- x$ rdf:type Message, $x$ inBlog $y$ |
| $e_6$ | age | $q(x, y):- x$ rdf:type Person, $x$ hasAge $y$ |
| $e_7$ | purchased | $q(x, y):- x$ bought $y$ |
| $e_8$ | classifiedAs | $q(x, y):- x$ rdf:type Product, $x$ rdf:type $y$ |
| $e_9$ | ratedBy | $q(x, y):- y$ gave $z, z$ rdf:type Rating, $z$ on $x, x$ rdf:type Product |
| $e_{10}$ | cost | $q(x, y):- x$ hasPrice $y$ |

**Table 1: Labels and queries of some nodes and edges of the analytical schema ($AnS$) shown in Figure 4.**

using $\delta$ we define a GAV view for each node and edge in the analytical schema. Just as an analytical schema defines (and delimits) the data available to the analyst in a typical relational DW scenario, in our framework, *the classes and properties modeled by an AnS (defined using $\delta$ and labeled by $\lambda$) are the only ones visible to further RDF analytics*, that is: analytical queries will be formulated against the *AnS* and not against the base data (as Section 4.2 will show). Example 6 introduces an *AnS* for the RDF graph in Figure 3.

EXAMPLE 6. (ANALYTICAL SCHEMA) *Figure 4 depicts an AnS for analyzing bloggers and items. The node and edge labels appear in the figure, while the BGP queries defining these nodes and edges are provided in Table 1. In Figure 4 a blogger ($n_1$) may have written posts ($e_4$) which appear on some site ($e_5$). A person may also have purchased items ($e_7$) which can be rated ($e_9$). The semantic of the remaining AnS nodes and edges can be easily inferred.*

The nodes and edges of an analytical schema define the perspective (or *lens*) through which to analyze an RDF dataset. This is formalized as follows:

DEFINITION 5. (INSTANCE OF AN $AnS$) *Let $\mathcal{S} = \langle \mathcal{N}, \mathcal{E}, \lambda, \delta \rangle$ be an analytical schema and G an RDF graph. The instance of $\mathcal{S}$ w.r.t. G is the RDF graph $\mathcal{I}(\mathcal{S}, \texttt{G})$ defined as:*

$$\bigcup_{n \in \mathcal{N}} \{s \text{ rdf:type } \lambda(n) \mid s \in q(\texttt{G}^\infty) \wedge q = \delta(n)\}$$
$$\cup$$
$$\bigcup_{e \in \mathcal{E}} \{s \ \lambda(e) \ o \mid s, o \in q(\texttt{G}^\infty) \wedge q = \delta(e)\}.$$

From now on, we denote the instance of an *AnS* either $\mathcal{I}(\mathcal{S}, \texttt{G})$ or simply $\mathcal{I}$, when that does not lead to confusion.

EXAMPLE 7. (ANALYTICAL SCHEMA INSTANCE) *Below we show part of the instance of the analytical schema introduced in Example 6. We indicate at right of each triple the node (or edge) of the AnS which produced it.*

$$\mathcal{I}(\mathcal{S}, \mathtt{G}') = \begin{cases} \{\text{user}_1 \text{ rdf:type } \textbf{Blogger}, & n_1 \\ \text{user}_1 \textbf{ acquaintedWith } \text{user}_2, & e_1 \\ \text{user}_1 \textbf{ identifiedBy } \text{"Bill"}, & e_2 \\ \text{post}_1 \textbf{ postedOn } \text{blog}_1, & e_5 \\ \text{user}_1 \textbf{ age } \text{"28"}, & e_6 \\ \text{product}_1 \text{ rdf:type } \textbf{Item}, & n_7 \\ \text{SmartPhone rdf:type } \textbf{Type}, & n_8 \\ \text{product}_1 \textbf{ cost } \text{"400"}, \ldots\} & e_{10} \end{cases}$$

Central to our notion of RDF warehouse is *the disjunctive semantics of an AnS*, materialized by the two levels of union ($\cup$) in Definition 5. *Each node and each edge of an AnS populates $\mathcal{I}$ through an independent RDF query*, and the resulting triples are unioned to produce the *AnS* instance. Defining *AnS* nodes and edges independently of each other is crucial for allowing our warehouse to:

- be an actual *RDF graph* (in contrast to tabular data, possibly with many *null*s, which would result if we attempted to fit the RDF in a relational warehouse). This addresses the requirement ($i$) from our motivating scenario (Section 1). It also guarantees that the *AnS* instance can be *shared, linked, and published* according to the best current Semantic Web practices;

- directly benefit from the *semantic-aware SPARQL query answering* provided by SPARQL engines. This answers our semantic-awareness requirement ($iii$), and also ($iv$) (ability to *query the schema*, notoriously absent from relational DWs);

- provide *as many entry points for analysis as there are AnS nodes*, in line with the flexible, decentralized nature of RDF graph themselves (requirement ($ii$)). As a consequence (see below), aggregation queries are very flexible, e.g., they can aggregate one entity in relation with another (count restaurants at proximity of landmarks, requirement ($v$) in Section 1);

- *support AnS changes easily* (requirement ($ii$)) since nodes and/or edge definitions can be freely added to (removed from) the *AnS*, with no impact on the other node/edge definitions, or their instances.

As an illustration of our point on heterogeneity (($i$) above), consider the three users in the original graph $\mathtt{G}$ (Figure 2) and their properties: user$_1$, user$_2$ and user$_3$ are part of the Blogger class in our *AnS* instance $\mathcal{I}$ (through $n_1$'s query), although user$_2$ and user$_3$ lack a name. However, those user properties present in the original graph, are reflected by the *AnS* edges $e_3$, $e_4$ etc. Thus, RDF heterogeneity is accepted in the base data and present in the *AnS* instance.

**Defining analytical schemas**. As customary in data analysis/warehouse, analysts are in charge of defining the schema, with significant flexibility in our framework for doing so. Typically, schema definition starts with the choice of a few concepts of interest, to be turned into *AnS* nodes. These can come from the application, or be "suggested" based on the RDF data itself, e.g., *the most popular types in the dataset* (RDF classes together with the number of resources belonging to the class), which can be obtained with a simple SPARQL query; we have implemented this in the GUI of our tool [13]. Core concepts and edges may also be identified through RDF summarization as in e.g., [12]. Further, SPARQL queries can be asked to identify *the most frequent*

relationships to which the resources of an AnS node participate, or *chains of relationships connecting instances of two AnS nodes* etc. In this incremental fashion, the *AnS* can be "grown" from a few nodes to a graph capturing all information of interest; throughout the process, SPARQL queries can be leveraged to assist and guide *AnS* design.

Once the queries defining *AnS* nodes are known, the analyst may want to check that an edge is *actually connected to a node adjacent to the edge*, in the sense: some resources in the node extent also participate to the relationship defined by edge. Let $n_1, n_2 \in \mathcal{N}$ be *AnS* nodes and $e_{n_1 \to n_2} \in \mathcal{E}$ an edge between them. This condition can be easily checked through a SPARQL query ensuring that:

$$ans(\delta(n_1)) \cap \Pi_{\text{domain}}(ans(\delta(e_{n_1 \to n_2}))) \neq \emptyset$$

**Extensions**. An *AnS* uses unary and binary BGP queries (introduced in Section 3) to define its instance, as the union of all *AnS* node/class and edge/property instances. This can be extended straightforwardly to unary and binary (full) SPARQL queries (allowing disjunction, filter, regular expressions, etc.) in the setting of RDF analytics, and even to unary and binary queries from (a mix of) query languages (SQL, SPARQL, XQuery, etc.), in order to analyze data integrated from distributed heterogeneous sources.

## 4.2 Analytical queries

Data warehouse analysis summarizes facts according to relevant criteria into so-called *cubes*. Formally, a cube (or analytical query) analyzes facts characterized by some *dimensions*, using a *measure*. We consider a set of dimensions $d_1, d_2, \ldots, d_n$, such that each dimension $d_i$ may range over the value set $\{d_i^1, \ldots, d_i^{n_i}\}$; the Cartesian product of all dimensions $d_1 \times \cdots \times d_n$ defines a multidimensional space $\mathcal{M}$. To each tuple $t$ in this multidimensional space $\mathcal{M}$ corresponds a *subset* $\mathcal{F}_t$ of the analyzed facts, having for each dimension $d_{i,1 \leq i \leq n}$, the value of $t$ along $d_i$.

A *measure* is a set of values[2] characterizing each analyzed fact $f$. The facts in $\mathcal{F}_t$ are summarized by the *cube cell* $\mathcal{M}[t]$ by the result of an *aggregation* function $\oplus$ (e.g., count, sum, average, etc.) applied to the union of the measures of the $\mathcal{F}_t$ facts: $\mathcal{M}[t] = \oplus(\bigcup_{f \in \mathcal{F}_t} v_f)$.

An *analytical query* consists of two (rooted) queries and an aggregation function. The first query, known as a *classifier* in traditional data warehouse settings, defines the *dimensions* $d_1, d_2, \ldots, d_n$ according to which the facts matching the query root will be analyzed. The second query defines the *measure* according to which these facts will be summarized. Finally, the aggregation function is used for *summarizing* the analyzed facts.

To formalize the connection between an analytical query and the *AnS* on which it is asked, we introduce a useful notion:

DEFINITION 6. (BGP QUERY TO *AnS* HOMOMORPHISM) *Let $q$ be a BGP query whose labeled directed graph is $G_q = \langle \mathcal{N}, \mathcal{E}, \lambda \rangle$, and $\mathcal{S} = \langle \mathcal{N}', \mathcal{E}', \lambda', \delta' \rangle$ be an AnS. An homomorphism from $q$ to $\mathcal{S}$ is a graph homomorphism $h : G_q \to \mathcal{S}$, such that:*

- *for every $n \in \mathcal{N}$, $\lambda(n) = \lambda'(h(n))$ or $\lambda(n)$ is a variable;*
- *for every $e_{n \to n'} \in \mathcal{E}$: (i) $e_{h(n) \to h(n')} \in \mathcal{E}'$ and (ii) $\lambda(e_{n \to n'}) = \lambda'(e_{h(n) \to h(n')})$ or $\lambda(e_{n \to n'})$ is a variable;*

---

[2]It is a set rather than a single value, due to the structural heterogeneity of the *AnS* instance, which is an RDF graph itself: each fact may have zero, one, or more values for a given measure.

- *for every* $e_1, e_2 \in \mathcal{E}$, *if* $\lambda(e_1) = \lambda(e_2)$ *is a variable,*
  *then* $h(e_1) = h(e_2)$;
- *for* $n \in \mathcal{N}$ *and* $e \in \mathcal{E}$, $\lambda(n) \neq \lambda(e)$.

The above homomorphism is defined as a correspondence from the query to the *AnS* graph structure, which preserves labels when they are not variables (first two items), and maps all the occurrences of a same variable *labeling different query edges* to the same label value (third item). Observe that a similar condition referring to occurrences of a same variable *labeling different query nodes* is not needed, since by definition, all occurrences of a variable in a query are mapped to the same node in the query's graph representation. The last item (independent of $h$) follows from the fact that the labeling function of an *AnS* is injective. Thus, a query with a same label for a node and an edge cannot have an homomorphism with an *AnS*.

We are now ready to introduce our analytical queries. In keeping with the spirit (but not the restrictions!) of classical RDWs [22, 23], a *classifier* defines the level of data aggregation while a *measure* allows obtaining values to be aggregated using *aggregation functions*.

DEFINITION 7. (ANALYTICAL QUERY) *Given an analytical schema* $\mathcal{S} = \langle \mathcal{N}, \mathcal{E}, \lambda, \delta \rangle$, *an* analytical query *(AnQ) rooted in the node* $r \in \mathcal{N}$ *is a triple:*

$$Q = \langle c(x, d_1, \ldots, d_n), m(x, v), \oplus \rangle$$

*where:*
- $c(x, d_1, \ldots, d_n)$ *is a query rooted in the node* $r_c$ *of its graph* $G_c$, *with* $\lambda(r_c) = x$. *This query is called the* classifier *of* $x$ *w.r.t. the* $n$ *dimensions* $d_1, \ldots, d_n$.
- $m(x, v)$ *is a query rooted in the node* $r_m$ *of its graph* $G_m$, *with* $\lambda(r_m) = x$. *This query is called the* measure *of* $x$.
- $\oplus$ *is a* function *computing a value (a literal) from an input set of values. This function is called the* aggregator *for the measure of* $x$ *w.r.t. its classifier.*
- *For every homomorphism* $h_c$ *from the classifier to* $\mathcal{S}$ *and every homomorphism* $h_m$ *from the measure to* $\mathcal{S}$, $h_c(r_c) = h_m(r_m) = r$ *holds.*

The last item above guarantees the "well-formedness" of the analytical query, that is: the facts for which we aggregate the measure are indeed those classified along the desired dimensions. From a practical viewpoint, this condition can be easily and naturally guaranteed by giving explicitly in the classifier and the measure either the type of the facts to analyze, using $x$ rdf:type $\lambda(r)$, or a property describing those facts, using $x$ $\lambda(e_{r \to n})$ o with $e_{r \to n} \in \mathcal{E}$. As a result, since the labels are unique in an *AnS* (its labeling function is injective), every homomorphism from the classifier (respectively the measure) to the *AnS* does map the query's root node labeled with $x$ to the *AnS*'s node $r$.

EXAMPLE 8. (ANALYTICAL QUERY) *The query below asks for the number of sites where each blogger posts, classified by the blogger's age and city:*

$$\langle c(x, y_1, y_2), m(x, z), count \rangle$$

*where the classifier and measure queries are defined by:*
$$c(x, y_1, y_2) \text{:- } x \text{ age } y_1, x \text{ livesIn } y_2$$
$$m(x, z) \text{:- } x \text{ wrotePost } y, y \text{ postedOn } z$$

The semantics of an analytical query is:

DEFINITION 8. (ANSWER SET OF AN ANQ) *Let* $\mathcal{I}$ *be the instance of an AnS with respect to some RDF graph. Let* $Q = \langle c(x, d_1, \ldots, d_n), m(x, v), \oplus \rangle$ *be an AnQ against* $\mathcal{I}$. *The* answer set *of* $Q$ *against* $\mathcal{I}$, *denoted* $ans(Q, \mathcal{I})$, *is:*

$$ans(Q, \mathcal{I}) = \{\langle d_1^j, \ldots, d_n^j, \oplus(q^j(\mathcal{I})) \rangle \mid \langle x^j, d_1^j, \ldots, d_n^j \rangle \in c(\mathcal{I})$$
$$\text{and } q^j \text{ is defined as } q^j(v) \text{:- } m(x^j, v)\}$$

*assuming that each value returned by* $q^j(\mathcal{I})$ *is of (or can be converted by the SPARQL rules [35] to) the input type of the aggregator* $\oplus$. *Otherwise, the answer set is undefined.*

In other words, the analytical query returns each tuple of dimension values found in the answer of the classifier query, together with the aggregated result of the measure query. The answer set of an *AnQ* can thus be represented as a cube of $n$ dimensions, holding in each cube cell the corresponding aggregate measure. In the following, we focus on analytical queries whose answer sets are not undefined.

EXAMPLE 9. (ANALYTICAL QUERY ANSWER) *Consider the query in Example 8, over the AnS in Figure 4. Some triples from the instance of this analytical schema were shown in Example 7. The classifier query' answer set is:*

$$\{\langle \text{user}_1, 28, \text{``Madrid''} \rangle, \langle \text{user}_3, 35, \text{``NY''} \rangle\}$$
*while that of the measure query is:*
$$\{\langle \text{user}_1, \text{blog}_1 \rangle, \langle \text{user}_1, \text{blog}_2 \rangle, \langle \text{user}_2, \text{blog}_2 \rangle, \langle \text{user}_3, \text{blog}_2 \rangle\}$$
*Aggregating the blogs among the classification dimensions leads to the AnQ answer:*

$$\{\langle 28, \text{``Madrid''}, 2 \rangle, \langle 35, \text{``NY''}, 1 \rangle\}$$

In this work, for the sake of simplicity, we assume that an analytical query has only one measure. However, this can be easily relaxed, by introducing a set of measure queries with an associated set of aggregation functions.

# 5. ANALYTICAL QUERY ANSWERING

We now consider practical strategies for *AnQ* answering.

**The *AnS* materialization approach**. The simplest method consists of materializing the instance of the *AnS* (Definition 5) and storing it within an RDF data management system (or RDF-DM, for short); recall that the *AnS* instance is an RDF graph itself defined using GAV views. Then, to answer an *AnQ*, one can use the RDF-DM to process the classifier and measure queries, and the final aggregation. While effective, this solution has the drawback of storing the whole *AnS* instance; moreover, this instance may need maintenance when the analyzed RDF graph changes.

**The *AnQ* reformulation approach**. To avoid materializing and maintaining the *AnS* instance, we consider an alternative solution. The idea is to rewrite the *AnQ* using the GAV views of the *AnS* definition, so that evaluating the reformulated query returns exactly the same answer as if materialization was used. Using query rewriting, one can store the original RDF graph into an RDF-DM, and use this RDF-DM to answer the reformulated query.

Our reformulation technique below translates standard query rewriting using GAV views [19] to our RDF analytical setting.

DEFINITION 9. (*AnS*-REFORMULATION OF A QUERY) *Given an analytical schema* $\mathcal{S} = \langle \mathcal{N}, \mathcal{E}, \lambda, \delta \rangle$, *a BGP query* $q(\bar{x})$ :- $t_1, \ldots, t_m$ *whose graph is* $G_q = \langle \mathcal{N}', \mathcal{E}', \lambda' \rangle$, *and the non-empty set* $\mathcal{H}$ *of all the homomorphisms from* $q$ *to* $\mathcal{S}$, *the* reformulation of $q$ *w.r.t.* $\mathcal{S}$ *is the union of join queries* $q_{\mathcal{S}}^{\bowtie} = \bigcup_{h \in \mathcal{H}} q_h^{\bowtie}(\bar{x})$ :- $\bigwedge_{i=1}^m q_i(\bar{x}_i)$ *such that:*
- *for each triple* $t_i \in q$ *of the form* s rdf:type $\lambda'(n_i)$, $q_i(\bar{x}_i)$ *in* $q_h^{\bowtie}$ *is defined as* $q_i = \delta(h(n_i))$ *and* $\bar{x}_i = $ s;
- *for each triple* $t_i \in q$ *of the form* s $\lambda'(e_i)$ o, $q_i(\bar{x}_i)$ *in* $q_h^{\bowtie}$ *is defined as* $q_i = \delta(h(e_i))$ *and* $\bar{x}_i = $ s, o.

This definition states that for a BGP query stated against an $AnS$, the reformulated query amounts to translating all its possible interpretations w.r.t. the $AnS$ (modeled by all the homomorphisms from the query to the $AnS$) into a union of join queries modeling them. The important point is that *these join queries are defined onto the RDF graph* over which the $AnS$ is wrapped.

EXAMPLE 10. ($AnS$-REFORMULATION OF A QUERY) *Let $q(x, y_1)$ be a BGP query referring to the AnS in Figure 4.*

$$q(x, y_1) :\text{-} \ x \ \text{rdf:type Blogger}, x \ \text{acquaintedWith} \ y_1$$

*The first atom $x$ rdf:type Blogger in $q$ is of the form s rdf:type $\lambda(n_1)$, for the node $n_1$. Consequently, $q_{\mathcal{S}}^{\bowtie}$ contains as a conjunct the query:*

$$q(x) :\text{-} \ x \ \text{rdf:type Person}, x \ \text{wrote} \ y, y \ \text{inBlog} \ z$$

*obtained from $\delta(n_1)$ in Table 1.*

*The second atom in $q$, $x$ acquaintedWith $y$ is of the form s $\lambda(e_1)$ o for the edge $e_1$ in Figure 4, while the query defining $e_1$ is: $q(x, y) :\text{-} \ z \ \text{rdfs:subPropertyOf knows}, x \ z \ y$. As a result, $q_{\mathcal{S}}^{\bowtie}$ contains the conjunct:*

$$q(x, y_1) :\text{-} \ z_1 \ \text{rdfs:subPropertyOf knows}, x \ z_1 \ y_1$$

*Thus, the reformulated query amounts to:*

$$q_{\mathcal{S}}^{\bowtie}(x, y_1) :\text{-} \ x \ \text{rdf:type Person}, x \ \text{wrote} \ y, y \ \text{inBlog} \ z,$$
$$z_1 \ \text{rdfs:subPropertyOf knows}, x \ z_1 \ y_1$$

*which can be evaluated directly on the graph G in Figure 2.*

Theorem 1 states how BGP query reformulation w.r.t. an $AnS$ can be used to answer analytical queries correctly.

THEOREM 1. (REFORMULATION-BASED ANSWERING) *Let $\mathcal{S}$ be an analytical schema, whose instance $\mathcal{I}$ is defined w.r.t. an RDF graph G. Let $Q = \langle c(x, d_1, \ldots, d_n), m(x, v), \oplus \rangle$ be an analytical query against $\mathcal{S}$, $c_{\mathcal{S}}^{\bowtie}$ be the reformulation of $Q$'s classifier query against $\mathcal{S}$, and $m_{\mathcal{S}}^{\bowtie}$ be the reformulation of $Q$'s measure query against $\mathcal{S}$. We have:*

$$ans(Q, \mathcal{I}) = \{ \langle d_1^j, \ldots, d_n^j, \oplus(q^j(\text{G}^\infty)) \rangle \mid \langle x^j, d_1^j, \ldots, d_n^j \rangle \in c_{\mathcal{S}}^{\bowtie}(\text{G}^\infty)$$
$$\text{and } q^j \text{ is defined as } q^j(v) :\text{-} \ m_{\mathcal{S}}^{\bowtie}(x^j, v) \}$$

*assuming that each value returned by $q^j(\text{G}^\infty)$ is of (or can be converted by the SPARQL rules [35] to) the input type of the aggregator $\oplus$. Otherwise, the answer set is undefined.*

The theorem states that in order to answer $Q$ on $\mathcal{I}$, one first reformulates $Q$'s classifier into $c_{\mathcal{S}}^{\bowtie}$ and answers it *directly against* G (not against $\mathcal{I}$ as in Definition 8): this is how reformulation avoids materializing $\mathcal{I}$. Then, for each tuple $\langle x^j, d_1^j, \ldots, d_n^j \rangle$ returned by the classifier, the following steps are applied: instantiate the reformulated measure query $m_{\mathcal{S}}^{\bowtie}$ with the fact $x^j$, leading to the query $q^j$; answer the latter against G; finally, aggregate its results through $\oplus$. The proof follows directly, by two-way inclusion.

The trade-offs between materialization and reformulation have been thoroughly analyzed in the literature [22]; we leave the choice to the RDF warehouse administrator.

# 6. OLAP RDF ANALYTICS

On-Line Analytical Processing (OLAP) [3] technologies enhance the abilities of data warehouses (so far, mostly relational) to answer multi-dimensional analytical queries.

The analytical model we introduced is specifically designed for graph-structured, heterogeneous RDF data. In this section, we demonstrate that our model is able to express RDF-specific counterparts of all the traditional OLAP concepts and tools known from the relational DW setting.

Typical OLAP operations allow transforming a cube into another. In our framework, a cube corresponds to an $AnQ$; for instance, the query in Example 8 models a bi-dimensional cube on the warehouse related to our sample $AnS$ in Figure 4. Thus, we model traditional OLAP operations on cubes as $AnQ$ rewritings, or more specifically, rewritings of *extended AnQ*s which we introduce below:

DEFINITION 10. (EXTENDED $AnQ$) *As in Definition 7, let $\mathcal{S}$ be an AnS, and $d_1, \ldots, d_n$ be a set of dimensions, each ranging over a non-empty finite set $V_{i,1 \leq i \leq n}$. Let $\Sigma$ be a total function over $\{d_1, \ldots, d_n\}$ associating to each $d_i$, either $\{d_i\}$ or a non-empty subset of $V_i$. An extended analytical query $Q$ is defined by a triple:*

$$Q :\text{-} \ \langle c_\Sigma(x, d_1, \ldots, d_n), m(x, v), \oplus \rangle$$

*where (as in Definition 7) $c$ is a classifier and $m$ a measure query over $\mathcal{S}$, $\oplus$ is an aggregation operator, and moreover:*

$$c_\Sigma(x, d_1, \ldots, d_n) =$$
$$\bigcup\nolimits_{(\chi_1, \ldots, \chi_n) \in \Sigma(d_1) \times \ldots \times \Sigma(d_n)} c(x, \chi_1, \ldots, \chi_n)$$

In the above, the extended classifier $c_\Sigma(x, d_1, \ldots, d_n)$ is the set of all possible classifiers obtained by *substituting each dimension variable $d_i$ with a value in $\Sigma(d_i)$*. The function $\Sigma$ is introduced to constrain some classifier dimensions, i.e., it plays the role of a filter-clause restricting the classifier result. The *semantics of an extended analytical query* is easily derived from the semantics of a standard $AnQ$ (Definition 8) by replacing the tuples from $c(\mathcal{I})$ with tuples from $c_\Sigma(\mathcal{I})$. In other words, an extended analytical query can be seen as a union of a set of standard $AnQ$s, one for each combination of values in $\Sigma(d_1), \ldots, \Sigma(d_n)$. Conversely, an analytical query corresponds to an extended analytical query where $\Sigma$ only contains pairs of the form $(d_i, \{d_i\})$.

We can now define the classical *slice* and *dice* OLAP operations in our framework:

**Slice.** Given an extended query $Q = \langle c_\Sigma(x, d_1, \ldots, d_n), m(x, v), \oplus \rangle$, a slice operation over a dimension $d_i$ with value $v_i$ returns the extended query $\langle c_{\Sigma'}(x, d_1, \ldots, d_n), m(x, v), \oplus \rangle$, where $\Sigma' = (\Sigma \setminus \{ (d_i, \Sigma(d_i)) \}) \cup \{ (d_i, \{v_i\}) \}$.

The intuition is that slicing binds an aggregation dimension to a single value.

EXAMPLE 11. (SLICE) *Let $Q$ be the extended query corresponding to the query-cube defined in Example 8, that is: $\langle c_\Sigma(x, y_1, y_2), m(x, z), count \rangle$, $\Sigma = \{ (y_1, \{y_1\}), (y_2, \{y_2\}) \}$ (the classifier and measure are as in Example 8). A slice operation on the age dimension $y_1$ with value 35 results in replacing the extended classifier of $Q$ with $c_{\Sigma'}(x, y_1, y_2) = \{c(x, 35, y_2)\}$ where $\Sigma' = \Sigma \setminus \{ (y_1, \{y_1\}) \} \cup \{ (y_1, \{35\}) \}$.*

**Dice.** Similarly, a dice operation on $Q$ over dimensions $\{d_{i_1}, \ldots, d_{i_k}\}$ and corresponding sets of values $\{S_{i_1}, \ldots, S_{i_k}\}$, returns the query $\langle c_{\Sigma'}(x, d_1, \ldots, d_n), m(x, v), \oplus \rangle$, where $\Sigma' = (\Sigma \setminus \bigcup_{j=i_1}^{i_k} \{ (d_j, \Sigma(d_j)) \}) \cup \bigcup_{j=i_1}^{i_k} \{ (d_j, S_j) \}$.

Intuitively, dicing forces several aggregation dimensions to take values from specific sets.

EXAMPLE 12. (DICE) *Consider again the initial cube $Q$ from Example 8 and a dice operation on both age and location dimensions with values $\{28\}$ for $y_1$ and $\{Madrid, Kyoto\}$ for $y_2$. The dice operation replaces the extended classifier of $Q$ with $c_{\Sigma'}(x, y_1, y_2) = \{c(x, 28, \text{"Madrid"}), c(x, 28, \text{"Kyoto"})\}$ where $\Sigma' = \Sigma \setminus \{ (y_1, \{y_1\}), (y_2, \{y_2\}) \} \cup \{ (y_1, \{28\}), (y_2, \{\text{"Madrid"}, \text{"Kyoto"}\}) \}$.*

**Drill-in and drill-out**. These operations consist of adding and removing a dimension to the classifier, respectively. Rewritings for drill operations can be easily formalized. Due to space limitations we omit the details, and instead exemplify below a drill-in example.

EXAMPLE 13. (DRILL-IN) *Consider the cube Q from Example 8, and a drill-in on the age dimension. The drill-in rewriting produces the query* $Q = \langle c'_{\Sigma'}(x, y_2), m(x, z), count \rangle$ *with* $\Sigma' = \{ (y_2, \{y_2\}) \}$ *and* $c'(x, y_2) = x$ livesIn $y_2$.

**Dimension hierarchies**. Typical relational warehousing scenarios feature hierarchical dimensions, e.g., a value of the *country* dimension corresponds to several *regions*, each of which contains many *cities* etc. Such hierarchies were not considered in our framework thus far[3].

To capture hierarchical dimensions, we introduce dedicated built-in *properties* to model the **nextLevel** relationship among parent-child dimensions in a hierarchy. For illustration, consider the addition of a new State node and a new **nextLevel** edge to the *AnS* in Figure 4. Below, only part of that *AnS* is shown, highlighting the new nodes and edges with dashed lines:
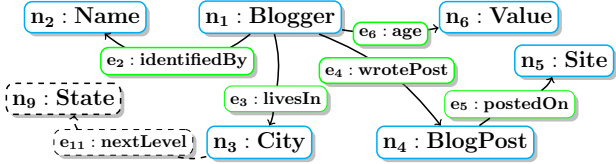


In a similar fashion one could use the **nextLevel** property to support *hierarchies among edges*. For instance, relationships such as *isFriendsWith* and *isCoworkerOf* can be rolled up into a more general relationship *knows* etc.

Based on dimension hierarchies, *roll-up/drill-down* operations correspond to *adding to/removing from* the classifier, triple atoms navigating such **nextLevel** edges.

EXAMPLE 14. (ROLL-UP) *Recall the query in Example 8. A roll-up along the* City *dimension to the* State *level yields* $\langle c'_{\Sigma'}(x, y_1, y_3), m(x, z), count \rangle$, *where:*

$c'_{\Sigma'}(x, y_1, y_3) :\text{-} x$ age $y_1, x$ livesIn $y_2, y_2$ nextLevel $y_3$.

*The measure component remains the same, and* $\Sigma'$ *in the rolled-up query consists of the obvious pairs of the form* $(d, \{d\})$. *Note the change in both the head and body of the classifier, due to the roll-up.*

## 7. EXPERIMENTS

We demonstrate the performance of our RDF analytical framework through a set of experiments. Section 7.1 outlines our implementation and experimental settings. We describe experiments on $\mathcal{I}$ materialization in Section 7.2, evaluate *AnQ*s in Section 7.3 and OLAP operations in Section 7.4, then we conclude. Due to space limitation, experiments performed with query reformulation are delegated to [14].

### 7.1 Implementation and settings

We implemented the concepts and algorithms presented above within our WaRG tool [13]. WaRG is built on top of kdb+ v3.0 (64 bits) [2], an in-memory column DBMS used in decision-support analytics. kdb+ provides arrays (tables), which can be manipulated through the q interpreted programming language. We store in kdb+ the RDF graph G, the *AnS* definitions, as well as the *AnS* instance, when we

---

[3]Dimension hierarchies should not be confused with the hierarchies built using the predefined RDF(S) properties, such as rdfs:subClassOf, e.g., in Figure 2.
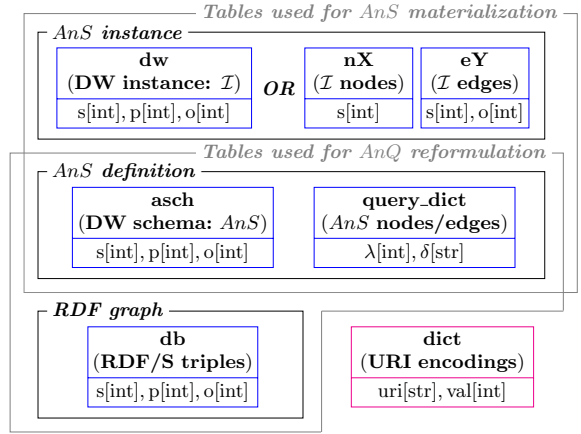


**Figure 5: Data layout of the RDF warehouse.**

| G size | schema size | dictionary | $G^\infty$ size |
|---|---|---|---|
| $3.4 \times 10^7$ triples, 4.4 GB | $5.5 \times 10^3$ triples, 746 KB | $7 \times 10^6$ entries | $3.8 \times 10^7$ triples |

**Table 2: Dataset characteristics.**

choose to materialize it. We translate BGP queries into q programs that kdb+ interprets; any engine capable of storing RDF and processing conjunctive RDF queries could be easily used instead.

**Data organization**. Figure 5 illustrates our data layout in kdb+. The URIs within the RDF dataset are encoded using integers; the mapping is preserved in a q dictionary data structure, named **dict**. The saturation of G, denoted $G^\infty$ (Section 3), is stored in the **db** table. Analytical schema *definitions* are stored as follows. The **asch** table stores the analytical schema triples: $\lambda(n)$ $\lambda(e_{n \to n'})$ $\lambda(n')$. The separate **query_dict** dictionary maps the labels $\lambda$ for nodes and edges to their corresponding queries $\delta$. Finally, we use the **dw** table to store the *AnS* instance $\mathcal{I}$, *or* several tables of the form **nX** and **eY** if a partitioned-table storage is used (see Section 7.2). While **query_dict** and **db** suffice to create the instance, we store the analytical schema definition in **asch** to enable checking incoming analytical queries for correctness w.r.t. the *AnS*.

kdb+ stores each table column independently, and does not have a database-style query optimizer. It is quite fast since it is an in-memory system; at the same time, it relies on the q programmer's skills for obtaining an efficient execution. We try to avoid low-performance formulations of our queries in q, but further optimization is possible and more elaborate techniques (e.g., cost-based join reordering etc.) would further improve performance.

**Dataset**. Our experiments used the *Ontology* and *Ontology Infobox* datasets from the DBpedia Download 3.8; the data characteristics are summarized in Table 2. For our *scalability experiments* (Section 7.2), we replicated these datasets to study scalability in the database size.

**Hardware**. The experiments ran on an 8-core DELL server at 2.13 GHz with 16 GB of RAM, running Linux 2.6.31.14. All times we report are averaged over five executions.

### 7.2 Analytical schema materialization

Loading the (unsaturated) G took about 3 minutes, and computing its full saturation $G^\infty$ 22 minutes. We designed an *AnS* of 26 nodes and 75 edges, capturing a set of concepts and relationship of interest. *AnS* node queries have one or two atoms, while edge queries consist of one to three atoms.
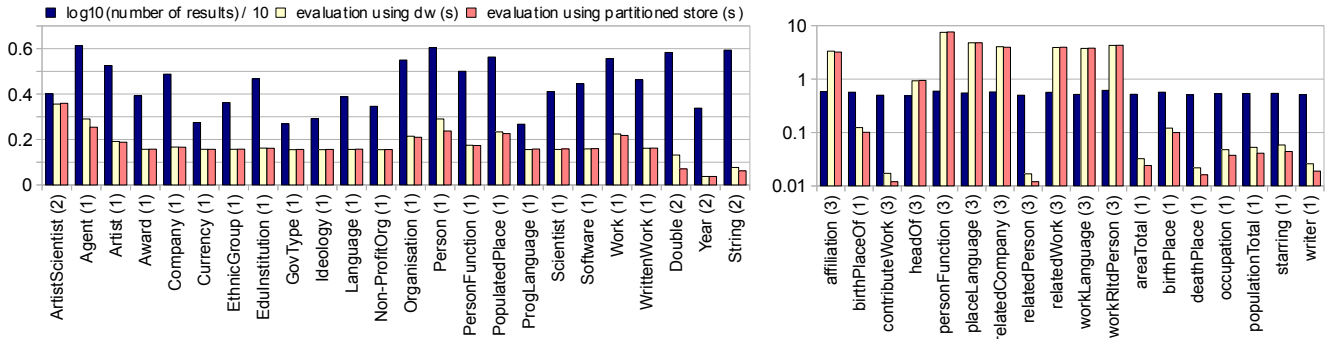
**Figure 6: Evaluation time (s) and number of results for *AnS* node queries (left) and edge queries (right).**
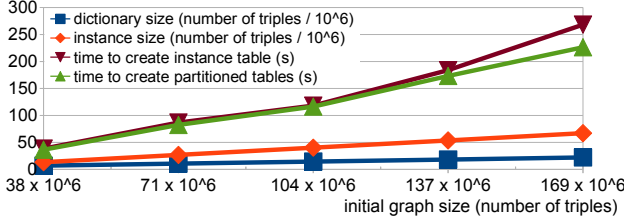


**Figure 7: $\mathcal{I}$ materialization time vs. $\mathcal{I}$ size.**

We considered two ways of materializing the instance $\mathcal{I}$. First, we used a single table (**dw** in Figure 5). Second, inspired from RDF stores such as [21], we tested a *partitioned data layout* for $\mathcal{I}$ as follows. For each distinct node (modeling triples of the form **s** rdf:type $\lambda_X$), we store a table with the subjects **s** declared of that type; this leads to a set of tables denoted **nX** (for **n**ode), with $X \in [1, 26]$. Similarly, for each distinct edge (**s** $\lambda_Y$ **o**) a separate table stores the corresponding triple subjects and objects, leading to the tables **eY** with $Y \in [1, 75]$.

Figure 6 shows for each node and edge query (labeled on the $y$ axis by $\lambda$, chosen based on the name of a "central" class or property in the query): (*i*) the number of query atoms (in parenthesis next to the label), (*ii*) the number of query results (we show $log_{10}(\#res)/10$ to improve readability), (*iii*) the evaluation time when inserting into a single **dw** table, and (*iv*) the time when inserting into the partitioned store. *For 2 node queries and 57 edge queries, the evaluation time is too small to be visible (below 0.01 s)*, and we omitted them from the plots. The *total time to materialize* the instance $\mathcal{I}$ ($1.3 \times 10^7$ triples) was 38 seconds.

**Scalability**. We created larger RDF graphs such that the size of $\mathcal{I}$ would be multiplied by a factor of 2 to 5, with respect to the $\mathcal{I}$ obtained from the original graph G. The corresponding $\mathcal{I}$ materialization time are shown in Figure 7, demonstrating linear scale-up w.r.t. the data size.

### 7.3 Analytical query answering over $\mathcal{I}$

We consider a set of *AnQ*s, each adhering to a specific *query pattern*. A pattern is a combination of: (*i*) the number of atoms in the classifier query (denoted **c**), (*ii*) the number of dimension variables in the classifier query (denoted **v**), and (*iii*) the number of atoms in the measure query (denoted **m**). For instance, the pattern **c5v4m3** designates queries whose classifiers have 5 atoms, aggregate over 4 dimensions, and whose measure queries have 3 atoms. We used **12 distinct patterns** for a total of **1,097 queries**.

The graph at the top of Figure 8 shows for each query pattern, the number of queries in the set (in parenthesis after
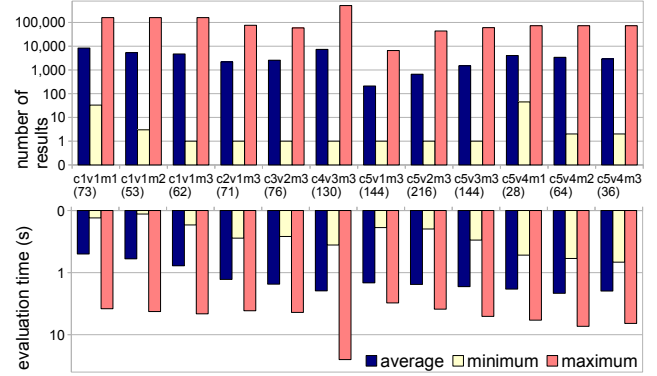


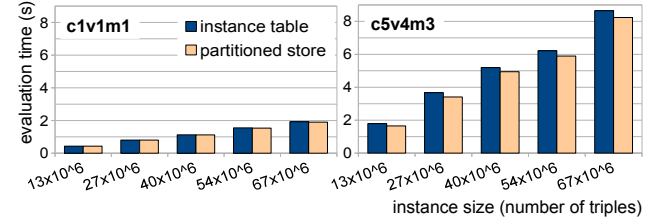**Figure 8: *AnQ* statistics for query patterns.**



**Figure 9: *AnQ* evaluation time over large datasets.**

the pattern name), and the average, minimum and maximum number of query results. The largest result set (for c4v3m3) is $514, 240$, while the second highest (for c1v1m3) is $160, 240$. The graph at the bottom of Figure 8 presents the average, minimum and maximum query evaluation times among the queries of each pattern.

Figure 8 shows that query result size (up to hundreds of thousands) is the most strongly correlated with query evaluation time. Other parameters impacting the evaluation time are the number of atoms in the classifier and measure queries, and the number of aggregation variables. These parameters are to be expected in an in-memory execution engine such as kdb+. Observe the moderate time increase with the main query size metric (the number of atoms); this demonstrates robust performance even for complex *AnQ*s.

Figure 9 shows the average evaluation time for queries belonging to the sets c1v1m1 and c5v4m3 over increasing tables, using the instance triple table and the partitioned store implementations. In both cases the evaluation time increases linearly with the size of the dataset. The graph shows that the partitioned store brings a modest speed-up (about 10%); for small queries, the difference is unnoticeable. Thus, without loss of generality, in the sequel we consider only the single-table **dw** option.
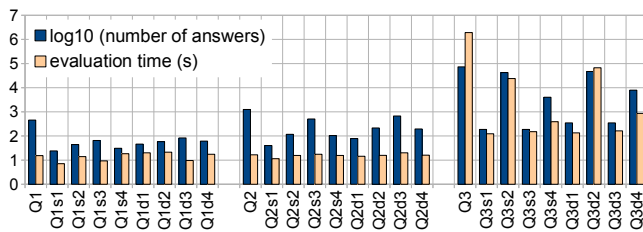
**Figure 10: Slice and dice over $AnQ$s.**

## 7.4 OLAP operations

We now study the performance of OLAP operations on analytical queries (Section 6).

**Slice and dice**. In Figure 10, we consider three c5v4m3 queries: $Q_1$ having a small result size (455), $Q_2$ with a medium result size $(1,251)$ and $Q_3$ with a large result size $(73,242)$. For each query we perform a slice (dice) by restricting the number of answers for each of its 4 dimension variables, leading to the OLAP queries $Q_{1s1}$ to $Q_{1s4}$, $Q_{1d1}$ to $Q_{1d4}$ and similarly for $Q_2$ and $Q_3$. The figure shows that the slice/dice running time is strongly correlated with the result size, and is overall small (under 2 seconds in many cases, 4 seconds for $Q_3$ slice and dice queries having $10^4$ results).

**Drill-in and drill-out**. The queries following the patterns c5v1m3, c5v2m3, c5v3m3 and c5v4m3 were chosen starting from the ones for c5v4m3 and eliminating one dimension variable from the classifier (without any other change) to obtain c5v3m3; removing one further dimension variable yielded the c5v2m3 queries etc. Recalling the definitions of drill-in and drill-out (Section 6), it follows that the queries in c5v**n**m3 are drill-ins of c5v**(n+1)**m3 for $1 \le n \le 3$, and conversely, c5v**(n+1)**m3 result from drill-out on c5v**n**m3. Their evaluation times appear in Figure 8.

## 7.5 Conclusion of the experiments

Our experiments demonstrate the feasibility of our full RDF warehousing approach, which exploits standard RDF functionalities such as triple storage, conjunctive query evaluation, and reasoning. We showed robust scalable performance when loading and saturating G, and building $\mathcal{I}$ in time linear in the input size (even for complex, many-joins node and edge queries). Finally, we proved that OLAP operations can be evaluated quite efficiently in our RDF cube ($AnQ$) context. While further optimizations are possible, our experiments confirmed the interest and good performance of our proposed all-RDF Semantic Web warehousing approach.

**Perspective: OLAP operations evaluated on $AnQ$s.** The OLAP operations described thus far were applied on $AnQ$s and evaluated against $\mathcal{I}$. We are interested in improving performance of such operations by evaluating them directly on the materialized results of previous analytical queries (significantly reducing the input data and benefiting from the regular-structure $AnQ$ results). We plan to analyze the situations where such "shortcuts" are applicable.

## 8. RELATED WORK

Relational warehousing has been well studied [23, 22]. Web data warehouses have been presented as interconnected corpora of XML documents and Web services [5], or as distributed knowledge bases [6]. In [30], a large RDF knowledge base, Yago [32], is enriched with information gathered from the Web. These works did not consider RDF analytics.

[16, 34] propose RDF(S) vocabularies (pre-defined classes and properties) for describing *relational* multidimensional data in RDF; [16] also maps OLAP operations into SPARQL queries. [27] presents a semi-automated approach for deriving a RDW from an ontology. In contrast with the above, in our approach, the $AnS$ instance is an RDF graph itself thus seamlessly preserves the heterogeneity, semantics, and ability to query the schema with the data present in RDF.

In the area of RDF data management, previous works focused on efficient stores [4, 11, 31], indexing [36], query processing [28] and multi-query optimization [26], view selection [17] and query-view composition [25], or Map-Reduce based RDF processing [20, 21]. BGP query answering techniques have been studied intensively, e.g., [18, 29], and some are deployed in commercial systems such as Oracle 11g, which provides a "Semantic Graph" extension etc. *Our work defines a novel framework for RDF analytics, based on analytical schemas and queries*; these can be efficiently deployed on top of any RDF data management platform, to extend it with analytic capabilities.

Analysis cubes and OLAP operations on cubes over graphs are also defined in [40]. However, their approach does not handle *heterogeneous* graphs, and thus it cannot handle multi-valued attributes (e.g., a movie being both a comedy and a romance), nor data semantics, both central in RDF. Further, their approach only focuses on *counting edges* in contrast with our flexible $AnQ$ (Section 4.2).

In [10], graph data can be aggregated in a spatial fashion by grouping connected nodes into *regions* (think of a street map graph); based on this simple aggregation, an OLAP framework is built. Beyond being RDF-specific (unlike [10]), our framework also introduces analytical graph schemas, and allows for much more general aggregation.

[24] proposes techniques for transforming OLAP queries into SPARQL. Query answering is optimized by materializing data cubes. Such processing can be added to our framework in order to further optimize $AnQ$ answering.

The separation between grouping and aggregation present in our $AnQ$s is similar to the MD-join operator [9] for RDWs.

Finally, SPARQL 1.1 [35] features SQL-style grouping and aggregation. Deploying our framework on an efficient SPARQL 1.1 platform enables taking advantage both of its efficiency and of the high-level, expressive, flexible RDF graph analysis concepts introduced in this work.

## 9. CONCLUSION

DW models and techniques have had a strong impact on the usages and usability of data. In this work, we proposed the first approach for specifying and exploiting an *RDF data warehouse*, notably by $(i)$ defining an analytical schema that captures the information of interest, and $(ii)$ formalizing analytical queries (or cubes) over the $AnS$. Importantly, instances of $AnS$ are *RDF graphs themselves*, which allows to exploit the semantics and rich, heterogeneous structure (e.g., jointly query the schema and the data) that make RDF data rich and interesting.

The broader area of *data analytics*, related to data warehousing, albeit with a significantly extended set of goals and methods, is the target of very active research now, especially in the context of massively parallel Map-Reduce processing etc. Efficient methods for deploying $AnS$s and $AnQ$ evaluation in such a parallel context are part of our future work.

# 10. REFERENCES

[1] The DBpedia Knowledge Base. `http://dbpedia.org`.

[2] [kx] white paper. kx.com/papers/KdbPLUS_Whitepaper-2012-1205.pdf.

[3] OLAP council white paper. http://www.olapcouncil.org/research/resrchly.htm.

[4] D. J. Abadi, A. Marcus, S. R. Madden, and K. Hollenbach. Scalable semantic web data management using vertical partitioning. In *VLDB*, 2007.

[5] S. Abiteboul. Managing an XML warehouse in a P2P context. In *CAiSE*, 2003.

[6] S. Abiteboul, E. Antoine, and J. Stoyanovich. Viewing the web as a distributed knowledge base. In *ICDE*, 2012.

[7] S. Abiteboul, R. Hull, and V. Vianu. *Foundations of Databases*. Addison-Wesley, 1995.

[8] S. Abiteboul, I. Manolescu, P. Rigaux, M.-C. Rousset, and P. Senellart. *Web Data Management and Distribution*. Cambridge University Press, Dec 2011.

[9] M. Akinde, D. Chatziantoniou, T. Johnson, and S. Kim. The MD-join: An operator for complex OLAP. In *ICDE*, pages 524–533, 2001.

[10] D. Bleco and Y. Kotidis. Business intelligence on complex graph data. In *EDBT/ICDT Workshops*, 2012.

[11] M. A. Bornea, J. Dolby, A. Kementsietsidis, K. Srinivas, P. Dantressangle, O. Udrea, and B. Bhattacharjee. Building an efficient RDF store over a relational database. In *SIGMOD Conference*, pages 121–132, 2013.

[12] S. Campinas, T. E. Perry, D. Ceccarelli, R. Delbru, and G. Tummarello. Introducing RDF graph summary with application to assisted SPARQL formulation. *2012 23rd International Workshop on Database and Expert Systems Applications*, 0:261–266, 2012.

[13] D. Colazzo, T. Ghosh, F. Goasdoué, I. Manolescu, and A. Roatiş. WaRG: Warehousing RDF Graphs (demonstration). In *Bases de Données Avancées (informal national French conference, no proceedings)*, 2013. See `https://team.inria.fr/oak/warg/`.

[14] D. Colazzo, F. Goasdoué, I. Manolescu, and A. Roatiş. Warehousing RDF graphs. In *Bases de Données Avancées (informal national French conference, no proceedings)*, 2013.

[15] O. Erling and I. Mikhailov. RDF Support in the Virtuoso DBMS. *Networked Knowledge - Networked Media*, pages 7–24, 2009.

[16] L. Etcheverry and A. A. Vaisman. Enhancing OLAP analysis with web cubes. In *ESWC*, 2012.

[17] F. Goasdoué, K. Karanasos, J. Leblay, and I. Manolescu. View selection in Semantic Web databases. *PVLDB*, 5(1), 2012.

[18] F. Goasdoué, I. Manolescu, and A. Roatiş. Efficient query answering against dynamic RDF databases. In *International Conference on Extending Database Technology*, pages 299–310, 2013.

[19] A. Y. Halevy. Answering queries using views: A survey. *VLDB J.*, 10(4):270–294, 2001.

[20] J. Huang, D. J. Abadi, and K. Ren. Scalable SPARQL Querying of Large RDF Graphs. *PVLDB*, 4(11), 2011.

[21] M. Husain, J. McGlothlin, M. M. Masud, L. Khan, and B. M. Thuraisingham. Heuristics-Based Query Processing for Large RDF Graphs Using Cloud Computing. *IEEE Trans. on Knowl. and Data Eng.*, 2011.

[22] M. Jarke, M. Lenzerini, Y. Vassiliou, and P. Vassiliadis. *Fundamentals of Data Warehouses*. Springer, 2001.

[23] C. S. Jensen, T. B. Pedersen, and C. Thomsen. *Multidimensional Databases and Data Warehousing*. Synthesis Lectures on Data Management. Morgan & Claypool Publishers, 2010.

[24] B. Kämpgen and A. Harth. No size fits all – running the star schema benchmark with SPARQL and RDF aggregate views. In *ESWC 2013, LNCS 7882*, pages 290–304, Heidelberg, Mai 2013. Springer.

[25] W. Le, S. Duan, A. Kementsietsidis, F. Li, and M. Wang. Rewriting queries on SPARQL views. In *WWW*, pages 655–664, 2011.

[26] W. Le, A. Kementsietsidis, S. Duan, and F. Li. Scalable multi-query optimization for SPARQL. In *ICDE*, pages 666–677, 2012.

[27] V. Nebot and R. B. Llavori. Building data warehouses with semantic web data. *Decision Support Systems*, 52(4), 2012.

[28] T. Neumann and G. Weikum. The RDF-3X engine for scalable management of RDF data. *VLDB J.*, 19(1), 2010.

[29] J. Pérez, M. Arenas, and C. Gutierrez. nSPARQL: A navigational language for RDF. *J. Web Sem.*, 8(4):255–270, 2010.

[30] N. Preda, G. Kasneci, F. M. Suchanek, T. Neumann, W. Yuan, and G. Weikum. Active knowledge: dynamically enriching RDF knowledge bases by web services. In *SIGMOD*, 2010.

[31] L. Sidirourgos, R. Goncalves, M. Kersten, N. Nes, and S. Manegold. Column-store support for RDF data management: not all swans are white. *PVLDB*, 1(2), 2008.

[32] F. M. Suchanek, G. Kasneci, and G. Weikum. YAGO: A large ontology from Wikipedia and WordNet. *J. Web Sem.*, 6(3), 2008.

[33] W3C. Resource description framework. http://www.w3.org/RDF/.

[34] W3C. The RDF data cube vocabulary. http://www.w3.org/TR/vocab-data-cube/, 2012.

[35] W3C. SPARQL 1.1 query language. http://www.w3.org/TR/sparql11-query/, March 2013.

[36] C. Weiss, P. Karras, and A. Bernstein. Hexastore: sextuple indexing for Semantic Web data management. *PVLDB*, 1(1), 2008.

[37] Jena. `http://jena.sourceforge.net`.

[38] Owlim. `http://owlim.ontotext.com`.

[39] Sesame. `http://www.openrdf.org`.

[40] P. Zhao, X. Li, D. Xin, and J. Han. Graph cube: on warehousing and OLAP multidimensional networks. In *SIGMOD*, pages 853–864, 2011.