Warehousing RDF Graphs

Dario Colazzo, François Goasdoué, Ioana Manolescu, Alexandra Roatis

Warehousing RDF Graphs*

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

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

Ioana Manolescu
Inria Saclay and
Université Paris-Sud, France
ioana.manolescu@inria.fr

Alexandra Roatiş
Université Paris-Sud and
Inria Saclay, France
alexandra.roatis@lri.fr

Abstract

Research in data warehousing (DW) has developed expressive and efficient tools for the multidimensional analysis of large amounts of data. As more data gets produced and shared in RDF, analytic concepts and tools for analyzing such irregular, graph-shaped, semantic-rich data need to be revisited. We introduce *the first all-RDF model for warehousing RDF graphs*. Notably, we define *analytical schemas* and *analytical queries* for RDF, corresponding to the relational DW star/snowflake schemas and cubes. We also show how *typical OLAP operations* can be performed on our RDF cubes, and experiments on a fully-implemented platform demonstrating the practical interest of our approach.

Keywords: RDF, data warehouse, OLAP

Résumé

La recherche sur les entrepôts de données a mené des techniques et outils efficaces pour l'analyse multidimensionnelle de grandes quantités de données. Avec la monté en puissance de la production et du partage de données RDF, les concepts et outils d'analyse multidimensionnelle pour ces données irrégulières, de type graphe, et sémantiquement riches ont besoin d'être revisités. Nous introduisons le premier modèle tout-RDF pour les entrepôts de graphes RDF. Notamment, nous définissons les schémas analytiques et les requêtes analytiques pour RDF, correspondant aux schémas en étoile/flocon et cubes des entrepôts relationnels. Nous montrons aussi comment les opérations OLAP typiques peuvent être effectuées sur nos cubes RDF, et nous présentons quelques expériences validant l'intérêt pratique de notre approche.

*This work was performed while the authors were with LRI (Université Paris-Sud) and Inria Saclay, France.

1 Introduction

Databases of *facts*, each characterized by multiple *dimensions*, whose values are recorded in *measures*, are at the core of *multidimensional data warehouses* (DWs in short) [17]. The facts can then be analyzed by means of *aggregating* the measures, e.g., *"what is the average sale price of item A every month in every store?"* One of the pioneer books on the topic is Immon's [15], which lists a set of data warehouse characteristics: the data is *integrated* (possibly through an *E*xtract-*T*ransform-*L*oad process that feeds the warehouse with well-structured data; data is *typically non volatile*, since a recorded fact or measure is unlikely to change in the future, data only gets added to the warehouse; finally, *time* is an important dimension in most DW applications.

Relational data warehousing. Data warehouses are typically built to *analyze (some aspects of) an enterprise's business processes*. Thus, a first crucial task is *choosing* among the many data sources available to the analyst, those that are interesting for a given class of business questions that the DW is designed for answering. The analysts then describe the facts, dimensions, and measures to be analyzed. Then, for each relevant business question, an *analytical query* is formulated, by (*i*) classifying facts along a set of dimensions and (*ii*) reporting the aggregated values of their measures. Such queries are commonly known as *cubes*. For all its practical applications, data warehousing has attracted enormous interest, both from practitioners [18] and from the research community [12, 16, 26]; warehousing tools are now part of major relational database servers. Relational data warehousing is thus a pretty mature area.

Semantic Web data and RDF. Recent years have witnessed a steady and important interest in Semantic Web data, represented within the W3C's Resource Description Framework (or RDF, in short) [27]. The RDF model allows describing *resources* (either digital or taken from the real world), by specifying the *values* of their *properties*. Thus, an RDF information unit is a triple `s p o`, with `s`, `p` and `o` standing for the *subject*, *property* and *object*, respectively, in RDF terminology. The RDF language is increasingly being used in order to *export, share, and collaboratively author data* in many settings. For instance, it serves as a *metadata language* to describe cultural artifacts in large digital libraries, and to encode protein sequence data, as in the Uniprot data set. RDF is a natural target for representing heterogeneous facts contributed by millions of Wikipedia users, gathered within the DBpedia data source, as well as for the *Linked Open Data* effort, aiming at connecting and sharing collectively produced data and knowledge.

RDF analytics. The current popularity of RDF raises interest in *models and tools for RDF data analytics*. For instance, consider applications seeking to *harvest, aggregate and analyze user data* from various sources (such as social networks, blog posts, comments on public Web sites etc.). The data is *heterogeneous*; it may include facts about the user such as age, gender or region, an endorsement of a restaurant the user liked etc. The data is *graph-structured*, since it describes relationships between users, places, companies etc. It comes from multiple sources and may have attached *semantics*, based on some ontologies for which RDF is an ideal format.

Analyzing Semantic Web data with warehouse-style tools has many applications. For instance, in a *City Open Data* scenario, RDF data sources describing providers of goods and services in a city can be integrated, e.g., based on a city map, while social and Web feeds referring to the providers can be warehoused to analyze service usage, identify trends and user groups, and take decisions such as opening a new branch in an under-served city

area, or better tune a transport service to the user needs. Efforts to open city data are significantly advanced or under way in major French cities, including Rennes, Grenoble and Paris.

Despite the perceived need, there is currently *no satisfactory conceptual and practical solution for large-scale RDF analytics*. Relational DW tools are not easily adaptable, since loading RDF data in a relational analytical schema may lead to facts with unfilled or multiply-defined dimensions or measures; the latter does not comply with the relational multidimensional setting and DW tools. More important, to fully exploit RDF graphs, *the heterogeneity and rich semantics of RDF data should be preserved* through the warehouse processing chain and up to the analytical queries. In particular, RDF analytical queries should be allowed to *jointly query the schema and the data*, e.g., ask for most frequently specified properties of a CollegeStudent, or the three largest categories of Inhabitants. *Changes to the underlying database* (such as adding a new subclass of Inhabitant) *should not cause the warehouse schema to be re-designed; instead, the new resources (and their properties) should propagate smoothly to the analysis schema and cubes*.

In this work, we *establish formal models, validated through a fully deployed efficient tool, for warehouse-style analytics on Semantic Web data*. To our knowledge, this is the first work proposing an all-RDF formal model for data warehousing. Our contributions are:

- We provide a formal model for *analytical schemas*, capturing *the data of interest for a given analysis* of an RDF data set. Importantly, an analytical schema instance is an RDF graph itself, and as such, it preserves the heterogeneity, semantics, and flexible mix of schema and data present in the RDF model.
- We introduce *analytical queries*, RDF counter-parts of the relational analytical cubes, supporting typical analytical operations (slice, dice, etc.). We show how these operations can be realized efficiently on top of a conjunctive query processor (relational or RDF-specific).
- We implemented an RDF warehouse prototype on top of an efficient in-memory column-based store. Our experiments confirm its interest and performance.

The remainder of this paper is organized as follows. Section 2 recalls the RDF model, based on which Section 3 presents our analytical schemas and queries, and Section 4 studies efficient methods for evaluating analytical queries. Section 5 introduces typical analytical operations (slice, dice etc.) on our RDF analytical cubes. We present our experimental evaluation in Section 6, discuss related work, and then conclude.

2 RDF graphs and queries

We introduce RDF data graphs in Section 2.1 and RDF queries in Section 2.2.

2.1 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 assume given a set U of URIs, a set L of literals (constants), and a set B of blank nodes (unknown URIs or literals), such that U, B and L are pairwise disjoint. As per the RDF specification [27], triple subject values belong to $U \cup B$, property values belong to U, and object values belong to $U \cup B \cup L$. Moreover, we use *typed RDF literals*, whose types belong to a set $\tau = \{\text{string, integer, double, } \ldots\}$.

Constructor	Triple	Relational notation
Class assertion	s rdf:type o	o(s)
Property assertion	s p o	p(s, o)

Figure 1: RDF statements.

$\mathtt{G} = \{\text{user}_1 \text{ hasName “Bill”, user}_1 \text{ hasAge “28”, user}_1 \text{ friend user}_3, \text{user}_1 \text{ bought product}_1,$
$\text{product}_1 \text{ rdf:type SmartPhone, user}_1 \text{ worksWith user}_2, \text{user}_2 \text{ hasAge “40”}, \ldots\}$

Figure 2: Sample RDF triples.

Blank nodes are essential features of RDF allowing to support *unknown URI/literal tokens*. For instance, one can use a blank node $\_{:}b_1$ to state that the country of $\_{:}b_1$ is *United States* while the city of the same $\_{:}b_1$ is *Washington*; at the same time, the population of *Washington* can be said to be an unspecified value $\_{:}b_2$.

Notation. We use s, p, o and $\_{:}b$ in triples (possibly with subscripts) as placeholders. That is, s stands for values in $U \cup B$, p stands for values in U, o represents values from $U \cup B \cup L$, and $\_{:}b$ denotes values in B. We also use strings between quotes as in “*string*” to denote string-typed literals from L. Finally, the set of values (URIs, blank nodes, literals) of an RDF graph G is denoted Val(G).

Figure 1 shows how to use triples to describe resources, that is, to express class (unary relation) and property (binary relation) assertions. The RDF standard [27] 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 which 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.

Definition 1 (Graph notation of an RDF graph) *An RDF graph is a labeled directed graph* $\mathtt{G} = \langle \mathcal{N}, \mathcal{E}, \lambda \rangle$ *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*;*
- $\lambda : \mathcal{N} \cup \mathcal{E} \to U \cup B \cup L$ *is a* labeling function *such that* $\lambda_{|\mathcal{N}}$ *is injective, with* $\lambda_{|\mathcal{N}^0} : \mathcal{N}^0 \to U \cup B \cup L$ *and* $\lambda_{|\mathcal{N}^{>0}} : \mathcal{N}^{>0} \to U \cup B$*, and* $\lambda_{|\mathcal{E}} : \mathcal{E} \to U$*.*

Example 1 (RDF Graph) *We consider an RDF graph comprising information about users and products. Figure 2 shows some of the triples, whereas Figure 3 depicts the whole dataset using its graph notation. 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.*

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 4 shows the allowed constraints and how to express them; in this figure, $\mathtt{s}, \mathtt{o} \in U \cup B$, while *domain* and *range* denote respectively the first and second attribute of every property.

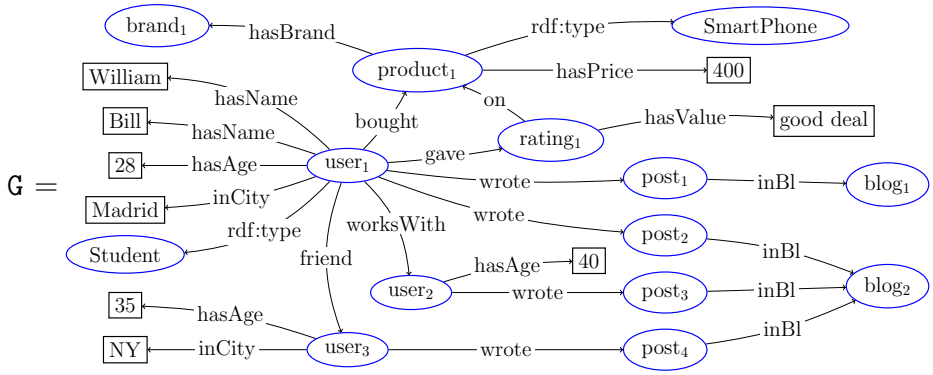


Figure 3: Running example: RDF graph.

Subclass constraint	
Triple	$\mathtt{s}$ rdfs:subClassOf $\mathtt{o}$
Relational (under OWA)	$\mathtt{s} \subseteq \mathtt{o}$
Subproperty constraint	
Triple	$\mathtt{s}$ rdfs:subPropertyOf $\mathtt{o}$
Relational (under OWA)	$\mathtt{s} \subseteq \mathtt{o}$
Domain typing constraint	
Triple	$\mathtt{s}$ rdfs:domain $\mathtt{o}$
Relational (under OWA)	$\Pi_{\text{domain}}(\mathtt{s}) \subseteq \mathtt{o}$
Range typing constraint	
Triple	$\mathtt{s}$ rdfs:range $\mathtt{o}$
Relational (under OWA)	$\Pi_{\text{range}}(\mathtt{s}) \subseteq \mathtt{o}$

Figure 4: RDFS statements.

Traditionally constraints can be interpreted in two ways [6] under the closed-world assumption (CWA) or under the open-world assumption (OWA). Under CWA, any fact not present in the database is assumed not to hold. Under this assumption, if the set of database facts does not respect a constraint, then the database is *inconsistent*. For instance, the CWA interpretation of a constraint of the form $R_1 \subseteq R_2$ is: any tuple in the relation R_1 *must* also be in the relation R_2 *in the database*, otherwise the database is inconsistent. On the contrary, under OWA, some facts may hold even though they are *not in the database*. For instance, the OWA interpretation of the same example 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).

The RDF data model [27] – and accordingly, the present work – is based on OWA, and this is how we interpret all the constraints in Figure 4.

Example 2 (RDF Schema) *Consider next to the graph* $\mathtt{G}$ *from Figure 3, the schema depicted in Figure 5. 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 that person etc.*

RDF schemas vs. relational schemas. It is worth stressing that the RDFS constraint language (outlined in Figure 4 and illustrated in the above example) is much "weaker" than those from the traditional relational setting. RDFS does not allow *enforcing* that instances of a given class have a specific property: if r_1 belongs to class C_1, this

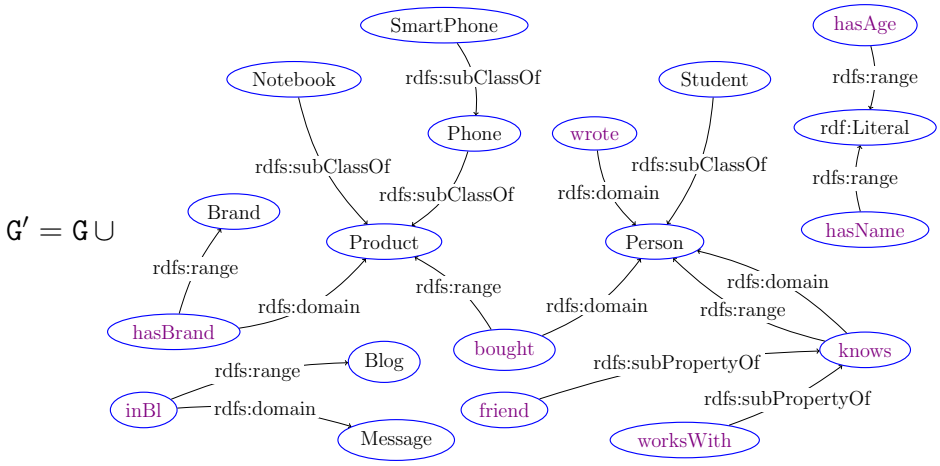


Figure 5: Running example: RDF Schema.

has no impact over which properties of r_1 are (or are not) specified in the RDF graph. In other words: the presence of a schema *does not* restrain and even less eliminate the heterogeneity of an RDF graph. Instead, the schema *adds semantics to the data*, which may still be very heterogeneous.

RDF entailment. Our discussion about OWA above illustrated an important RDF feature: *implicit triples*, considered to be part of the RDF graph even though they are not explicitly present in it. The W3C names *RDF entailment* the mechanism through which, based on the set of explicit triples and some *entailment rules* (to be described shortly), 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 $\mathtt{s\ p\ o}$ is entailed by a graph $\mathtt{G}$, denoted $\mathtt{G} \vdash_{\text{RDF}} \mathtt{s\ p\ o}$, if and only if there is a sequence of applications of immediate entailment rules that leads from $\mathtt{G}$ to $\mathtt{s\ p\ o}$ (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 $\mathtt{G}$, which is the RDF graph $\mathtt{G}^{\infty}$ defined as the fixpoint obtained by repeatedly applying $\vdash^{i}_{\text{RDF}}$ on $\mathtt{G}$.

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 $\mathtt{G}$ and its saturation: $\mathtt{G} \vdash_{\text{RDF}} \mathtt{s\ p\ o}$ if and only if $\mathtt{s\ p\ o} \in \mathtt{G}^{\infty}$. RDF entailment is part of the RDF standard itself; in particular, *the answers of a query posed on* $\mathtt{G}$ *must take into account all triples in* $\mathtt{G}^{\infty}$, since *the semantics of an RDF graph is its saturation.* RDF saturation is supported by many popular RDF platforms such as Sesame, Jena or OWLIM.

Immediate entailment rules. The RDF Schema specification comprises a set of immediate entailment rules. Of interest to us are the rules deriving RDF statements through the transitivity of class and property inclusions, and from inheritance of domain and range typing. Using a tabular notation, with the entailed (consequence) triple shown at the bottom, some examples are:

$$\frac{\begin{array}{c}\text{SmartPhone rdfs:subClassOf Phone}\\ \text{Phone rdfs:subClassOf Product}\end{array}}{\text{SmartPhone rdfs:subClassOf Product}}$$

and similarly:

$$\frac{\text{worksWith rdfs:subPropertyOf knows} \quad \text{knows rdfs:domain Person}}{\text{worksWith rdfs:domain Person}}$$

Some other rules derive entailed RDF statements, through the propagation of values (URIs, blank nodes, and literals) from sub-classes and sub-properties to their super-classes and super-properties, and from properties to classes typing their domains and ranges. Within our running example:

$$\frac{\dfrac{\text{worksWith rdfs:subPropertyOf knows} \quad \text{user}_1 \text{ worksWith user}_2}{\text{user}_1 \text{ knows user}_2} \quad \text{knows rdfs:domain Person}}{\text{user}_1 \text{ rdf:type Person}}$$

2.2 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. Subjects and properties can be URIs, blank nodes or variables; objects can also be literals.

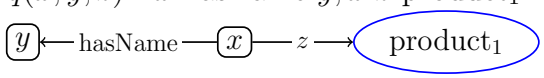
A boolean BGP query is of the form `ASK WHERE` $\{t_1, \ldots, t_\alpha\}$, while a non-boolean BGP query is of the form `SELECT` $\bar{x}$ `WHERE` $\{t_1, \ldots, t_\alpha\}$, where $\{t_1, \ldots, t_\alpha\}$ is a BGP; the query head variables $\bar{x}$ are called *distinguished variables*, and are a subset of the variables occurring in $t_1, \ldots, t_\alpha$.

Notation. Without loss of generality, in the following we will use the conjunctive query notation $q(\bar{x})$:- $t_1, \ldots, t_\alpha$ for both `ASK` and `SELECT` queries (for boolean queries $\bar{x}$ is empty). We use x, y, and z (possibly with subscripts) to denote variables in queries. We denote by $\texttt{VarBl}(q)$ the set of variables *and* blank nodes occurring in the query q. For a BGP query $q(\bar{x})$:- $t_1, \ldots, t_\alpha$, the head of q denoted $\texttt{head}(q)$ is $q(\bar{x})$ and the body of q denoted $\texttt{body}(q)$ is $t_1, \ldots, t_\alpha$.

BGP query graph. For our purposes, it is useful to view each triple atom in the body of a BGP q as a *generalized RDF triple*, where, beyond URIs, blank nodes and literals, *variables* may also appear in any of the subject, predicate and object positions. This naturally leads to a *graph notations for BGP queries*, which is the corresponding generalization of that for RDF graphs as described in Definition 1.

For instance, the query: $q(x, y, z)$:- x hasName y, x z product$_1$

is represented by the graph:



Query evaluation. Given a query q and a RDF graph $\texttt{G}$, the *evaluation of q against* $\texttt{G}$ is: $q(\texttt{G}) = \{\bar{x}_\mu \mid \mu : \texttt{VarBl}(q) \to \texttt{Val}(\texttt{G})$ is a total assignment s.t. $t_1^\mu \in \texttt{G}, t_2^\mu \in \texttt{G}, \ldots, t_\alpha^\mu \in \texttt{G}\}$ where we denote by t^μ the result of replacing every occurrence of a variable or blank node $e \in \texttt{VarBl}(q)$ in the triple t, by the value $\mu(e) \in \texttt{Val}(\texttt{G})$[1].

Notice that evaluation *treats the blank nodes in a query exactly as it treats non-distinguished variables*. Thus, in the sequel, without loss of generality, we consider queries

[1]We assume μ respects the SPARQL rules [29] governing the conversions of different-type atomic values (recall we use *typed* RDF literals) for join or selection comparisons etc.

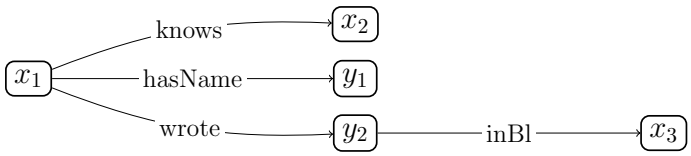


Figure 6: Rooted query example.

where all blank nodes have been replaced by distinct (new) non-distinguished variable symbols.

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

Example 3 (BGP Query Answering) *The following query asks for the names of those having bought a product related to* Phone*:*

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

In this example, $q(\mathtt{G}'^{\infty}) = \{\langle \text{“Bill”} \rangle\}$.
This answer results from $\mathtt{G}' \vdash_{\text{RDF}} \text{product}_1$ rdf:type Phone *and the assignment* $\mu = \{y_1 \to \text{user}_1, x \to \text{Bill}, y_2 \to \text{product}_1, y_3 \to \text{rdf:type}\}$.
Note that evaluating q *against* $\mathtt{G}'$ *leads to the incomplete (empty) answer set* $q(\mathtt{G}') = \{\langle\rangle\}$.

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*) [17]. 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 measure to be used (Section 3.2).

Definition 2 (Rooted Query) *Let* q *be a BGP query,* $\mathtt{G} = \langle \mathcal{N}, \mathcal{E}, \lambda \rangle$ *its graph and* $n \in \mathcal{N}$ *a node whose label is a variable in* q*. The query* q *is* rooted in n *iff* $\mathtt{G}$ *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 label.*

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

The graph representation of the query, given in Figure 6, shows that every node is reachable from the root x_1*.*

Next, we introduce the concept of *join* query, which joins some BGP queries on their distinguished variables, and projects out some of these distinguished variables. Join queries will be useful later on when defining data warehouse analyses, as well as operations (e.g., drill down, dice etc.) on the results of such 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* $q^{\bowtie}$*:*

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

Observe that the above definition considers queries that do not share non-distinguished variables (a.k.a. variables which are not present in the head of the query). 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 already 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:*

$$\begin{array}{ll} q_1(x_1, x_2) \text{:-} & x_1 \text{ hasAge } x_2,\ x_1 \text{ bought } y_1 \\ q_2(x_1, x_3) \text{:-} & x_1 \text{ wrote } y_2,\ y_2 \text{ inBl } x_3 \end{array}$$

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.,

$$\begin{array}{ll} q^{\bowtie}_{1,2}(x_1, x_2) \text{:-} & x_1 \text{ hasAge } x_2,\ x_1 \text{ bought } y_1, \\ & x_1 \text{ wrote } y_2,\ y_2 \text{ inBl } x_3 \end{array}$$

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

3 RDF graph analysis

We define here the basic ingredients of our approach for analyzing RDF graphs. An *analytical schema* is first designed, and then *mapped* onto an RDF graph to analyze, as we explain in Section 3.1. This defines the instance of the analytical schema to be analyzed with an *analytical query*, introduced in Section 3.2, modeling the chosen criteria (a.k.a. dimensions) and measurable or countable attributes (a.k.a. measures) of the analysis.

3.1 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 n can be analyzed using the schema nodes reachable from n as dimensions and measures.* This makes our analytical schema model extremely flexible and more general than the traditional DW setting where facts (at the center of a star or snowflake schema) are analyzed according to a specific set of measures.

From a Semantic Web (RDF) perspective, each analytical schema node corresponds to an RDF class assertion, while each analytical schema edge corresponds to an RDF property assertion; thus, the analytical schema is a full-fledged RDF instance itself. Formally:

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 $n \to n' \in \mathcal{E}$ a binary BGP query $\delta(n \to n') = q(x, y)$.*

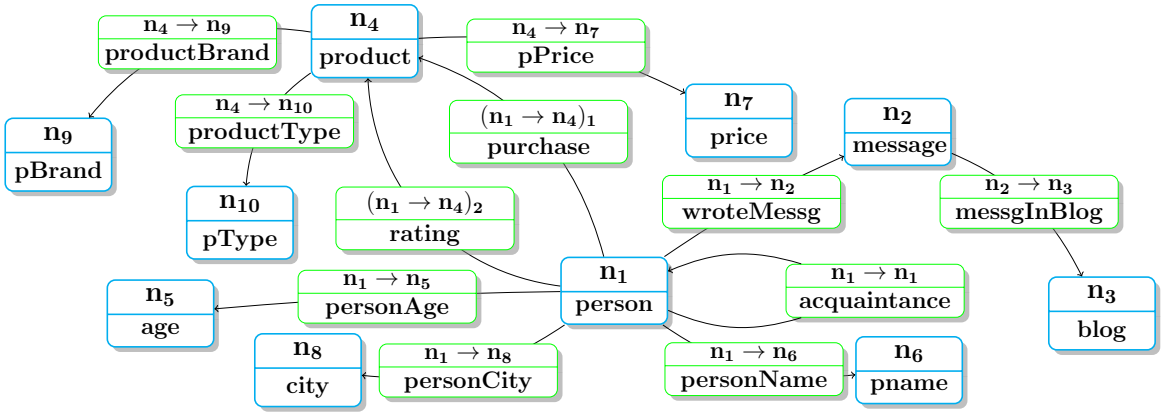


Figure 7: Sample Analytical Schema (AnS).

node n	$\lambda(n)$	$\delta(n)$
n_1	*person*	$q(x)$:- x rdf:type Person
n_2	*message*	$q(x)$:- y wrote x, x inBl b , b rdf:type Blog
n_4	*product*	$q(x)$:- x rdf:type Product
n_5	*age*	$q(x)$:- y hasAge x
n_6	*pname*	$q(x)$:- y hasName x
n_9	*pBrand*	$q(x)$:- y hasBrand x,
n_{10}	*pType*	$q(x)$:- x rdfs:subClassOf Product

edge $n \to n'$	$\lambda(n \to n')$	$\delta(n \to n')$
$n_1 \to n_1$	*acquaintance*	$q(x,y)$:- x knows y
$n_1 \to n_4$	*purchase*	$q(x,y)$:- x bought y
$n_1 \to n_5$	*personAge*	$q(x,y)$:- x rdf:type Person, x hasAge y
$n_2 \to n_3$	*messgInBlog*	$q(x,y)$:- x rdf:type Message, x inBl y
$n_4 \to n_{10}$	*productType*	$q(x,y)$:- x rdf:type Product, x rdf:type y

Table 1: Labels and queries of some nodes and edges of the analytical schema (AnS) shown in Figure 7.

From now on, to simplify the presentation, we assume that through λ, each node in the AnS defines a *new* class (not present in the original graph G), while each edge defines a new property[2]. Just as an analytical schema defines (and delimits) the data available to the analyst in a typical relational data warehouse scenario, in our framework, *the classes and properties modeled by an AnS (and labeled by λ) 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 3.2 will show). Example 6 introduces an AnS for the RDF graph in Section 2.1.

Example 6 (Analytical Schema) *Figure 7 depicts an AnS $\mathcal{S} = \langle \mathcal{N}, \mathcal{E}, \lambda, \delta \rangle$ for analyzing people and products. The node and edge labels appear in the figure, while (some of) the BGP queries defining these nodes and edges are provided in Table 1; the others are very similar. In Figure 7 a person (n_1) may have written messages ($n_1 \to n_2$) which appear on some blog ($n_2 \to n_3$). A person may also have bought products $(n_1 \to n_4)_1$ or may have commented on them $(n_1 \to n_4)_2$. The semantics for the rest of the schema can be easily derived from the figure.*

The nodes and edges of the AnS shown in Figure 7 are those considered of interest for our sample data analysis scenario. In other words, the AnS offers a perspective (or lens) through which to analyze an RDF database. This is formalized as follows:

[2]In practice, nothing prevents λ from returning URIs of class/properties from G and/or the RDF model, e.g., rdf:type etc.

$$\mathcal{I}(\mathcal{S}, \mathtt{G}') = \begin{array}{ll} \{\text{user}_1 \text{ rdf:type person}, & n_1 \\ \text{user}_2 \text{ rdf:type person}, & n_1 \\ \text{user}_3 \text{ rdf:type person}, & n_1 \\ \text{user}_1 \text{ acquaintance user}_2, & n_1 \to n_1 \\ \text{user}_1 \text{ acquaintance user}_3, & n_1 \to n_1 \\ \text{post}_1 \text{ messgInBlog blog}_1, & n_2 \to n_3 \\ \text{product}_1 \text{ rdf:type product}, & n_4 \\ \text{user}_1 \text{ personAge “28”}, & n_1 \to n_5 \\ \text{user}_1 \text{ personName “Bill”}, & n_1 \to n_6 \\ \text{Notebook rdf:type pType}, & n_{10} \\ \text{SmartPhone rdf:type pType}, & n_{10} \\ \text{product}_1 \text{ pPrice “\$400”}, \ldots\} & n_4 \to n_7 \end{array}$$

Table 2: Partial instance of the *AnS* in Figure 7.

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 $\mathcal{I}(\mathcal{S}, \mathtt{G})$ defined as:*

$$\bigcup_{n \in \mathcal{N}} \{s \text{ rdf:type } \lambda(n) \mid s \in q(\mathtt{G}^\infty) \wedge q = \delta(n)\} \cup$$
$$\bigcup_{n_1 \to n_2 \in \mathcal{E}} \{s \ \lambda(n_1 \to n_2) \ o \mid s, o \in q(\mathtt{G}^\infty) \wedge q = \delta(n_1 \to n_2)\}.$$

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

Example 7 (Analytical Schema Instance) *Table 2 shows part of the instance of the analytical schema introduced in Example 6. For each triple, we indicate at right the node (or edge) of the AnS which has produced it.*

Crucial to our ability to handle RDF heterogeneity is *the disjunctive semantics of an AnS*, materialized by the two levels of $\cup$ operators in Definition 5. *Each node and each edge of an AnS populates $\mathcal{I}$ through an independent query*, and the resulting triples are simply combined through unions. This has two benefits: (*i*) significant flexibility when designing the *AnS*, and (*ii*) the ability to build a (heterogeneous) data warehouse on top of a heterogeneous RDF graph.

Consider for instance the three users in the original graph G (Figure 3) and their properties: user_1, user_2 and user_3 are part of the person 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 $n_1 \to n_2$, $n_1 \to n_8$ etc. Thus, the inherent heterogeneity of RDF graphs is accepted in the base data and present in the *AnS* instance.

Defining analytical schemas. Just as in relational data warehouses, our approach requires the analyst to define the analytical schema, which in our context means picking the δ and λ queries associated to each *AnS* node and edge, respectively. Experimenting with the prototype implementing our approach, we have found the following simple method for devising the *AnS*. First, a *default AnS* is automatically created by (*i*) building an edge for each distinct property value in G, and (*ii*) building a node for each distinct domain and range of an edge obtained in (*i*). Based on this default *AnS*, we can inspect nodes and edges, for instance starting with the ones that are most populated (have the largest number of instances) and build with the help of our tool's GUI, more complex

queries to define a refined AnS, typically having fewer nodes and edges than the default one, but with more complex node and edge definitions.

On the instances of analytical schemas. For simplicity, an AnS uses unary and binary BGP queries (introduced in Section 2.2) to define its instance, as the union of every AnS node/class and edge/property instance. This can be extended straightforwardly to unary and binary (full) SPARQL queries 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 from distributed heterogeneous sources.

3.2 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[3] 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 $n \to n' \in \mathcal{E}$: (i) $h(n) \to h(n') \in \mathcal{E}'$ and (ii) $\lambda(n \to n') = \lambda'(h(n) \to h(n'))$ or $\lambda(n \to n')$ is a variable;*
- *for every $n_1 \to n_2, n_3 \to n_4 \in \mathcal{E}$, if $\lambda(n_1 \to n_2) = \lambda(n_3 \to n_4)$ is a variable, then $h(n_1 \to n_2) = h(n_3 \to n_4)$;*
- *for $n \in \mathcal{N}$ and $n_1 \to n_2 \in \mathcal{E}$, $\lambda(n) \neq \lambda(n_1 \to n_2)$.*

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 occurences 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.

[3]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.

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 core concepts known from the relational data warehouse literature, 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. It is worth noticing that, 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 $x\ \lambda(r \to n)\ o$ with $r \to n \in \mathcal{E}$. 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 next query asks for the number of blogs where the user posts, classified by the user age and city:*

$$\langle c(x, a, c), m(x, b), count \rangle$$

where the classifier and measure queries are defined by:

$$c(x, a, c) \text{:- } x\ personAge\ a, x\ personCity\ c$$
$$m(x, b) \text{:- } x\ wroteMessg\ o, o\ messgInBlog\ b$$

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 the type of each value returned by $q^j(\mathcal{I})$ belongs (or can be converted by the SPARQL rules [29]) 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 well-defined.

Example 9 (Analytical Query Answer) *Consider the query in Example 8, over the analytical schema in Figure 7. Some triples from the instance of this analytical schema were shown in Table 2. The classifier query returns:*

$$\{\langle \text{user}_1, 28, \text{“}Madrid\text{”}\rangle, \langle \text{user}_3, 35, \text{“}NY\text{”}\rangle\}$$

while the measure query returns:

$$\{\langle \text{user}_1, \text{blog}_1\rangle, \langle \text{user}_1, \text{blog}_2\rangle, \langle \text{user}_2, \text{blog}_2\rangle, \langle \text{user}_3, \text{blog}_2\rangle\}$$

Aggregating the blogs among the classification dimensions leads to the AnQ answer:

$$\{\langle 28, \text{“}Madrid\text{”}, 2\rangle, \langle 35, \text{“}NY\text{”}, 1\rangle\}$$

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

4 Analytical query evaluation

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. Then, to evaluate an *AnQ*, one simply delegates the evaluation of the classifier and measure queries, and of the final aggregation, to the RDF-DM. 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 propose an alternative solution. The idea is to reformulate the *AnQ* based on the *AnS* definition so that evaluating the reformulated query, returns exactly the same answer as if materialization was used. Using reformulation, one can store the original RDF graph into an RDF-DM, and delegate the reformulated query evaluation to the RDF-DM.

The technique builds on the reformulation of BGP queries, lying at the core of *AnQ*s, w.r.t. analytical schemas:

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 $q^{\bowtie}_{\mathcal{S}} = \bigcup_{h\in\mathcal{H}} q^{\bowtie}_h(\bar{x})$:- $\bigwedge_{i=1}^{m} q_i(\bar{x}_i)$ such that:*

- *for each triple $t_i \in q$ of the form* s rdf:type $\lambda'(n_i)$, *$q_i(\bar{x}_i)$ in $q^{\bowtie}_h$ 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 $\lambda'(n_i \rightarrow n_j)$ o, *$q_i(\bar{x}_i)$ in $q^{\bowtie}_h$ is defined as $q_i = \delta(h(n_i \rightarrow n_j))$ and $\bar{x}_i =$* s, o.

This definition basically says that, for a BGP query meaningful w.r.t. an AnS (there is at least an homomorphism from the query to the AnS), the reformulated query amounts to translating all its possible interpretations w.r.t. the AnS (modeled by all the homomorphism 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 the AnS is wrapped, using its node queries.

Example 10 (AnQ REFORMULATION) *Let* $q(x, o, s)$ *be:*

$$q(x, o, s) \text{:- } x \text{ rdf:type person}, x \text{ wroteMessg } o, o \text{ messgInBlog } s$$

The query q uses the vocabulary of the AnS and is meant to be evaluated over its instance $\mathcal{I}$. In the following, we reformulate q into $q^{\bowtie}_{\mathcal{S}}$, which uses the vocabulary of the initial graph `G` *and can be evaluated over it to obtain the same result as q over $\mathcal{I}$.*

The first atom x rdf:type person *in q is of the form* `s` rdf:type $\lambda(n_1)$*, for the node n_1 in the AnS in Figure 7. Consequently, $q^{\bowtie}_{\mathcal{S}}$ contains as a conjunct the query $q(x)$:- x* rdf:type *$Person$ (obtained from $\delta(n_1)$ in Table 1)*[4]*.*

The second atom in q is of the form `s` $\lambda(n_1 \rightarrow n_2)$ `o` *for the* person *node n_1 and the* message *node n_2 in Figure 7, while the query labeling $n_1 \rightarrow n_2$ is: $q(x, y)$:- x* wrote *y, y* rdf:type *$Message$. As a result, $q^{\bowtie}_{\mathcal{S}}$ contains the conjunct $q(x, o)$:- x* wrote *o, o* rdf:type *$Message$.*

Similarly, q's last atom corresponds to the edge $n_2 \rightarrow n_3$ in the AnS; it adds the conjunct $q(o, s)$:- o rdf:type Message, *o* inBl *s to $q^{\bowtie}_{\mathcal{S}}$. Thus, the reformulated query amounts to:*

$$q^{\bowtie}_{\mathcal{S}}(x, o, s) \text{:- } x \text{ rdf:type } Person, x \text{ wrote } o, o \text{ rdf:type } Message, o \text{ inBl } s$$

which can be evaluated directly on the graph `G` *in Figure 3.*

The following theorem 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 $\mathcal{S}$, and $c^{\bowtie}_{\mathcal{S}}$ be the reformulation of Q's classifier query against $\mathcal{S}$. We have:*

$$ans(Q, \mathcal{I}) = \{\langle d^j_1, \ldots, d^j_n, \oplus(q^{j\bowtie}_{\mathcal{S}}(\mathtt{G}^{\infty}))\rangle \mid \langle x^j, d^j_1, \ldots, d^j_n\rangle \in c^{\bowtie}_{\mathcal{S}}(\mathtt{G}^{\infty}) \text{ and } q^j \text{ is defined as } q^j(v) \text{:- } m(x^j, v)\}$$

assuming that the type of each value returned by $q^{j\bowtie}_{\mathcal{S}}(\mathtt{G}^{\infty})$ belongs (or can be converted by the SPARQL rules [29]) to the input type of the aggregator $\oplus$. Otherwise, the answer set is undefined.

The above theorem states that in order to answer Q on $\mathcal{I}$, one first needs to reformulate Q's classifier, and answer it *directly against* `G` (and not against $\mathcal{I}$ as in Definition 8): this is how reformulation avoids materializing the analytical schema instance. Then, for each tuple $(x^j, d^j_1, \ldots, d^j_n)$ returned by the classifier, the following steps are needed: instantiating the measure query m with the fact x^j leading to the query q^j, reformulating q^j w.r.t. $\mathcal{S}$ into $q^{j\bowtie}_{\mathcal{S}}$, answering the latter again against `G`, and, finally, aggregating its results through $\oplus$. The theorem follows quite directly by showing that two-way inclusion holds between the two terms.

[4]Recall that the person type used in q is part of the AnS (Figure 7), whereas the Person type belongs to the original RDF graph `G` (Figure 5).

5 OLAP RDF analytics

On-Line Analytical Processing (OLAP) [2] technologies enhance the abilities of data warehouses (usually relational) to answer multi-dimensional analytical queries.

The analytical model we introduced so far is specifically designed for graph-structured, heterogeneous RDF data. In this section, we demonstrate that our model is expressive enough to support RDF-specific counterparts of all the traditional OLAP concepts and tools known from the relational data warehouse 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 7. 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. Let Σ 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:

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

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 Σ 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 $c_\Sigma(\mathcal{I})$ (containing all tuples returned by included single classifier). This highlights that *an extended analytical query can be seen as a union of a set of standard AnQs*, 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 Σ 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 concrete value.

Example 11 (Slice) *Let Q be extended query corresponding to the query-cube defined in Example 8, that is:*

$$\langle c_\Sigma(x, a, c), m(x, y), count \rangle$$

with $\Sigma = \{(a, \{a\}),\ (c, \{c\})\}$ (classifier and measure queries are as in Example 8). A slice operation on the age dimension a with value 34 *results in replacing the extended classifier of Q with:*

$$c_{\Sigma'}(x, a, c) = \{c(x, 34,\ c)\}$$

where $\Sigma' = \Sigma \setminus \{(a, \{a\})\} \cup \{(a, \{34\})\}$.

Dice. Similarly, a dice operation on Q and 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:

$$\Sigma' = \Sigma \setminus (\cup_1^k\{(d_j, \Sigma(d_j)\}) \cup (\cup_1^k\{(d_j, S_j)\})$$

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

Example 12 (Dice) *Consider again the initial cube Q from Example 11, and a dice operation on both age and city dimensions with values $\{34\}$ for b and $\{\text{Madrid}, \text{Kyoto}\}$ for c. The dice operation replaces the extended quantifier of Q with $c_{\Sigma'}(x, a, c)$, consisting of:*

$$\{c(x, 28, \text{“Madrid”}),\ c(x, 28, \text{“Kyoto”})\}$$

where:

$$\Sigma' \ = \Sigma \setminus \{(a, \{a\})\,(c, \{c\})\} \cup \{(a, \{34\}),\ (s, \{\text{“Madrid”},\ \text{“Kyoto”}\})\}$$

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.

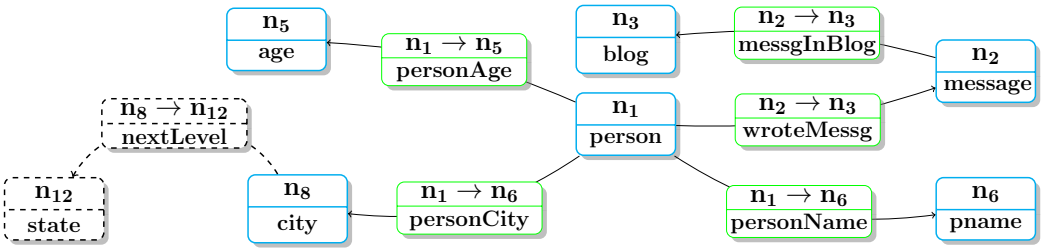
Example 13 (Drill-in) *Consider the cube Q from Example 11, and a drill-in on the age dimension. The drill-in rewriting produces the query:*

$$Q = \langle c'_\Sigma(x, c), m(x, y), count\rangle$$

with $\Sigma = \{(c, \{c\})\}$ *and* $c'(x, c) = x$ *city* c.

Dimension hierarchies. Typical relational warehousing scenarios feature hierarchical dimensions, e.g., a value of the *country* dimension corresponds to several *region*s, each of which contains many *cities* etc. Such hierarchies were not considered in our framework so far[5].

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 fragment of the AnS in Figure 7 relevant for our sample analytical query in Example 8, as shown below (new nodes and edges are dashed):



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

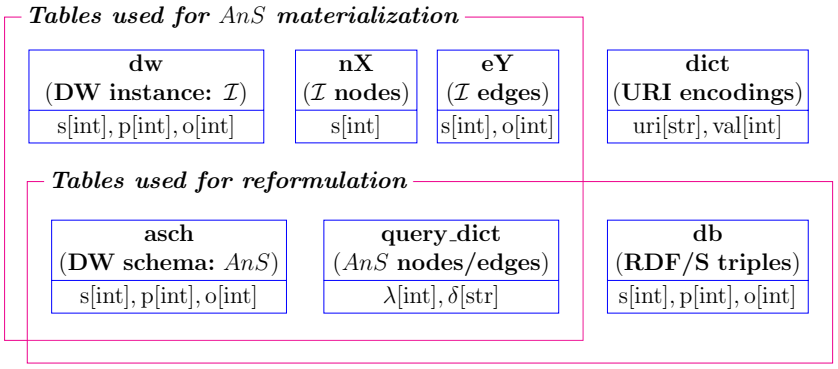


Figure 8: Data layout of the RDF warehouse.

G size	schema size	dictionary	$\mathtt{G}^{\infty}$ size
3.4×10^7 triples, 4.4 GB	5.5×10^3 triples, 746 KB	7×10^6 entries	3.8×10^7 triples

Table 3: Dataset characteristics.

In a similar fashion one could add use the **nextLevel** property to support a hierarchy among edges. For instance, one could state that 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 add to / remove from the classifier, triple atoms navigating such **nextLevel** edges.

Example 14 (ROLL-UP) *Recall the cube query in Example 8. A roll-up along the* city *dimension to the* state *level yields:*

$$\langle c_{\Sigma}(x, a, s), m(x, y), count \rangle$$

where:

$$c_{\Sigma}(x, a, s) \text{:- } x\ personAge\ a,\ x\ personCity\ c,\ c\ nextLevel\ s.$$

The measure component remains the same, and Σ *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.*

6 Experiments

We performed a set of experiments with our RDF analytical framework. Section 6.1 outlines our implementation and experimental settings. We describe experiments we carried to build *AnS* instances in Section 6.2, evaluate *AnQ*s in Section 6.3, discuss query reformulations in Section 6.4 and OLAP operations in Section 6.5, then we conclude.

6.1 Implementation and settings

Underlying platform: kdb+. Our RDF analytics approach can be deployed on top of any system capable of storing and querying RDF triples. We chose to use kdb+ v3.0 (64 bits) [1], 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 choose to materialize it. We translate BGP queries into q programs that kdb+ interprets.

Data organization. Figure 8 outlines our data layout in kdb+. The URIs within the RDF data set are encoded using integers; the mapping is preserved in a q dictionary data structure, named **dict**. The saturation of G, denoted $\mathtt{G}^{\infty}$ (Section 2.2), is stored in the **db** table. Analytical schema *definitions* are stored as follows. The **asch** table stores the analytical schema triples $\lambda(n_1)\ \lambda(n_1 \rightarrow n_2)\ \lambda(n_2)$ for all the nodes n_1, n_2 and edges $n_1 \rightarrow n_2 \in AnS$. The separate **query_dict** dictionary maps the encoded λ values for nodes and edges to their corresponding δ queries. Finally, we use the **dw** table to store the AnS instance $\mathcal{I}$, *or* i tables named **nX** and j tables named **eY** if a partitioned-table storage is used (see Section 6.2), where i, j are the number of nodes, respectively edges, in the AnS and X, Y uniquely identify the node, respectively edge, data stored in the table. 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 traditional query optimizer in the database sense. 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 triples from DBpedia, more specifically, three *ontology infobox* datasets and the RDFS schema from the version (http://dbpedia.org/Download38). The main characteristics of the data appear in Table 3. For our *scalability experiments* (Figures 10 and 12), we replicated the dataset several times (leading to doubling, tripling etc. the size of the analytical schema instance; see Section 6.2).

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.

6.2 AnS materialization

We loaded the (unsaturated) G in about 3 minutes, and we computed its full saturation $\mathtt{G}^{\infty}$ in 22 minutes. We specified 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.

We considered two ways of materializing the instance $\mathcal{I}$. First, we used a single table (**dw** in Figure 8). Second, inspired from RDF stores such as [14], we tested a partitioned data layout for $\mathcal{I}$ as follows. For each distinct node (modeling triples of the form s rdf:type λ_X), we store a table with the subjects s declared of that type (**nX** with $X \in [1, 26]$). Further, for each distinct edge (s λ_Y o) a separate table stores the corresponding triple subjects and objects (**eY** with $Y \in [1, 75]$).

Figure 9 shows for each node and edge query (labeled on the y axis by λ, chosen based on the name of a "central" class or property in the query[6]): (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),*

[6]The dbpo, xsd and warg namespaces respectively correspond to DBpedia, XML Schema and the AnS we defined.

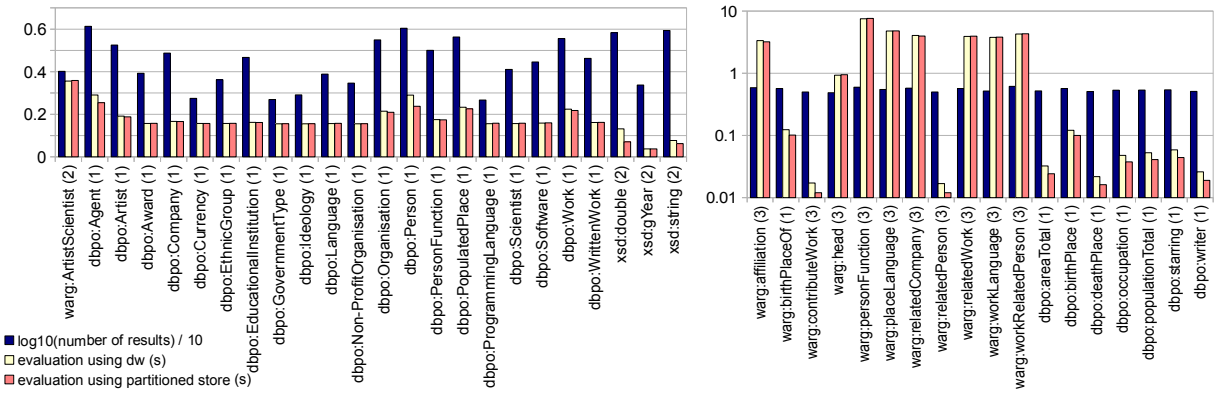


Figure 9: Evaluation time (s) and number of results for AnS node queries (left) and edge queries (right).

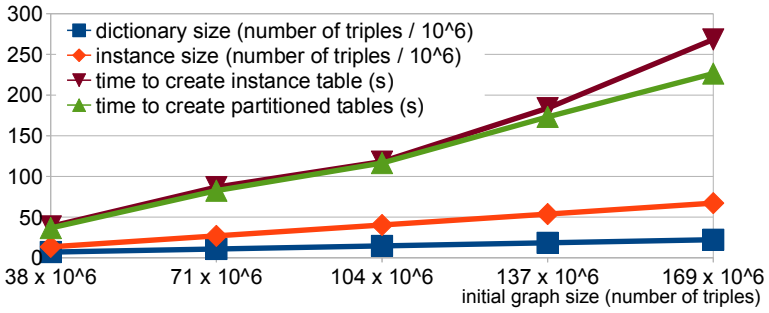


Figure 10: $\mathcal{I}$ materialization time vs. $\mathcal{I}$ size.

and we omitted them from the plots. The *total time to materialize* the instance $\mathcal{I}$ (1.3×10^7 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 is shown in Figure 10, demonstrating linear scale-up w.r.t. the data size.

6.3 AnQ evaluation over $\mathcal{I}$

We consider a set of AnQs, 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 11 shows for each query pattern, the number of queries in the set (in parenthesis after 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 11 shows the average, minimum and maximum query evaluation time among the queries of each pattern.

As can be seen in the figure, the query result size (up to hundreds of thousands) is the most strongly correlated with the query evaluation time. The other parameters impacting

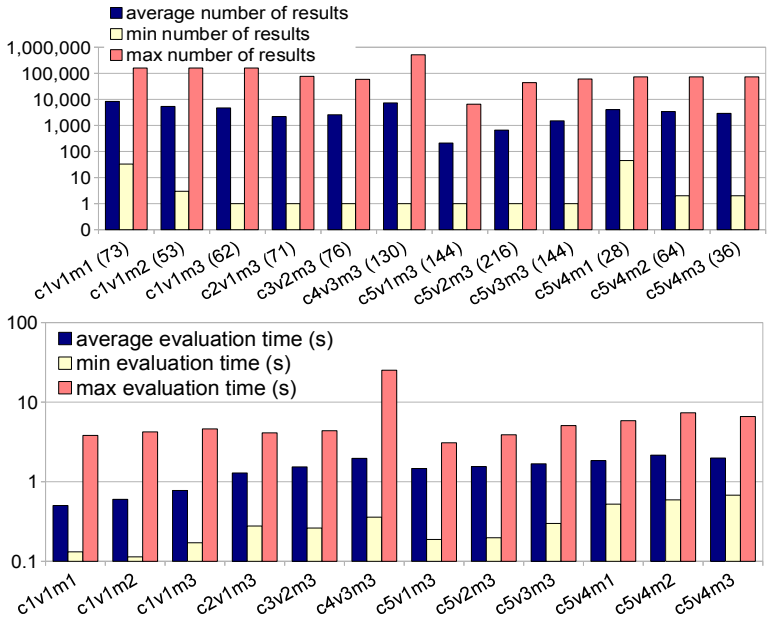


Figure 11: *AnQ* statistics for query patterns.

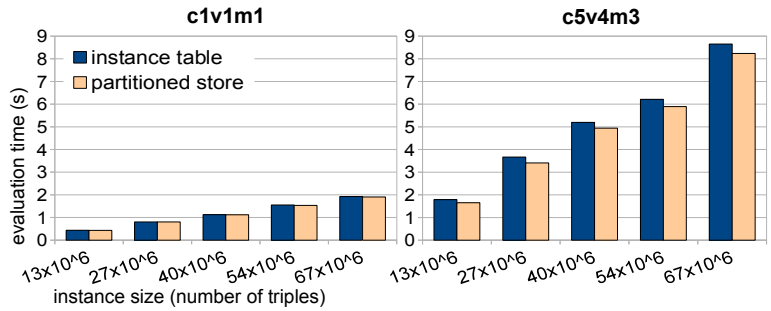


Figure 12: *AnQ* evaluation time over large datasets.

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 quite robust performance even for complex *AnQ*s.

Figure 12 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.

6.4 Query reformulation

We now study the performance of *AnQ* evaluation through reformulation (Section 4), through a set of 32 queries mathcing the pattern c1v1m1.

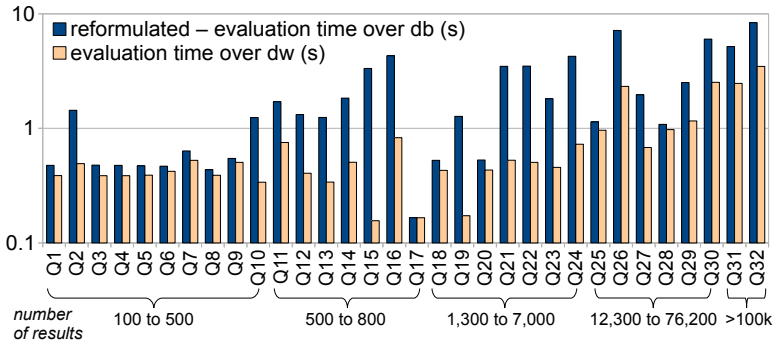


Figure 13: AnQ reformulation.

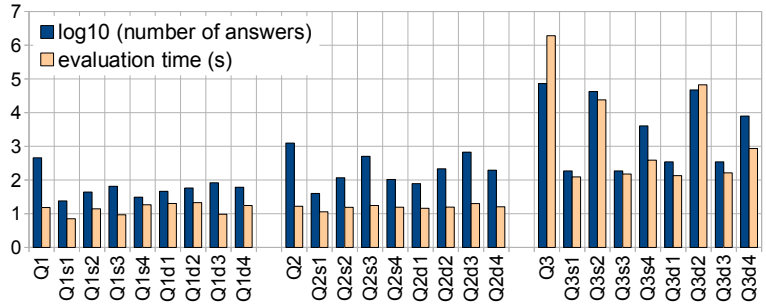


Figure 14: Slice and dice over AnQs.

Figure 13 shows for each query, the number of answers (under the chart), the evaluation time over **db** when reformulated and the evaluation time over $\mathcal{I}$. As expected, reformulation-based evaluation is slower, because reformulated queries have to re-do some of the AnS materialization work. It turns out that the queries for which the difference is largest (such as Q_{15}, Q_{16} or Q_{19}) are those whose reformulation against the AnS definition have the largest numbers of atoms, one or more of which are of the form $x\ y\ z$. Evaluating complex joins including those of this form (matching all **dw**) is expensive, compared to evaluating them on the materialized $\mathcal{I}$. However, the extra-time incurred by query reformulation can be seen as the price to pay to avoid AnS's instance maintenance time upon base data updates.

6.5 OLAP operations

We now study the performance of OLAP operations on AnQ results (Section 5).

Slice and dice. In Figure 14, 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 of 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 quite 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 5), it follows that the queries in c5v**n**m3 are drill-ins of c5v**(n+1)**m3 for 1≤n≤3, and conversely, c5v**(n+1)**m3 result from drill-out on c5v**n**m3. Their evaluation times appear in Figure 11 (1-2 seconds on average).

OLAP operations evaluated on *AnQ*s. The OLAP operations described so far were applied on *AnQ*s and evaluated from scratch (against the database **dw**). Alternatively, we experimented with OLAP operations applied directly on the materialized results of previous analytical queries. As expected, applying for instance the slice operations on Q_1-Q_3 in Figure 14 was faster by 1-3 orders of magnitude than evaluation on the original graph, since the input data is much smaller (and the regular-structure *AnQ* results are easier to deal with than the original heterogeneous graph).

6.6 Conclusion of the experiments

Our experiments have demonstrated the feasibility of deploying our full RDF warehousing approach on top of a concrete system, in our case based on kdb+; thanks to the genericity of our proposal and its reliance on standard RDF queries, any system providing a triple store, conjunctive query evaluation, and possibly reasoning can be easily used instead. Our experiments have demonstrated robust scalable performance when loading and saturating $\mathtt{G}$, building $\mathcal{I}$ in time linear in the input size (even though it includes complex, many-joins node and edge queries), finally we have shown that OLAP operations in our framework are evaluated efficiently based on the database, and extremely fast based on already-materialized *AnQ*s (equivalent of cubes in our setting). Our experiments have also confirmed that *AnQ* evaluation is faster based on a materialized analytical schema instance, than by reformulation against the *AnS* definition, as in the traditional DW setting. While further optimizations could be certainly applied at many points (as in traditional DW platforms), our experiments confirmed the interest and good performance of our proposed all-RDF Semantic Web warehousing approach.

7 Related Works and Discussion

Relational DWs have been thoroughly studied [17], and many efficient tools exist. Warehouses for Web data have been defined as repositories of interconnected XML document and Web services [4], or as distributed knowledge bases specified in a rule-based language [5]. In [23], a large RDF knowledge base (such as Yago [25]) is enriched with information gathered from the Web. None of these works considered RDF analytics.

RDF(S) vocabularies (pre-defined classes and properties) have been proposed for describing *relational* multi- dimensional data and cubes [28, 10] in RDF; [10] also maps OLAP operations into SPARQL queries over cubes described in the vocabulary. In contrast, we introduce analytical schemas and queries based on which one can *define RDF cubes* over heterogeneous, semantic-rich RDF graphs; our approach is not tied to any vocabulary. Going in the opposite direction, [19] presents a semi-automated approach for deriving a *relational* DW from a Semantic Web domain ontology. In contrast, our approach is *all-RDF*: the analytical schema instance is RDF, thus analytical queries can still exploit rich RDF semantics, e.g., reasoning and joint querying of the schema and data.

In the area of RDF data management, recent works tackled storage [3, 24], indexing [30], query processing [21], updates [22], cardinality estimations [20], materialized views [11], and Map-Reduce based RDF processing [13, 14]; among the industrial systems, Oracle 11g provides a "Semantic Graph" extension etc. In this context, *our work is the first work to formalize RDF analytics and propose analytic schema and queries with well-defined semantics*. As we have shown, these concepts can be efficiently implemented on top of a conjunctive query processing engine, or an RDF data management system, extending their functionality and enabling users to fully take advantage of the information comprised in their data.

Recent works have focused on graph warehousing and thus are related to our proposal. The model of [31] first introduced the idea of defining the nodes and edges of the analytical schema ("graph cube" in their terminology) through independent queries (the aggregation applied to edges is independent of that applied to nodes); further, their OLAP manipulations of graph cubes bear some similarity to our corresponding notions in Section 5. However, their approach was not meant for *heterogeneous* graphs, and thus it cannot handle multi-valued attributes, e.g., a movie being both a comedy and a romance; in contrast, our models handles this naturally. Moreover, unlike our approach, their model does not attach labels to edges, and does not consider graphs with semantics (such as RDF graphs). Also, this approach only focuses on *counting edges* between various groups of nodes, whereas our framework captures aggregation in a more general, database-style interpretation (where one has the choice between many kinds of aggregation), yet being rich with RDF semantics.

The work of [8] aims at extending DW and business intelligence operations to graphs. They do not consider the semantically rich RDF model, but rather encode geo-spatial informations: nodes are connected via paths segments, each of which has a cost related to the distance between nodes in the graph. Their analytical schemas consist of nodes and paths; aggregation is performed over *records*, which are subgraphs of the analytical schema.We share with [8] the principle of separation between schema and data. However, in our model edges are assigned diverse *meanings*, different from the discussed work where an edge models solely a *measure* (an assigned cost). The analytical query definitions also share some general principles such as the usage of path queries in an OLAP framework. However the diverse meanings assigned to our edges enable us to define multiple measures on which to perform aggregations, while in [8] aggregation can be done only on the path costs. Their interpretation of roll-up operations is also very different from ours: in their framework, a roll-up collapses neighbor nodes into one, based strictly on the proximity criteria, e.g., collapsing several locations or cities, into a single node representing that geographical area.

In [9], the authors consider multidimensional modeling using the Object-Oriented paradigm. Conceptually, their *complex objects* and *complex relationships* are similar to our *AnS* nodes and edges, respectively defining new entities and potential analysis axes. While their complex relationships rely on already defined relationships between classes, our edges are defined independently from the nodes they connect. In both works the dimensions and facts are not static or predefined and can be selected at data cube creation time. In [9] hierarchies are defined through two modeling concepts (*attribute hierarchy* and *object hierarchy*) distinct from the complex relationships, while in our case hierarchies are represented as a special type of edges.

The separation between grouping and aggregation present in our analytical queries is similar to the MD-join operator introduced in [7] for relational DWs.

Finally, the recent SPARQL 1.1 language [29] includes group-by and aggregation constructs closely inspired from SQL. Efficient RDF data management platforms supporting it will be ideal candidates for deploying our full-RDF analytics chain, providing analytical schemas and queries, and navigation within RDF cubes through OLAP-style operations.

8 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.

References

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

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

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

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

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

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

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

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

[9] D. Boukraâ, O. Boussaïd, F. Bentayeb, and D. E. Zegour. A layered multidimensional model of complex objects. In C. Salinesi, M. C. Norrie, and O. Pastor, editors, *CAiSE*, volume 7908 of *Lecture Notes in Computer Science*, pages 498–513. Springer, 2013.

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

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

[12] V. Harinarayan, A. Rajaraman, and J. D. Ullman. Implementing data cubes efficiently. In *SIGMOD*, 1996.

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

[14] 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.

[15] W. H. Immon. *Building the Data Warehouse*. Wiley, 1992.

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

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

[18] R. Kimball and M. Ross. *The Data Warehouse Toolkit*. Wiley, 2002. 2nd Edition.

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

[20] T. Neumann and G. Moerkotte. Characteristic sets: Accurate cardinality estimation for RDF queries with multiple joins. In *ICDE*, 2011.

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

[22] T. Neumann and G. Weikum. x-RDF-3X: Fast querying, high update rates, and consistency for RDF databases. *PVLDB*, 3(1), 2010.

[23] 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.

[24] 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.

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

[26] D. Theodoratos and T. K. Sellis. Data warehouse configuration. In *VLDB*, 1997.

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

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

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

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

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