Abstract—This paper presents Nyaya, a flexible system for the management of Semantic-Web data which couples an efficient storage mechanism with advanced and up-to-date ontology reasoning capabilities. Nyaya is capable of processing large Semantic-Web datasets, expressed in a variety of formalisms, by transforming them into a collection of Semantic Data Kiosks that expose the native meta-data in a uniform fashion using Datalog\(^2\), a very general rule-based language. The kiosks form a Semantic Data Market, where the data in each kiosk can be uniformly accessed using conjunctive queries and where users can specify user-defined constraints over the data.

Nyaya is easily extensible and robust to updates of both data and meta-data in the kiosk. In particular, a new kiosk of the semantic data market can be easily built from a fresh Semantic-Web source expressed in whatsoever format by extracting its constraints and importing its data. In this way, the new content is promptly available to the users of the system. The approach has been experimented using well known benchmarks with very promising results.

I. INTRODUCTION

Ever since Tim Berners Lee presented, in 2006, the design principles for Linked Open Data\(^1\), the public availability of Semantic-Web data has grown rapidly. Today, practitioners, organizations and universities are all contributing to what is called the Web of Data by building RDF repositories either from scratch or transforming data stored in traditional formats [1]. In this trend, ontology languages such as RDFS and OWL (with all their relatives and versions) provide a means to annotate data with meta-data (constraints forming a TBox, in knowledge base terminology) enabling different forms of reasoning, depending on the expressiveness of the adopted language.

However, this spontaneous growth of available data is not yet supported by sufficiently efficient and equally general management systems, since each dataset is usually processed within a language-dependent (and usually rigid) management system that implements the specific reasoning and query-processing algorithms of the language used to specify the constraints. For these reasons the problem of storing, reasoning over, and querying large Semantic-Web datasets in a flexible and efficient way represents a challenging area of research and a profitable opportunity for industry [2].

In this paper we present Nyaya\(^2\), a system for Semantic-Web data management which provides (i) an efficient and general storage policy for Semantic-Web datasets, along with (ii) advanced and language-independent ontology reasoning capabilities and (iii) the possibility to express conjunctive queries and user-defined constraints over the dataset depending on the application needs. Reasoning and querying in Nyaya is based on Datalog\(^\pm\) [3], a very general logic-based language that captures the most common tractable ontology languages and provides efficiently-checkable, syntactic conditions for decidability and tractability.

To show how Nyaya works, we refer to the conceptual architecture reported in Figure 1. This architecture is not meant to define an actual implementation strategy. Rather, it aims to illustrate interactions among the main features of our system. The whole framework relies on a collection of persistent repositories of Semantic-Web data that we have called kiosks. A kiosk is populated by importing an available RDF dataset, possibly coupled with meta-data that specify constraints on its content using a generic Semantic-Web language such as RDF(S) and OWL (or variants thereof). In order to allow inferencing, “native” vocabularies and meta-data are extracted and represented as Datalog\(^\pm\) constraints. In addition, Nyaya adopts non-recursive Storage Programs expressed in Datalog in order to link the vocabulary entities of a kiosk to their actual representation in the storage. This also enables a form of separation of concerns between the reasoning and query-processing algorithms and the logical organization of the storage that can be changed without affecting querying. On the opposite side, each kiosk exposes a uniform query interface to the stored data and allows access to the meta-data.

A collection of kiosks constitutes what we have called a Semantic Data Market, a place that exposes the content of all the kiosks in a uniform way and where users can issue queries and collect results, possibly by specifying further constraints over the available data using Datalog\(^\pm\) in addition to the original ones.

Users operate in the semantic data market using a high level front-end (e.g., a SPARQL Endpoint\(^3\) or a visual Query

\(^1\)http://linkeddata.org/

\(^2\)Nyaya, literally “recursion” in Sanskrit, is the the name of the school of logic in the Hindu philosophy.

\(^3\)http://semanticweb.org/wiki/SPARQL_endpoint
by Example interface). The user query is translated into a Datalog\pm query and reformulated in terms of the Datalog\pm constraints. Since NYAYA focuses on FO-reducible constraints, the result of the translation process is a union of conjunctive queries (UCQ). In a subsequent step the UCQ is reformulated in terms of the storage programs of each kiosk obtaining a set of UCQ (one for each kiosk) that takes into account the relationship between the entities occurring in the query and the structures in the persistent storage of each kiosk. Each UCQ is then easily rewritable into actual queries over the persistent storage system (SQL in our current relational implementation) that retrieve the tuples answering the original query.

An important aspect of this framework is that it is easily extensible: when a fresh Semantic-Web source expressed in whatsoever format is made available, a new kiosk can be easily built from it by extracting its meta-data and importing its data. In this way, the content of the source is promptly available to the users of the system. Conversely, if a user wants to query the same kiosk by adopting a different set of constraints, the queries are automatically reformulated accordingly and issued to the kiosk.

In sum, the main contributions of this work are the following:
- the definition of Semantic Data Market, an extension of standard Knowledge Bases where the ontological constraints and the logical organization of persistent information are uniformly represented in the same language,
- an algorithm for rewriting-based access to Kiosks of the semantic data market,
- an efficient storage and model-independent mechanism for Semantic-Web data, coupled with a rule-based inference and query-answering engine.

The paper is organized as follows: Section II discusses some related work, while Section III describes the architecture of NYAYA and the storage technique we use, based on metamodeling. Section IV introduces the fundamental theoretical notions underlying reasoning and query-processing algorithms in NYAYA, subsequently presented in Section V. Section VI describes the experimental setting and gives the performance figures. Finally, Section VII draws the conclusions and delineates our future work.

II. RELATED WORK

Managing Semantic-Web data represents an important and profitable area of research. We now briefly discuss some representative systems, focusing on the two major issues concerning Semantic-Web data management, namely: storage and querying.

Considering the storage, we identify two main trends: (i) the first focuses on developing native storage systems to exploit ad-hoc optimisations while (ii) the second adapts mature data-models (relational, object-oriented, graph [4]) to Semantic-Web data. Generally speaking, native storage systems (such as AllegroGraph\(^4\) or OWLIM\(^5\)) are more efficient in terms of load and update time. On the other side, relying on an existing data-management system improves query efficiency due to the availability of mature and effective optimisations. In this respect, a drawback of native approaches consists exactly in the need for re-thinking query optimizations and transaction processing techniques.

Early approaches to Semantic-Web data management (e.g., RDFSuite\(^5\), Sesame\(^6\), KAON\(^7\), TAP\(^8\), Jena\(^9\), Virtuoso\(^10\) and the semantic extensions implemented in Oracle Database 11g R2 [6] focused on the triple (subject, predicate, object) or quads (graph, subject, predicate, object) data models, each of which can be easily implemented as one large relational table with three/four columns. Queries are formulated in an RDF(S)-specific query language (e.g., SPARQL\(^11\), SeRQL\(^12\), RQL\(^13\) or RDQL\(^14\)), translated to SQL and sent to the RDBMS. However, during query-processing the number of self-joins that

\(^4\)http://www.franz.com/products/allegrograph/
\(^5\)http://www.ontotext.com/owlim/
\(^6\)http://www.openrdf.org/
\(^7\)http://kaon.semanticweb.org/
\(^8\)http://tap.stanford.edu/
\(^9\)http://jena.sourceforge.net/
\(^10\)http://virtuoso.openlinksw.com/
\(^11\)http://www.w3.org/TR/rdf-sparql-query/.
\(^12\)http://www.openrdf.org/doc/sesame/users/ch06.html
\(^13\)http://139.91.183.30:9090/RDF/RQL/.
\(^14\)http://www.w3.org/Submission/RDQL/.
must be executed makes this approach unfeasible in practice and the optimisations introduced to overcome this problem (such as property-tables [7], GSPO/OGPS indexing [8]) have proven to be highly query-dependent [9] or to introduce significant computational overhead [10]. Abadi et al. [9] proposed the Vertical Partitioning approach, where the triple table is split into two two-column tables in such a way that it is possible to exploit fast merge Joins in order to reconstruct information about multiple properties for subsets of subjects. However, Sidirougos et. al [11], analysed the approach concluding that its hard-coded nature makes it difficult to generalise and use in practice.

On the querying side we analyse the problem w.r.t. two dimensions, namely reasoning support and inference materialisation. Earlier systems did not allow reasoning either because efficient algorithms were not available or because the data language (for example RDF) was too simple to allow inferencing. In such cases the efficiency of query-processing depends only on the logical and physical organization of the data and on the adopted query language. On the other side, reasoning is a fundamental requirement in the Semantic-Web and efficient reasoning algorithms are now available [12], [13]. However, a system that supports reasoning must face the problem of inference materialisation.

A first family of systems materialises all the inferences, this enable ad-hoc indexing and dramatically improves query-processing performance. By contrast, loading-time is inflated because all the complexity of inferencing is deferred to this moment and could potentially generate an exponential blow-up of space occupation. Moreover update-management is also critical since the inferred data must be incrementally updated accordingly. Full materialisation is suitable for really stable datasets, especially when the queries are known in advance. The second class of systems, such as QuOnto [14] and REQUIEM [15], executes inferencing on-the-fly, thus the complexity of query processing depends on both the logical organisation of the data and the expressiveness of the adopted data language and the queries. This approach has many advantages: in the first place, space occupation is reduced to the minimum necessary to store the dataset and it is possible to define a suitable trade-off for inference materialisation depending on which queries are executed most frequently. Another advantage is the robustness to updates: since inferences are computed every time, there is no need for incremental updates. The major drawback of on-the-fly inferencing is the impact of reasoning time on query processing, however, the systems QuOnto and REQUIEM are built upon languages such as DL-Lite [12] and ECHIO™ [16] that trade expressiveness for tractability of query answering through intensional query reformulation [17] or database completion [18]; the latter, however, requires either write-access to the target database or enough main memory to store the produced inferences.

To the best of our knowledge NYAYA is the first system that adopts a flexible combination of tractable reasoning with a highly performant storage system.

III. NYAYA REPRESENTATION OF DATA AND META-DATA

Following an approach for the uniform management of heterogeneous data models [19], data and meta-data are represented in NYAYA using a meta-model \( M \) made of a generic set of constructs, each of which represents a primitive used in known semantic models (e.g., RDF, RDFS and OWL) for describing the domain of interest. The adoption of a meta-model able to describe all the data models of interest has two main advantages. On the one hand, it provides a framework in which semantic models can be handled in a uniform way and, on the other hand, it allows the definition of model-independent reasoning capabilities.

Constructs in \( M \) are chosen by factoring out common primitives in different models. For instance, \( M \) includes the construct CLASS that is used in both RDFS and OWL to represent a set of resources. Each construct is associated with an object identifier, a name, a set of properties, and a set of references to other constructs. The approach is basically independent of the actual storage model (e.g., relational, object-oriented, or XML-based). However, since NYAYA relies on a relational implementation of the meta-model, in the following we assume that each construct correspond to a table.

The core of \( M \) is constituted by the following constructs.

- CLASS(OID, Name)
- DATAPROPERTY(OID, Name, ClassOID, isKey, ClassOID)
- OBJECTPROPERTY(OID, Name, isKey, isNegative, isTransitive, isSymmetric, isFunctional, isInverseFunctional, SubjectClassOID, ObjectClassOID)
- I-CLASS(OID, URI, ClassOID)
- I-DATAPROPERTY(OID, Name, Value, i-ClassOID, Property-OID)
- I-OBJECTPROPERTY(OID, Name, Subject-ClassOID, Object-ClassOID, Property-OID)

The first three tables serves to store sets of resources, atomic properties of resources (i.e., properties whose objects are literals) and object properties of resources (i.e., properties whose objects are resources themselves), respectively. The DATAPROPERTY table has a reference to the CLASS it belongs to and has the range data type (e.g., integer or string) as attribute. The OBJECTPROPERTY construct is used to represent RDF statements and has two references to the subject class and the object class involved in the statement. In addition, it has attributes that specify notably features of the property. The I-CLASS construct is used to model resources that are instances of the class specified by its Class OID property. Similarly for I-DATAPROPERTY and I-OBJECTPROPERTY.

Assume we have represented (by means of RDF triples) the bloodline relationships among the Aeneid’s characters Priam, Hector, Paris and Astyanax as shown in Figure 2. We use four individuals, \( i_0, \ldots, i_3 \), having the name of the above characters as property. The individuals are linked by two instances of the property father and one instance of the property brother representing that Hector and Paris are brothers and are both sons of Priam. Moreover, we represent the fact that Astyanax as the first-born of Hector.

The tables of the core meta-model storing the entities in Figure 2 are reported in Fig. 3. Note that data and meta-data
are managed in an uniform way.

![Diagram](image)

Fig. 2. Aeneid’s Bloodlines (RDF)

are managed in an uniform way.

![Diagram](image)

Fig. 3. A relational implementation of the meta-model of NYAYA

RDF schema can be included in this framework by simply adding to \( \mathcal{M} \) the following further constructs, which capture the notions of subclass and subproperty (in its two forms) respectively:

- \( \text{SUBCLASS}(\text{OID}, \text{classOID}, \text{subClassOID}) \)
- \( \text{SUBDATAPROPERTY}(\text{OID}, \text{propertyOID}, \text{subpropertyOID}) \)
- \( \text{SUBOBJECTPROPERTY}(\text{OID}, \text{propertyOID}, \text{subpropertyOID}) \)

Indeed, one of the main advantages of this approach is that it is easily extensible. For example, containers can be represented by adding three further constructs: \( \text{CONTAINER} \) (representing sequences, alternatives or bags) \( \text{SIMPLEELEMENT} \) and \( \text{CLASSELEMENT} \) (representing literal and resource elements of a container, respectively).

In Fig. 4 we show an UML-like diagram representing a fragment of \( \mathcal{M} \) involving the main constructs of OWL-QL. The rectangles represent constructs and the arrows references between them. Notice how the core of \( \mathcal{M} \) (enclosed in the dashed box) can serve to represent the facts of both OWL-QL and RDFS(DL) ontologies; the full specification of the meta-model can be found at [http://mais.dia.uniroma3.it/Nyaya](http://mais.dia.uniroma3.it/Nyaya).

In practical applications, the above tables could be very large and therefore a suitable tuning involving indexing and partitioning needs to be performed to guarantee good performances. This subject will be addressed in Section VI.

IV. REASONING AND QUERYING

Before going into the details of how reasoning and querying are actually done in NYAYA we introduce some fundamental notions.

A. Ontological Query Answering

An ontological schema \( S \) (or simply schema) is a set of unary and binary first-order predicates. A term \( t \) in a predicate may be an individual name (or simply an individual), a constant (e.g., strings and integers), a null (i.e., an anonymous individual or an unknown constant) or a variable. An atomic formula (i.e., an atom) is either a formula \( p(t) \) or \( p(t, t’) \), where \( p \) is a predicate of \( S \) and \( t, t’ \) are terms. We call position \( t \) the \( i \)-th argument of a predicate \( p \in S \). A unary predicate is called a concept name (or simply a concept), while binary predicates are referred to as role names (or simply roles) when both terms are individuals, and attribute names (or simply attributes) when the first term is an individual and the second is a constant.

Similarly to databases, the semantics of an ontological schema is given in terms of its models. Consider three pairwise disjoint sets \( \Delta_i, \Delta_c \) and \( \Delta_a \) of symbols such that: \( \Delta_i \) is a finite set of individuals, \( \Delta_c \) is a finite set of constants and \( \Delta_a \) is a (possibly infinite) set of, not necessarily distinct, labelled nulls. An ontological instance (or simply instance) \( \mathcal{I} \) for a schema \( S \) is a (possibly infinite) set of atoms of the form \( p(t), r(t, t’) \) and \( u(t, t’’, t’’) \) (named facts), where \( p, r \) and \( u \) are respectively a concept, a role and an attribute in \( S \) while \( t, t’ \in (\Delta_i \cup \Delta_c) \) and \( t’’ \in (\Delta_a \cup \Delta_c) \). The ontological instance can be seen as the extensional counterpart of the concept of interpretation of description logics (and logics in general) [20]. We say that an ontological instance \( \mathcal{I} \) is a model for an ontological schema \( S \) if and only if \( \mathcal{I} \models S \).

The structure of an ontology is defined by means of an ontology-modelling language that is used to specify the relationships among the entities in the ontology. The expressiveness of such languages ranges from simple assertions over the set of individuals and constants as in the RDF
language\textsuperscript{15} to complex languages such as OWL2-DL\textsuperscript{16} and FLORA-2\textsuperscript{17} whose semantics is given in terms of Description Logics [20] and F-Logic [21] respectively. The ontological structures defined by such languages are equivalent to enforce a set of first-order constraints restricting the number of models of the ontological schema.

The query language we consider is that of unions of conjunctive queries (UCQs) whose expressive power is the same of select-project-join Relational Algebra queries.

A conjunctive query (CQ) \( q \) over an ontological schema \( S \) is a formula of the form \( q(X) \Leftarrow \exists Y \phi(X, Y) \), where \( \phi(X, Y) \) is a conjunction of atoms over \( S \). \( X \) and \( Y \) are sequences of variable or constants in \( \Delta_c \). The answer to a CQ \( q \) over an ontological instance \( I \) denoted as \( \text{ans}(q, I) \) is the set of all the tuples \( t \in \Delta_c \) for which there exists a homomorphism \( h: X \cup Y \rightarrow \Delta_c \) such that \( h(\phi(X, Y)) \subseteq I \) and \( h(t) = t \).

If we add a set of constraints \( \Sigma \) the problem becomes that of ontological conjunctive query answering under constraints i.e., the problem of deciding whether \( I \cup \Sigma \models q \), where \( q \) is a conjunctive query.

The semantics we adopt for query answering is the certain answers semantics \textsuperscript{22}, requiring that an answer to a query be an answer in all the possible models of a given schema. Formally, let \( \text{mods}(\Sigma, I) \) be the set of all the (possibly infinite) sets of atoms \( D \) such that \( D \models I \cup \Sigma \). The answer to a CQ \( q \) over an instance \( I \) w.r.t. a set of constraints \( \Sigma \) (denoted by \( \text{ans}_c(q, I) \)) is the set of all the tuples which belong to \( \text{ans}(q, D) \) for all \( D \in \text{mods}(\Sigma, I) \).

Given a set of constraints \( \Sigma \), we say that query answering under \( \Sigma \) is first-order reducible, henceforth denoted by FO-reducible if and only if, given a query \( q \), it is possible to construct a first-order query \( q_{\text{rew}} \) (called the perfect rewriting) such that \( \Sigma \cup I \models q \) if and only if \( I \models q_{\text{rew}} \) for every instance \( I \). This class of constraints corresponds to the class of all the non-recursiv Datalog programs \textsuperscript{23}, for which data-complexity of query answering (i.e., fixed set of rules) is in uniform AC\textsuperscript{0} \textsuperscript{24}.

**B. Datalog\textsuperscript{\pm}**

Datalog\textsuperscript{\pm} is a family of extensions to the Datalog language \textsuperscript{23} where the rules are tuple generating dependencies, equality generating dependencies and negative constraints.

A tuple-generating dependency (TGD) is a formula \( \sigma \) over a schema \( S \) of the form \( \phi(X, Y) \rightarrow \exists Z \psi(X, Z) \) where \( \phi \) and \( \psi \) are conjunctions of atoms respectively called the head and the body of the TGD. Whenever \( Z \) is empty we say that the TGD is full because it completely specifies the variables in the head. Given a TGD \( \sigma \) we denote as \( U_{\sigma} \) the set of all the positions in \( \text{head}(\sigma) \) where a universally quantified variable occurs.

An equality-generating dependency (EGD) is a formula \( \eta \) over \( S \) of the form \( \forall X \phi(X) \rightarrow X_i = X_j \) where \( \phi(X) \) is a conjunction of atoms and \( X_i, X_j \in X \).

Given an EGD \( \eta \) and an atom \( a \in \text{body}(\eta) \), we denote as \( J^\eta_a \) (joined-positions) the set of all the positions \( \pi_a \) in such that the variable \( V \) occurring in \( \pi_a \) occurs also in another atom \( b \in \text{body}(\eta) \) in a position \( \pi_b \neq \pi_a \). We also denote as \( P^\eta_a(V) \) the set of all the positions of a variable \( V \) in an atom \( a \in \text{body}(\eta) \).

A negative constraint (NC) is a first-order sentence of the form \( \forall X \phi(X) \rightarrow \bot \), where \( \bot \) denotes the truth constant false. In other words, a negative constraint specifies that certain formulas must be false in every model of a given theory.

NCs have been proven useful for modelling various forms of ontological structures \textsuperscript{3}, \textsuperscript{25} as well as conceptual schemas such as Entity-Relationship (ER) diagrams \textsuperscript{26}.

It is known that query answering under TGDs is undecidable \textsuperscript{27}. However, the interaction of any decidable set of TGDs and NCs is known to be safe if it is decidable for the set of TGDs under consideration \textsuperscript{28}. When a set of EGDs is considered together with a set of TGDs, the problem of query answering is undecidable in the general case since EGDs generalise the well-known class of functional dependencies in databases. For this reason the interaction between EGDs and TGDs must be controlled to retain decidability. This can be done by assuring that EGDs and TGDs are non-conflicting \textsuperscript{29}, \textsuperscript{25}.

A TGD \( \sigma \) is non-conflicting with an EGD \( \eta \) iff for each atom \( a \in \text{body}(\eta) \), if \( \text{pred}(a) \neq r \) then one of the following conditions is satisfied: (i) \( U_{\sigma} \) is not a superset of \( J^\eta_a \) and each existential variable in \( \sigma \) occurs just once, or (ii) there exists a variable \( V \) such that \( P^\eta_a(V) \geq 2 \), and there exist two positions \( \pi_1, \pi_2 \in P^\eta_a \) such that in \( \text{head}(\eta) \) at \( \pi_1, \pi_2 \) we have either two distinct existential variables, or one existential and one universal variable. It is known that query answering under non-conflicting EGDs and TGDs is decidable \textsuperscript{3}, \textsuperscript{25}.

**Example 1.** (non-conflicting dependencies)

Consider the following dependencies:

\[
\eta: \text{firstBorn}(X,Y) \land \text{firstBorn}(X,Z) \rightarrow Y=Z
\]

\[
\sigma_1: \text{father}(X,Y) \rightarrow \text{firstBorn}(Y,X)
\]

\[
\sigma_2: \text{firstBorn}(X,Y) \rightarrow \text{father}(Y,X)
\]

The EGD \( \eta \) states that a father may have only one first-born child, the TGD \( \sigma_1 \) says that all the children of a father are first-born children (i.e., they are all twins) while the TGD \( \sigma_2 \) states that all the first-born children are also children. Clearly, if we assume that \( \eta \) holds, then, intuitively, \( \sigma_1 \) conflicts with \( \eta \) because they are both true exclusively in case of only children; instead, \( \sigma_2 \) is non-conflicting. This intuition is captured by the notion of non-conflicting dependencies of Datalog\textsuperscript{\pm}, in fact the set of joined-positions for firstBorn is \( J^\text{firstBorn}_1 = \{ \text{firstBorn}[1] \} \) and the set of universal positions in \( \sigma_1 \) is \( U_{\sigma_1} = \{ \text{firstBorn}[1], \text{firstBorn}[2] \} \) (i.e., conflicting) while \( U_{\sigma_2} = \emptyset \) (i.e., non-conflicting).

NYAYA implements Linear Datalog\textsuperscript{\pm}, a member of the Datalog\textsuperscript{\pm} family where the TGDs have a single atom in the body and for which conjunctive query answering is FO-
Table I

<table>
<thead>
<tr>
<th>DL</th>
<th>OWL-2-QL</th>
<th>RDF(S)DL</th>
<th>Datalog*</th>
</tr>
</thead>
<tbody>
<tr>
<td>Person ⊑ name ⊑ father Person</td>
<td>✓</td>
<td>-</td>
<td>Person(X) → ∃Y ∃Z name(X,Y) ∧ father(Z,X) ∧ Person(Z), Y=Z (EGD)</td>
</tr>
<tr>
<td>Doms(brother) ⊑ Person</td>
<td>✓</td>
<td>✓</td>
<td>brother(X,Y) → Person(X), father(Y,X) → Person(Y)</td>
</tr>
<tr>
<td>Doms(brother) ⊑ Person</td>
<td>✓</td>
<td>✓</td>
<td>brother(X,Y) → Person(Y)</td>
</tr>
<tr>
<td>Doms(father) ⊑ Person</td>
<td>✓</td>
<td>✓</td>
<td>father(X,Y) → Person(Y)</td>
</tr>
<tr>
<td>Doms(firstBorn) ⊑ Person</td>
<td>—</td>
<td>✓</td>
<td>firstBorn(X,Y) → Person(X)</td>
</tr>
<tr>
<td>firstBorn ⊑ father</td>
<td>✓</td>
<td>—</td>
<td>firstBorn(X,Y) → father(X,Y)</td>
</tr>
<tr>
<td>brother ⊑ brother*</td>
<td>✓</td>
<td>✓</td>
<td>brother(X,Y) → brother(Y,X)</td>
</tr>
<tr>
<td>func(firstBorn)</td>
<td>—</td>
<td>✓</td>
<td>firstBorn(X,Y) ∧ first Born(X,Z) → Y=Z (EGD)</td>
</tr>
</tbody>
</table>

| name(i1, 'Hector') | ✓ | ✓ | ✓ |
| name(i2, 'Paris')  | ✓ | ✓ | ✓ |
| name(i3, 'Astyanax') | ✓ | ✓ | ✓ |
| Person(i1)         | ✓ | ✓ | ✓ |
| father(i0,i1)      | ✓ | ✓ | ✓ |
| father(i0,i2)      | ✓ | ✓ | ✓ |
| brother(i1,i2)     | ✓ | ✓ | ✓ |
| firstBorn(i1,i3)   | ✓ | ✓ | ✓ |

Table II shows the typical structure of the rules in a semantic data-kiosk. The rules of $\Sigma_D$ are LTGDs encoding the definition of the concept $\text{Person}$, the role $\text{father}$ and the attribute $\text{name}$ while the rules of $\Sigma_S$ always constitute a safe and non-recursive Datalog program since all the atoms in the bodies never occur in any of the heads and, for each rule, all the variables appearing in the head also appear in the body.

Knowing that in NYAYA query-answering over $D$ under $\Sigma_D$ is FO-reducible, a natural question is now whether query answering remains FO-reducible also in the presence of $\Sigma_S$.

First of all, observe that every derivation using the rules in $\Sigma_S$ has length 1, because no atom ever contributes to the production of another body atom. Note now that this particular structure of the rules in $\Sigma_S$ imposes a natural stratification of the rules in $\Sigma_T$ w.r.t. those in $\Sigma_S$. Query answering can be then reduced to a two-step process where every conjunctive query $q$ is first rewritten in terms of the $\Sigma_T$ (by virtue of the FO-reducibility of Linear Datalog) obtaining a UCQ $Q_{\Sigma_T}$ that is a perfect rewriting of $q$ w.r.t. $\Sigma_T$. Every $q_{\Sigma_T}^{i} \in Q_{\Sigma_T}$ can be then rewritten in terms of $\Sigma_S$ obtaining another UCQ $Q_{\Sigma_S}^{i}$, that is a perfect rewriting of $q_{\Sigma_T}^{i}$ w.r.t. $\Sigma_S$. The union of all the $Q_{\Sigma_S}^{i}$ for every $i \in |Q_{\Sigma_T}|$ produced by the process is thus a perfect rewriting of $q$ w.r.t. $\Sigma_T \cup \Sigma_S$.

Based on the above considerations, the following theorem establishes that, provided that query answering under $\Sigma_T$ is FO-reducible, the logical organization of the data in the persistent storage does not affect the theoretical complexity of query answering.

**Theorem 1:**
Consider a set $\Sigma_T$ consisting of a Linear Datalog rules and a set $\Sigma_S$ consisting of a set of positive safe non-recursive Datalog rules $\Sigma_S$ such that (i) for each $\sigma \in \Sigma_S$ head($\sigma$) appears only in the rules of $\Sigma_T$ and (ii) every predicate appearing in $\Sigma_T$ appears only in $\Sigma_T$ or in the head of a rule $\sigma \in \Sigma_S$. Then, conjunctive query answering under $\Sigma_T \cup \Sigma_S$ is FO-reducible.
The most important aspect is that FO-reducibility of Linear Datalog\textsuperscript{\textregistered} programs can be syntactically checked for each program and does not depend on the language as it is the case for other Semantic-Web languages such as DL-Lite and RDFS(DL). Consider the example of Table I: due to the restrictions on the language imposed by RDFS(DL) and DL-Lite, none of them is able to represent the example completely. Instead Datalog\textsuperscript{\textregistered} can entirely capture the example, because the specific program is FO-reducible.

V. REWRITING-BASED ACCESS TO SEMANTIC DATA KIOSKS

The meta-model presented in Section III imposes a strong separation between the data and the meta-data in a kiosk. This section shows how this logical organisation supports efficient reasoning and querying.

Given a query \( q \), the actual computation of the rewriting is done by using the rules in \( \Sigma_T \) and \( \Sigma_S \) as rewriting rules in the style of [12] and applying backward-chaining resolution.

Given two atoms \( a,b \) we say that they unify if there exists a substitution \( \gamma \), called unifier for \( a \) and \( b \), such that \( \gamma(a) = \gamma(b) \). A most general unifier (MGU) is a unifier for \( a \) and \( b \), denoted as \( \gamma_{a,b} \), such that for each other unifier \( \gamma \) for \( a \) and \( b \), there exists a substitution \( \gamma' \) such that \( \gamma = \gamma' \circ \gamma_{a,b} \). Notice that if two atoms unify then there exists a MGU and it is unique up to variable renaming.

In order to describe how the perfect-rewriting is constructed, we now define the notion of applicability of a TGD to an atom of a query, assuming w.l.o.g. that the TGDs have a single atom in their head.

Definition 2: (Applicability) [26]

Let \( \sigma \) be a TGD over a schema \( S \) and \( q \) a CQ over \( R \). Given and an atom \( a \in body(q) \), we say that \( \sigma \) is applicable to \( a \) whenever \( a \) and \( head(\sigma) \) unify through a MGU \( \gamma_{a,\sigma} \) and the following conditions are satisfied:

1) if the term at position \( \pi \) is either a constant or a shared variable in \( q \), then the variable at position \( \pi \) in \( head(\sigma) \) occurs also in \( body(\sigma) \).
2) if a shared variable in \( q \) occurs only in \( a \) at positions \( \pi_1, \ldots, \pi_m \), for \( m \geq 2 \), then either the variable at position \( \pi_i \) in \( head(\sigma) \), for each \( i \in \{1, \ldots, m\} \), occurs also in \( body(\sigma) \), or at positions \( \pi_1, \ldots, \pi_m \) in \( head(\sigma) \) we have the same existential variable.

Whenever a TGD \( \sigma \) is applicable to an atom \( a \) of a query \( q \) we substitute \( \gamma_{a,\sigma}(body(\sigma)) \) to the atom \( a \) in \( q \) producing a so-called partial-rewriting \( q_\sigma \) of \( q \) such that \( q_\sigma \subseteq q \). Roughly speaking, we use the TGD as a rewriting rule whose direction is from the right to the left and the notion of applicability guarantees that every time an atom of a query \( q \) is rewritten using the TGD what we obtain is always a sub-query of \( q \).

Conditions (1) and (2) above are used to ensure soundness since a naive backward-resolution rewriting may produce unsound partial-rewritings as shown by the following example.

Example 2: (Loss of soundness)

Consider the LTGD \( \sigma : person(X) \rightarrow \exists Y name(X,Y) \) over the schema \( S = \{name, person\} \) and the conjunctive query \( q(A) \leftarrow name(X, Hector) \) asking for all the persons in the database named Hector. Without the restrictions imposed by the applicability condition a naive rewriting would produce the partial-rewriting \( q_\sigma \leftarrow person(A) \) where the information about the constant Hector gets lost. Consider now the database \( D = \{person(i_0), person(i_1), name(i_1, Hector)\} \). The answer to \( q_\sigma \) is the set \( \{person(i_0), person(i_1)\} \) where the tuple \( person(i_0) \), corresponding to Priam, is not a sound answer to \( q \).

However, the applicability condition might prevent some of the partial-rewritings from being generated thus loosing completeness as shown by the following example:

Example 3: (Loss of completeness)

Consider the following set of LTGDs \( \Sigma \) over the schema \( S = \{father, person\} \):

\[
\sigma_1: \quad person(X) \rightarrow \exists Y father(Y, Z) \\
\sigma_2: \quad father(X, Y) \rightarrow person(X)
\]

and the conjunctive query \( q(B) \leftarrow father(A, B), person(A) \).

The only viable strategy in this case is to apply \( \sigma_2 \) to the atom \( person(A) \) in \( q \), since the atom \( father(A, B) \) is blocked by the shared variable \( A \). The obtained partial-rewriting is thus
the query \( q_1^3(B) \leftarrow \text{father}(A, B), \text{father}(A, X_1) \). Note that, in \( q_1^3 \), the variable \( A \) remains shared thus it is not possible to apply \( \sigma_1 \). However, the query \( q_2^2(B) \leftarrow \text{person}(B) \) is also a partial rewriting but its generation is prevented by the applicability condition.

In order to solve this problem and achieve completeness, approaches such as \([12]\) resort on exhaustive factorisations that, however, produce a non-negligible number of redundant queries. We now define a restricted form of factorisation that generates only those queries that actually lead to partial-rewritings. This corresponds to the identification of all the atoms in the query whose shared existential variables come from the same atom in all the derivations and they can be thus unified with no loss of information.

**Definition 3: (Factorisability)**

Given a query \( q \) and a set of TGDs \( \Sigma \), we say that a position \( \pi \) in an atom \( a \) of \( q \) is *existential* if there exists a TGD \( \sigma \) such that a unifies with head(\( \sigma \)) and the term at position \( \pi \) in head(\( \sigma \)) is an existential variable. A set of atoms \( S \subseteq \text{body}(q) \) is *factorisable* w.r.t a TGD \( \sigma \) iff the following conditions hold:

- \( a \) and \( b \) unify through a MGU \( \gamma_{a,b} \) for all \( a, b \in S \)
- if a variable \( X \) appears in an existential position in an atom of \( S \) then \( X \) appears only in existential positions in all the other atoms of \( q \).

**Example 4: (Factorisability)**

Consider, as an example, the following CQs:

\[
q_1(A) \leftarrow \text{father}(A, B), \text{father}(C, B)
\]

\[
q_2(A) \leftarrow \text{father}(A, B), \text{father}(B, C)
\]

and the TGD \( \sigma : \text{person}(X) \rightarrow \exists Y \text{father}(Y, X) \). In \( q_1 \), the atoms \( \text{father}(A, B) \) and \( \text{father}(C, B) \) unify through the MGU \( \gamma = \{ C \rightarrow A \} \) and they are also factorisable since the variables \( C \) and \( A \) appear only in existential positions (i.e., \( \text{father}[1] \)) in \( q_1 \). The factorisation results in the query \( q_1(A) \leftarrow \text{father}(A, B); \) it is worth noting that, before the factorisation, \( \sigma \) was not applicable to \( q_1 \) while it is applicable to \( q_3 \). On the contrary, in \( q_2 \), despite the atoms \( \text{father}(A, B) \) and \( \text{father}(B, C) \) unify, the variable \( C \) appears in position \( \text{father}[2] \) which is not existential w.r.t \( \sigma \), thus the atoms are not factorisable.

We are now ready to describe the algorithm \( \text{LTGD-rewrite} \) whose pseudo-code is presented as Algorithm 1.

The perfect-rewriting of a CQ \( q \) is computed by exhaustive expansion of the atoms in \( \text{body}(q) \) by means of the applicable TGDs in \( \Sigma \). Each application of a TGD leads to a new query \( q_n \) (i.e., a partial-rewriting) retrieving a subset of the sound answers to \( q \); these queries are then factorised and stored into the set \( Q_n \). The process is repeated until no new partial-rewritings can be generated (up to variable renaming). The CQ \( q(X) \leftarrow \bigvee_{n \in \mathbb{N}, \text{body}(q_r(X, Y))} \text{perfect-rewriting} \) constitutes a perfect-rewriting i.e, a rewritten query that produces sound and complete answers to \( q \) under the constraints defined by \( \Sigma \).

**Algorithm 1 The algorithm LTGD-rewrite**

**Require:** A schema \( S \), a set of LTGDs \( \Sigma \) over \( S \) and a CQ \( q \) over \( S \)

**Ensure:** A set of partial rewritings \( Q \) for \( q \) over \( S \)

\[
\begin{align*}
Q_n \leftarrow \emptyset \\
\text{repeat} \\
\text{for all } q \in Q_n \text{ do} \\
\text{for all } \sigma \in \Sigma \text{ do} \\
\text{factorisation} \\
q' \leftarrow \text{factorise}(q) \\
\text{rewriting} \\
\text{for all } a \in \text{body}(q') \text{ do} \\
\text{if } \text{isApplicable}(\sigma, a, \gamma_{a,\sigma}) \text{ then} \\
q_n \leftarrow q_n \cup \{ q_n \} \\
Q_n \leftarrow Q_n \cup \{ q_n \} \\
\text{end if} \\
\text{end for} \\
\text{end for} \\
\text{end for} \\
\text{until } Q_n = \emptyset \\
\text{return } Q_r
\end{align*}
\]

In the following, an example of application of LTGD-rewrite is given:

**Example 5: (\( \Sigma_\Sigma \) Rewriting)**

Consider the set of LTGDs \( \Sigma \) of Example 3 and the query \( q(B) \leftarrow \text{father}(A, B), \text{person}(A) \). LTGD-rewrite first applies \( \sigma_1 \) to the atom \( \text{person}(A) \) since \( \sigma_1 \) is not applicable. The produced partial rewriting is the query \( q_1^3(B) \leftarrow \text{father}(A, B), \text{father}(A, V^1) \) whose atoms are factorisable through the MGU \( \gamma = V^1 \rightarrow B \). The factorised query is \( q_2^2(B) \leftarrow \text{father}(A, B) \). LTGD-Rewrite can now apply \( \sigma_1 \) to \( q_2 \) since the variable \( A \) in the existential position is now a non-shared variable. The obtained partial-rewriting is the query \( q_3^2(B) \leftarrow \text{person}(B). \) Let \( D \) be the set of partial rewriting produced by our algorithm is thus \( Q_r = \{ q, q_1^3, q_2^2 \} \).

Each partial-rewriting produced by LTGD-rewriter is then rewritten in terms of the storage tables using the rules in \( \Sigma_S \). In this case the rewriting is generated using the same technique adopted in Algorithm 1, however given a query \( q \) generated by LTGD-rewriter we know that: (i) for each atom in the query there exists only one rule in \( \Sigma_S \) capable of expanding it and (ii) if the head of the rule unifies with an atom of the query then the rule is always applicable because there are no existentials in the heads of the rules in \( \Sigma_S \). The algorithm then starts from \( q \) and rewrites its atoms until all of them have been expanded. In addition, in NYAQA we also exploit the knowledge about the logical organisation of the storage e.g., we know that (i) the foreign-key constraints in the storage tables are always satisfied by construction and (ii) the OIDs of the tuples in \( D \) are generated with a known
hash algorithm, this allows us to replace the foreign keys with the corresponding target values (hashes), thus simplifying the queries.

**Example 6: (ΩS Rewriting)**

Consider the set \( \Sigma_S \) of Table II and the query \( q \) of Example 5. The corresponding expansion is the following query:

\[
Q_S(A) \leftarrow\text{OBJECTPROPERTY}(Z_0,Z_1,Z_2,Z_3),\text{OBJECTPROPERTY}(Z_4,'father',Z_4,Z_5),\text{OBJECTPROPERTY}(Z_2,A,Z_6),\text{OBJECTPROPERTY}(Z_2,A,Z_7),\text{CLASS}(Z_6,'Person').
\]

The knowledge derived from the storage meta-model (see Figure 3) enables further optimisations of the above query such as the replacement of foreign-keys with the corresponding value that produce the following optimized query:

\[
Q_S(A) \leftarrow\text{OBJECTPROPERTY}(Z_0,Z_1,Z_2,'04'),\text{CLASS}(Z_1,A,'01'),\text{CLASS}(Z_2,A,Z_6).
\]

Since the produced queries are always conjunctive queries they can be easily rewritten in SQL and executed by a relational DBMS. The following is the SQL representation of the optimised query \( Q_S \) above:

```
SELECT C1.URI
FROM I-CLASS AS C1
WHERE C1.OID = OP1.SubjectI-ClassOID AND
C2.OID = OP1.ObjectI-ClassOID AND
C1.ClassOID = '01' AND
OP1.PropertyOID = '04'
```

VI. IMPLEMENTATION AND EXPERIMENTAL SETTING

**NYAYA** is implemented in Java and its inference engine is based on the IRIS Datalog engine\(^{18}\). The persistent storage has been implemented in Oracle 11g R2, exploiting its native partitioning techniques. In particular **NYAYA** adopts referential partitioning, that allows the partitioning of related tables based on referential constraints. In other words, the relations referring to the data are partitioned with respect to their foreign keys to the relations referring to the schema level (i.e. meta-data). For instance the `I-CLASS, I-DATAPROPERTY` and `I-OBJECTPROPERTY` were partitioned with respect to the `classOID` and `propertyOID` references respectively. Each partition has a check constraint to redirect data inserted into the parent table and query processing into the children. Moreover, we defined clustered B+ tree indexes on the `SubjectClassOID` attribute of `I-OBJECTPROPERTY` and `I-ClassOID` attribute of `I-DATAPROPERTY`, and unclustered B+ tree indexes on the `ObjectClassOID` of `I-OBJECTPROPERTY`, `Value` of `I-DATAPROPERTY` and `URI` of `I-CLASS`. In our implementation we generated OIDS and translated textual data (e.g value of `I-DATAPROPERTY`) by hash coding to gain more performance.

A prototype implementation of **NYAYA** is available online at http://pamir.dia.uniroma3.it:8080/nyaya.

In the remainder of this section we discuss the results of our experimental campaign conducted to evaluate the performance of our system.

**A. Performance Evaluation**

We compared **NYAYA** with two well-known systems: BigOWL\(^{19}\) and IODT\(^{20}\). We used BigOWL v. 3.3 on File System (i.e. as Backend store), while IODT v. 1.5 including the Scalable Ontology Repository (SOR), was used over DB2 v.9.1. All the experiments were performed on a dual quad core 2.66GHz Intel Xeon, running Linux RedHat, with 8 GB of memory, 6 MB cache, and a 2-disk 1Tbyte striped RAID array.

In our experiments we used two widely-accepted benchmarks: LUBM\(^{21}\) and UOBM [30] (i.e. we generated up to 50 Universities, that is 12.8 million triples). Since the expressiveness of the meta-data in the above datasets exceeds the one of Datalog\(^{\pm}\) (i.e., OWL-Lite for LUBM and OWL-DL for UOBM), in our experiments we used the best FO-reducible approximation of the meta-data for all the systems.

Moreover we used Wikipedia\(^{3}\), a conversion of the English Wikipedia\(^{2}\) into RDF. This is a monthly updated dataset containing around 47 million triples. The dataset contains the two main classes `Article` and `Category` (where Category is subclass of `Article`), four datatypes (i.e. `text`, `dc:Title`, `dc:modified`, `dc:contributor`) and seven objectproperties (i.e. `skos:subject`, `skos:narrower`, `link` with `subProperties internalLink`, `externalLink`, `interwikiLink`, and `redirectsTo`). The expressiveness of its meta-data completely falls into the Datalog\(^{\pm}\) so no adaptations were made.

We conducted three groups of experiments: (i) data loading, (ii) querying and reasoning, and (iii) maintenance.

<table>
<thead>
<tr>
<th>Category</th>
<th>LUBM(50)</th>
<th>UOBM(50)</th>
<th>Wikipedia(3)</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>BigOWL</strong></td>
<td>8 (mins)</td>
<td>77 (mins)</td>
<td>9,18 (mins)</td>
</tr>
<tr>
<td><strong>IODT</strong></td>
<td>18 (hours)</td>
<td>65 (hours)</td>
<td>0.5 (hours)</td>
</tr>
<tr>
<td><strong>NYAYA</strong></td>
<td>71 (hours)</td>
<td>181 (hours)</td>
<td>2.38 (hours)</td>
</tr>
</tbody>
</table>

**Data loading.** As shown in Table III, **NYAYA** outperforms significantly IODT and BigOWL\(^{19}\). The main advantage of our framework is that we execute pure data loading while other systems have to pre-compile the knowledge-base producing a (possibly exponential) blow-up of the triples to be stored. The loading time for BigOWL\(^{19}\) changes dramatically from LUBM to UOBM (i.e. 135 times slower) due to the increasing complexity of the meta-data, and from LUBM to Wikipedia\(^{3}\) (i.e. 532 times slower), due to the large set of facts. Moreover our framework is supported by the meta-model

\(^{18}\)http://www.iris-reasoner.org/

\(^{19}\)http://www.ontotext.com/owl/big/

\(^{20}\)http://www.alphaworks.ibm.com/tech/semanticstk

\(^{21}\)http://swat.cse.lehigh.edu/projects/lubm/

\(^{2}\)http://labs.systemone.net/wiki/wikipedia3
presented in Section III that allows a parallel execution of data import into the DBMS. BigOWLlim has to process all triples in memory, while IODT exploiting the Multi Dimensional Clustering (MDC) [31] organization has to maintain complex block indexes pointing to groups of records.

**Querying and reasoning.** In this group of experiments, we employed the 14 and 13 benchmark queries from LUBM and UOBM respectively. Due to space limitations we omit the query lists here. For Wikipedia3 we defined six representative queries, as follows:

- Q1(X) :- Article(X), contributor(X,'Adam Bishop').
- Q2(X) :- Article(X), link('wiki:List_of_cities_in_Brazil',X).
- Q3(X) :- Article(X), link(X,Y), internalLink('wiki:United_States', Y).
- Q4(X) :- link(X,Y), link(Y,Z), link(Z,'wiki:E-Mail').
- Q5(X) :- Article(X).
- Q6(Y) :- Article(X), internalLink('wiki:Algorithm',X) internalLink(X,Y).

For OWL querying and reasoning, we performed cold-cache experiments (i.e. by dropping all file-system caches before restarting the various systems and running the queries) and warm-cache experiments (i.e. without dropping the caches). We repeated all procedures three times and measured the average execution time.

**UOBM50.** We ran queries with increasing complexity. In particular we evaluated three components for each time: preprocessing, execution and traversal. BigOWLlim exploits the preprocessing to load statistics and indexes in memory, while NYAYA computes rewriting tasks (i.e. $\Sigma_G$, $\Sigma_S$ and SQL rewritings). IODT doesn’t adopt any preprocessing. The query run-times are shown in Figure 5.

In general BigOWLlim performs better than IODT, and NYAYA is the best among the three approaches despite some
queries where the generated $\Sigma_O$ rewriting produces a large number of CQs to be executed. However, we are currently investigating optimisation techniques to significantly reduce the number of CQs produced by query-rewriting. We obtained average cold-cache times of 1257 with respect to 1783 of BigOWLlim and 1993 of IODT. The warm-cache times of NYAYA (i.e. geometric mean of 39) are comparable with BigOWLlim (i.e. geometric mean of 42) and better than IODT (i.e. geometric mean of 168). NYAYA spends most of the computation of the $\Sigma_O$ rewriting and then benefits from generating UCQs where each CQ can be executed in parallel and is efficiently answerable. Moreover, as for data loading, the entire procedure is well-supported by the persistent organization of data. BigOWLlim and IODT mainly focus on the execution step. Therefore the amount of data to query (i.e. reduced in our system) plays a relevant role. The warm-cache times demonstrates how BigOWLlim is well optimized to reduce memory consumption for triple storage and query processing. We omit the LUBM results because the systems present a similar behaviour.

Wikipedia3. Figure 6 shows the query run-times. NYAYA is the best (in average) among the three approaches, but in this case IODT performs better than BigOWLlim. This is due to the significant amount of data to process, although the $\Sigma_O$ is simpler. All queries present the main hierarchy inferencing on Article and link. In particular Q3 requires to traverse all articles linking to other articles linked by http://en.wikipedia.org/wiki/United_States while Q4 requires articles subject of a complex chain of links. Both the queries stress as the reasoning mechanism as the storage organization of all systems. In this case NYAYA outperforms all competitors by a large margin. We improve cold cache times by a geometric mean of 8618 with respect to 21506 of BigOWLlim and 16519 of IODT. The warm-cache times of NYAYA (i.e. geometric mean of 760) are comparable with IODT (i.e. geometric mean of 785) and better than BigOWLlim (i.e. geometric mean of 1338). In this situation IODT demonstrates high performance and scalability in query and reasoning with respect to BigOWLlim.

Maintenance. In these last experiments we tested maintenance operations in terms of insertion (deletion) of new (old) terms or assertions and update of existing terms or assertions into our datasets.

Figure 7 shows the performance of each system with respect to maintenance operations (i.e., insert, delete and update) of data and meta-data w.r.t. the LUBM, UOBM and Wikipedia datasets. We consider the average response time (msec) to insert, delete and update a predicate (i.e., concept, role or attribute) among the meta-data or a fact in the database. The results highlight how NYAYA outperforms the other systems when dealing with inserts and deletions of both data and meta-data. In other words, in both BigOWLlim and IODT, to insert a new predicate or a fact entails re-compiling a (possibly huge) set of tuples matching or referring to the new predicate (e.g. subclasses, property restrictions, and so on) while in NYAYA this is done by inserting or removing a
single-tuple in the storage. On the other side, in order to update an existing predicate in all the systems the meta-data (e.g., foreign keys, integrity constraints and so on) of all the facts referring to that predicate must be updated. As a consequence, the complexity of this task depends also on the amount of data stored in the database. In this respect, Figure 7 shows the behaviour of NYAYA w.r.t. meta-data updates when pure-SQL operations are used (i.e. DBMS independent) and when partition-maintenance functions provided by the DBMS (i.e. Oracle) are adopted.

VII. CONCLUSION AND FUTURE WORK

In this paper we have presented the development of a system, called NYAYA, for the uniform management of different repositories of Semantic Web data. NYAYA provides advanced reasoning capabilities over collections of Semantic-Web data sources taking advantage, when available, of meta-data expressed in diverse ontology languages and at varying levels of complexity. In addition, it provides the ability to (i) query the sources with a uniform interface, (ii) update efficiently both data and meta-data, and (iii) easily add new sources. The reasoning and querying engine of NYAYA relies on Datalog\(^{\text{\dagger}}\), a very general logic-based language that captures the most common tractable ontology languages. The effectiveness and the efficiency of the system have been tested over available data repositories and benchmarks.

This work opens a number of interesting and challenging directions of further research. First, we are planning to extend the approach to other FO-rewritable members of the Datalog\(^{\text{\dagger}}\) family, such as Sticky TDGs [25]; the next natural step is to further extend NYAYA to deal with non-FO-reducible languages like Guarded and Weakly-Guarded Datalog\(^{\text{\dagger}}\). Moreover, on a different front, it would be interesting to exploit NYAYA for supporting also the reconciliation and the integration of the data of the Semantic Data Market.

REFERENCES