A Model Based Approach to Semantic Web Data Management

De Virgilio, Pierluigi Del Nostro, Giorgio Gianforme, Stefano Paolozzi, Riccardo Torlone

RT-DIA-136-2008

September 2008

Dipartimento di Informatica e Automazione,
Università di Roma Tre,
Rome, Italy.
dvr@dia.uniroma3.it, pdn@dia.uniroma3.it, giorgio.gianforme@gmail.com,
stefano.paolozzi@gmail.com, torlone@dia.uniroma3.it
ABSTRACT

The Semantic Web is gaining increasing interest to fulfill the need of sharing and reusing information. In this context, RDF has been conceived to provide an easy way to represent any kind of data and metadata, according to a graph-based, lightweight model and a straight XML serialization. Although RDF has the advantage of being general and simple, it cannot be used as a storage model as it is, since it can be easily shown that even simple management operations involve serious performance limitations.

Starting from an analysis of the current state of the art for RDF data management, we present in this paper a novel approach for storing, managing and processing RDF data in an effective and efficient way. The approach is based on an organization that is particularly suited for RDF constructs, but it can be easily extended to other models that relies on RDF, like RDFS and/or OWL. We refer to real world scenarios in which large RDF data repositories need to be navigated and processed, even if the schema is not known in advance. We consider complex data management operations, which go beyond simple selections or projections and involve the navigation of huge portions of data sources.

A tool has been developed to test the feasibility of the approach over large RDF repositories (datasets of millions of triples). We compare the performance of the proposed framework with prior solutions and present experimental results supporting its effectiveness and efficiency.
1 Introduction

From the origins of the World Wide Web the activity of publishing information has been so intense that a huge amount of data is currently available over the internet. The technologies that have become standards for the creation of Web content are oriented to human users. Moreover, the absence of a unique point of reference for the semantic of data does not allow the use of software agents that understand the meaning of data, performing advanced searches, inferencing, wrapping, etc. The interpretation of the semantic of data is competence of the reader, with the obvious consequence of misunderstandings. This initial approach reveals weakness in sharing and reusing of information, needs that continuously grow with the increase of available data over the Web.

In 2001 Tim Berners Lee [5] proposed the concept of Semantic Web, a world where information is processable by machines other than humans, opening new interesting perspectives. In order to let a rough data be unambiguously interpreted by a computer, it is necessary to describe it by means of metadata that associate a well defined meaning. For this purpose the W3C developed the Resource Description Framework (RDF) [22] that is commonly used as the data model for the Semantic Web. It is a family of specifications used to express statements on Web resources, uniquely identified by an URI (Uniform Resource Identifier). The statement is in the form of a triple \((\text{Subject}, \text{Predicate}, \text{Object})\) expressing that a resource (Subject) is related to another resource or to a value (Object) through a property. An RDF document can then be seen as a directed labeled graph where nodes are resources or literals and arcs represent predicates, also called properties in RDF terminology. Nodes without content (blank nodes) are used in order to express incomplete information, queries or resources with a complex structure (specified via other triples). Many serializations of the abstract model of RDF exist, to allow the exchange of metadata across applications, the most diffused are RDF/XML and Notation 3. To provide a datatyping model for RDF, the W3C defined RDF Schema that allow the knowledge designer to define classes with subclasses, properties with range and domain, containers and other elements to define a slight more advanced knowledge structure.

Due to its simplicity RDF is easily manageable by applications and increasingly used over the Web. However the rise of Semantic Web technologies requires a growing number of meta-information to be managed, transformed and queried. The maintenance and querying of RDF documents represent crucial activities to profit from available semantic information. The concept of maintenance is related to the insertion or deletion of a node in an RDF document. Insertion operations add new knowledge to the RDF document, through inference or explicit insertions. Deletion operations represent the elimination of information from the RDF document, adding some complex problems because of potentially dangerous operations due to existing dependencies between nodes. In this paper we will consider maintenance related to scalability issues by applying our approach to a growing RDF document. Querying an RDF document is also an active field of research, as shown by the large number of languages developed to this aim (see [9] for a recent survey). To perform queries, RDF data can be stored in different kinds of repository. As we will detail in the following, proposals in this sense mainly concentrate on a triple organization of data with some tuning operations to improve query execution. Managing RDF triples in one table, however, compromise scalability and performances because of the high number of self-joins needed. To avoid this problem, some approaches present a different data organization. For example 3Stores [10] separates literals from resource
values while in [1] different RDF storage schemas are considered (i.e. vertical partitioning, property tables, column oriented DBMS).

In this paper we propose a storage system that is specifically suited to work with huge RDF documents. We address the issues of storing and querying RDF exploiting a meta-model technique. Differently from the approaches in literature, that provide implementation solutions, we follow a process that starts with the definition of a meta-representation of the RDF data model at a conceptual level. In this phase we find out the structural aspects of interest of RDF. The next step is a logical translation of the conceptual model for a relational database. Eventually, at the physical level, we choose optimization techniques based on indexing and partitioning. Here we concentrate on pure RDF (with no schema information) but the representation technique that we use is extensible: the model can be enriched to represent structural aspects of more complex models like RDF(S) and OWL.

The approach we propose here can be used for an agile management of RDF documents. More precisely, the most relevant contributions of our work are:

(i) The creation of a high level description model to represent the information stored in RDF document, pointing out the implicit semantics of elements through constructs.

(ii) A logical organization of data, based on the constructs structure of the model, referring to a relational database.

(iii) Optimizations based on indexing and partitioning of relational implementation to allow high performance querying and maintaining.

We compare our experimental results with two other approaches: the standard RDF triple [6] and Vertical Partitioning [1]. The aspects that we have measured are the scalability of the approaches and query performance over a set of representative queries.

The paper is structured as follows. In Section 2 we present a motivating example. In Section 3 we discuss some related works. in Section 4 we illustrate the details of our design process to manage RDF data. In Section 5 we show experimental results of performance and scalability and in Section 6 we sketch concluding remarks and future works.

2 Running Example

Commonly, an RDF document is organized in a set of triples that can be stored in one single relational table using a three-column schema. Let us consider the Figure 1 showing RDF classes (left side) and corresponding triple instances (right side) of an RDF document. We consider both properties (relations between a resource and a literal) and predicates (relations between two resources). More in detail, each class Person has a property, Name, and two predicates, Child and Brother, representing family relationships between persons. There are four instances of the class Person (with URI1, URI2, URI3 and URI4 as URIs\(^1\), respectively), each one with a corresponding instance of the property Name (with values Priam, Hector, Astyanax and Paris, respectively) and linked by two instances of predicate Child and one instance of predicate Brother representing that Hector and Paris are sons of Priam and brothers, and Astyanax is son of Hector.

\(^1\)RDF exploits Universal Resource Identifiers (URIs), which appear as URLs that often use sequence of numbers as identifier. In this paper, our examples will use more intuitive names.
The RDF instance can be stored in a relational table `people`, as shown at the right side of Figure 1. We can perform various operations on the RDF data regarding both queries and update/maintenance. For example, we can formulate the SQL query to find all of the persons that have a child whose name is ‘Paris’ as follows:

```
SELECT p4.obj
FROM people AS p1, people AS p2, people AS p3, people AS p4
WHERE p1.prop="Child" AND p1.obj=p2.subj AND p2.prop="rdf:type"
    AND p2.obj='Person' AND p1.obj=p3.subj AND p3.prop="Name"
    AND p3.obj="Paris" AND p4.subj=p1.subj AND p4.prop='Name'
```

This query performs many self-joins over the same table. Since the table `people` could contain a relevant amount of statements, the entire process potentially presents a high execution complexity. Indeed, the execution time increases as the number of triples scales, because each filtering or join condition will require a scan or index lookup.

Let us consider a more complex query. We want to find the family relation between Astyanax and Hector as follows:

```
SELECT t1.prop
FROM people AS t1, people AS t2, people AS t3, people AS t4,
     people AS t5
WHERE t1.subj=t2.subj AND t2.prop='Name' AND t1.subj=t3.subj
    AND t3.obj='Person' AND t1.obj=t4.subj AND t4.prop='Name'
    AND t1.obj=t5.subj AND t5.obj='Person' AND
    ((t2.obj='Hector' AND t4.obj='Astyanax')
     OR (t4.obj='Hector' AND t2.obj='Astyanax'))
```

Also in this case the number of self-joins increases as the number of scan or index lookups.

Real world executions does not only involve a lot of joins or filters, which make critical the selectivity and optimization of query (i.e. limiting the benefit of using indexes), but
also maintaining operations as insertions, deletions and updates of triples. The maintenance of large RDF datasets represents a complex problem because of potentially dangerous operations due to existing dependencies between nodes: it is important to identify, for instance, which other nodes must be deleted as the effect of a single deletion.

3 Related Works

Managing RDF information represents an important and wide area of research and a number of methodologies and techniques, along with many tools have been developed [9]. In this section we discuss the current researches concerning the management of RDF documents.

We distinguish among three main direction in RDF management based studies, namely: i) storing, that can be a complex activity, due to the fact that some storage systems requiring fixed schemas may be unable to handle general data such as that from RDF, where the terms are not known in advance; ii) querying, that might be seen as more complex than “conventional” querying, because the meaning conveyed by RDF data has to be properly “understood” and processed; iii) maintaining, that is the set of update/deletion operations on RDF documents.

In this paper we mainly focus on storing, because through our representation it is possible to perform querying and maintaining of RDF documents via SQL statements.

RDF storing approaches can be divided into two main areas: the first one is based on native store system development while the second one focuses on the use of relational (or object-relational) databases to store RDF data. Comparing the two approaches, native store systems (such as AllegroGraph\(^2\) or OWLLIM \([13]\)) are more efficient in terms of load and update time. On the contrary, DBMS-based approach are more efficient in querying due to the availability of many query management features. However, native storing approaches have the drawback of having to redefine important database features such as: query optimization, transaction processing, and so on.

To efficiently query RDF data is important to organize these information in a convenient way. Some interesting studies were focused on the physical organization of RDF Data. Storing RDF in relational (and/or object-relational) database management systems has been the main topic of much research. Indeed it seems that a relational model is a natural fit for RDF. The RDF model is a set of statements about Web resources, identified by URIs. The statements have the form \(<subject,predicate,object>\), then an RDF model can be easily implemented as a relational table with three columns. Many RDF storage systems have used this kind of approach such as the ICSFORTH RDFSuite \([2]\), the Sesame schema-based repository and querying system \([6]\), the KAON\(^3\) Project, the TAP\(^4\) suite, Jena \([23]\), Oracle \([7]\), and many others. Most of the aforementioned systems use storing techniques that do not involve entire strings in the triples table; instead they store shortened versions. Oracle and Sesame map string URIs to integer identifiers so the data is normalized into two tables, one triples table using identifiers for each value, and one mapping table that maps the identifiers to their corresponding strings. 3Store \([10]\) uses a similar approach, the only difference is on the fact that the identifiers are created

---


\(^3\)KAON - the Karlsruhe Ontology and Semantic Web Tool Suite. [http://kaon.semanticweb.org/](http://kaon.semanticweb.org/)

by applying a hash function to each string.

Nevertheless, these approaches present various limitations. For instance, considering the NULL values (possibly many) management in case of blank nodes or the awkward expression of multi-valued attributes in a flattened representation. Therefore the typical triple store approach cannot leverage this higher-level knowledge. In order to overcome some of the aforementioned hindrances we propose an approach that can take into account also the blank nodes as described in the rest of the paper.

For scalability and high-performance, we believe that the triple-store approach must be augmented by other storage strategies that can be efficient and sufficiently general.

Another work that can be compared with our approach is the research of Abadi et al. [1]. Like them we propose a logical model to represent and manage the RDF information but their approach is based on the *vertical partitioning* of RDF. Practically the triples table is rewritten into $n$ two-column tables where $n$ is the number of unique properties in the data. In these tables, the first column contains the subjects that define that property and the second column contains the object values for those subjects. In Figure 2 the tables of vertical partitioning representation for the example of Section 2 are depicted. Therefore, their approach is based on the “semantic” of the properties. On the other side our approach is based on the particular meta-constructs used to represent the RDF elements.

Referring to the example of Section 2, we can formulate the SQL query to find all the name of the persons that have a child whose name is ‘Paris’ as follows:

```sql
SELECT N.obj
FROM Name AS N, Name AS N2, Child AS C
WHERE N.subj=C.subj AND C.obj=N2.subj
AND N2.obj="Paris"
```

As we can see the number self-joins are significantly reduced. However, let us analyze the second and more complex query (i.e. to find the family relation between *Astyanax* and *Hector*). Using the vertical partitioning approach, we must firstly extract the tables list (because each table represents a property), insert them in a view and perform the joins with this view. In this case the query becomes more complex (and we omit it for the sake of brevity) and the total number of joins will significantly increase.
To overcome some of the aforementioned problems, we develop an approach which main characteristics can be summarized in two features: strong flexibility and extendibility. The proposed approach represents concepts of RDF pointing out semantics of elements. We also take into account both blank nodes and RDF containers (i.e. Bag, Seq and Alt). In this way information stored in an RDF document are properly grouped making effective the querying process.

4 Management of RDF data

4.1 Overview

An RDF model is comparable to a directed labeled graph. However, it allows the presence of multiple edges between two nodes, and different edges between two nodes can share the same label. The nodes, representing resources, can be classified as URI references or blank nodes or literals (strings). The edges in the graph represent properties.

Formally, we can define the following sets: the set of resources $R$, the set of URI references $U$, the set of blank nodes $B$, the set of literals $L$, and the set of properties $P$. At RDF level these sets present the following properties

- $R = U \cup B$,
- $P \subset R$,
- $U$, $B$, and $L$ are pair-wise disjoint.

In $P$ the property $\text{rdf\text{\text{:type}}}$ defines the type of a particular resource instance: any resource can be the target of an $\text{rdf\text{\text{:type}}}$ property. Therefore an RDF model $M$ is a finite set of triples (i.e. statements) as:

$$M \subset R \times U \times (R \cup L)$$

Each triple or statement in an RDF model contains a resource, an URI reference (which stands for a property), and a resource or literal. The properties in an RDF model are the middle element of a triple, or they are a resource with an $\text{rdf\text{\text{:type}}}$ property to the $\text{rdf\text{:Property}}$ resource. So the set of properties of an RDF model $M$ is:

$$P = \{p|(s,p,o) \in M \lor (p,\text{rdf\text{\text{:type}},rdf\text{:Property}}) \in M\}$$

The graph model $G_{\text{RDF}}$ corresponding to an RDF model $M$ is:

- $G_{\text{RDF}} = (N_{\text{RDF}}, E_{\text{RDF}}, fl_{N_{\text{RDF}}}, fl_{E_{\text{RDF}}})$
- $fl_{N_{\text{RDF}}} = N_{\text{RDF}} \rightarrow R \cup L$
- $fl_{E_{\text{RDF}}} = E_{\text{RDF}} \rightarrow P$

using the following construction mechanism ($N_{\text{RDF}}$ and $E_{\text{RDF}}$ denote nodes and edges respectively, $fl_{N_{\text{RDF}}}$ and $fl_{E_{\text{RDF}}}$ their labels). For each $(s,p,o) \in M$, we add the nodes $n_s, n_o$ to $N_{\text{RDF}}$ (different only if $s \neq o$) and label them as $fl_{N_{\text{RDF}}}(n_s) = s$, $fl_{N_{\text{RDF}}}(n_o) = o$, $fl_{E_{\text{RDF}}}(e) = p$.\]
and add $e_p$ to $E_{\text{RDF}}$ as a directed edge between $n_s$ and $n_o$ and label that as $fl_{E_{\text{RDF}}}(e_p) = p$. In the case that $s$ and/or $o$ are in $B$, then $fl_{N_{\text{RDF}}}(n_s)$ and/or $fl_{N_{\text{RDF}}}(n_o)$ are not defined: blank nodes do not have labels. Nodes that have a label have a unique one, edges always have a label but can share it with other edges.

Although an RDF model has the advantage of being general and simple, it cannot be used as a storage model as it is since even simple management operations involve serious performance limitations.

The management of a relevant amount of RDF data requires, in our approach, a specific storing process, as shown in Figure 3.

This process can be characterized by three steps:

- The first step of the process is the parsing of RDF documents. We consider RDF data represented with different syntaxes (e.g. RDF/XML and N3). As a result of this step we obtain a triple representation of the documents (i.e. $\langle s, p, o \rangle$ statements).

- In the second step the system maps the triple representation into an internal organization. It presents
  
  - a *conceptual level*, proposing a simple conceptual model where a set of constructs properly represents RDF concepts. Each construct is used to properly represent elements of RDF documents, with the same semantics;
  
  - a *logical level*, implementing our conceptual model into a logical one. In our case we use the relational model;
  
  - a *physical level*, defining the physical design of the logical representation of previous level. There, we illustrate a special partitioning technique that relevantly increases the performance of the entire process.

Figure 3: RDF data Management storing process
In the last step the storing in a specific DBMS is performed. In our case we use a relational DBMS.

In the following we exploit our organization of RDF data, from the conceptual to the physical organization, through the logical one. We close the section highlighting the advantages of such data organizations.

4.2 Conceptual level

Our approach is inspired by works of Atzeni et al. [3, 20] that propose a framework for the management of heterogeneous data models in an uniform way. They leverage on the concept of metamodel that allows a high level description of models by means of a generic set of constructs. The various constructs have an identifier, are related each other by means of mandatory references and may have attributes that specify details of interest. Formally, a model can be represented as $M = \{C_1, C_2, \ldots, C_n\}$ where the $C_i$’s are constructs, that have the following structure:

$$C = (OID, attr_1, \ldots, attr_f, ref_1, \ldots, ref_m)$$

where $OID$ is the object identifier, the $attr_j$’s are the properties of the construct and the $ref_k$’s are the references of the construct.

Following this idea, we propose a simple model where a set of constructs properly represent concepts expressible with RDF. The main differences with respect to the approach of Atzeni et al. are:

- they consider a set of constructs able to represent several data models, while we consider a reduced set of constructs because we use those constructs to represent just the RDF model;
- they consider a well marked distinction between schema and instance level. Constructs and rules for the instance level are automatically derived from structures at schema level. For our purposes, we need to manage only instances, since RDF documents contain essentially data. RDFS is devoted to specify schemas for RDF documents, but the modeling of such standard is beyond the focus of this paper.

In order to represent the basic concept of a triple, two constructs should be enough, one to represent resources and one to represent statements involving two resources. The concept of resource is clearly defined and can’t be subsequently specified, while the concept of statement allows for a more precise modeling. In fact, we can distinguish between two kinds of statement on the basis of its object, that can be a literal with a primitive type value or a resource with its own URI. We named these constructs Property and Predicate, respectively. More in details the Resource construct has a URI attribute to store the URI of the resource (we use the internal OID representation as an URI for a blank node); the Property construct has a reference to the Resource it belongs to and has a value attribute to store the value of the object; the Predicate construct has two Resource references to represent the subject and the object of a statement.

Formally we define

$$M_{\text{RDF basic}} = \{C_{\text{Resource}}, C_{\text{Property}}, C_{\text{Predicate}}\}$$
where the constructs have the following structure:

\[
C_{Resource} = (OID, URI) \\
C_{Property} = (OID, Name, Value, ResourceOID) \\
C_{Predicate} = (OID, Name, SubjectResourceOID, ObjectResourceOID)
\]

Referring to an RDF Model $M$ and its corresponding graph representation $G_{RDF}$, we have the following correspondences with our conceptual model $M_{RDF}$:

- $\{R \cup U \cup B\} \mapsto C_{Resource}$
- $\{p \in P | (s, p, o) \text{ where } s, o \in R\} \mapsto C_{Predicate}$
- $\{(p \in P | (s, p, o) \text{ where } o \in L), L\} \mapsto C_{Property}$

where $A \mapsto C_i$ means that the instances of $C_i$ represent the elements of the set $A$. Moreover the labels $f_{l_{RDF}}$ and $f_{l_{RDF}}$ correspond respectively to the attributes $URI$ of $C_{Resource}$ and Name of $C_{Predicate}$ and $C_{Property}$.

This simple model can be extended in order to manage also RDF collections; this is done with the addiction of three new constructs. They are $Container$, $SimpleElement$ and $ResourceElement$. We use the first one to specify that a blank node (represented with a resource as well) is in practice a collection: this is done by means of the reference ResourceOID of the construct toward a resource. The construct $Container$ has also a property $Type$ to denote the type of collection (i.e. Seq, Alt, Bag). The others constructs represent elements of a collection that can be, respectively, literals and resources; a literal element has a $Value$ attribute to store the value of the object while a resource element has a reference toward a resource (to specify to which resource the elements of the collection belong to) and both have a reference to the container to which they belong to. Formally we define

\[
M_{RDF} = \{C_{Resource}, C_{Property}, C_{Predicate}, C_{Container}, C_{SimpleElement}, C_{ResourceElement}\}
\]
where the new constructs have the following structure:

\[
C_{\text{Container}} = (OID, Type, ResourceOID)
\]
\[
C_{\text{SimpleElement}} = (OID, Name, Value, ContainerOID)
\]
\[
C_{\text{ResourceElement}} = (OID, Name, ResourceOID, ContainerOID)
\]

In Figure 4 an UML-like diagram of our $M_{RDF}$ model is represented, where enclosed in the dashed box there is the $M_{RDF\text{basic}}$ model.

For instance, through our approach, we can represent the RDF document, presented in Section 2, as depicted in Figure 5, where we omit the OIDs for the sake of simplicity and represent references only by arrows. In the figure $\text{URI}_1$, $\text{URI}_2$, $\text{URI}_3$ and $\text{URI}_4$ represent the URIs of the instances of the resources $\text{Person}$.

### 4.3 Logical Level

We use a relational implementation of our conceptual model. For each construct we create a table and for each field of a construct we add a column to the table corresponding to such construct.

The OID attribute is the primary key for each table, and we add an integrity constraint for each reference, from the pointer construct to the pointed one (i.e. from the column corresponding to the reference field toward the OID column of the referenced construct).

In Figure 6 some tables of our logical organization are depicted (we omit the tables devoted
to represent collections for the sake of simplicity, because there are no collections in our running example). The value of the rows are those corresponding to our running example.

Notice that, since we exploit different levels of abstraction, the logical level could be implemented also by an object oriented model.

### 4.4 Physical level

The resulting tables of logical level could be very large. To this aim we use a partitioning technique referring to splitting what is logically one large table into smaller physical pieces.

The partitioning is based on table *inheritance*. Each partition represents a child table of a single parent table. Normally the parent table itself is empty; it exists just to represent the entire data set. The child table inherits the structure of the parent (i.e. attributes). The partitioning of a table is processed by the *range* defined on it. In other words the table is divided into different partitions defined by a key column or set of columns, with no overlap between the ranges of values assigned to different partitions.

In details we set up a partitioned table by following steps:

1. We create the parent table, from which all of the partitions will inherit.

2. The parent table will contain no data. We can define several constraints on this table (e.g. key, foreign key and so on) to be applied equally to all partitions.

3. We choose the range from the parent table (a single attribute or a set). Then based on this range, we create several child tables that inherit the structure (i.e. attributes and constraints) from the parent table. Normally, these tables will not add any columns to the set inherited from the parent.

4. We add table’s *check* constraints to the child tables to define the allowed key values in each partition.
5. We create an index on the key column(s) in each partition (as well as any other indexes we might want).

6. Finally we define a trigger or rule to redirect data inserted into the parent table to the appropriate partition.

7. Optionally we can iterate the partitioning on the resulting child tables.

In our case we defined the following ranges:

- $C_{\text{Predicate}}$: the key column is the $Name$ attribute. It redirects data into respective partition respect to a unique value. Then we can create indexes on the $SubjectResourceOID$ and $ObjectResourceOID$ attributes.

- $C_{\text{Property}}$: the key column is the $Name$ attribute (as well as for $C_{\text{Predicate}}$) and we can create indexes on the $ResourceOID$ and $Value$ attributes.

- $C_{\text{Resource}}$: the key column is the $URI$ attribute. It redirects data into respective partition respect to a range of values (e.g. alphabetical ranges).

Let’s consider the example depicted in Figure 7. It illustrates the physical design of the running example, described in Section 2. For instance the $C_{\text{Predicate}}$ table was partitioned into two tables respect to the two key values of $Name$ as $Child$ and $Brother$. Also the $C_{\text{Resource}}$ table was partitioned into two tables respect to the range values of $URI$ (in this case respect to alphabetical ranges $[a − h]$ and $[p − z]$).
4.5 Remarks

We have described our idea of modeling RDF data at different levels: conceptual, logical and physical. Each level provides relevant benefits. The novelty introduced at conceptual level is the characterization of RDF elements. Whatever is the syntax used in an RDF documents, all the elements are represented in a uniform way. We distinguish between resources and triples; then we characterize the triples on the basis of their object and we use a specific representation to address RDF containers and their elements. At logical level, the choice of a relational implementation grants us all the well known advantages. In particular, it is possible to use several specific tools and SQL querying facilities. At physical level, partitioning gives serious benefits, mostly when tables are very large. Let’s consider query performance that can be improved dramatically in certain situations. In particular when most of the heavily accessed rows of a table are in a single partition (or in few partitions). Also update operations (e.g. deletion) exploits the partition accessing only the corresponding portion of table to update. Notice that the set of child tables are not intended to remain static. So we can remove old partitions containing disused data or add and update partitions where new knowledge is added. In this case the most relevant advantage of the partitioning mechanism is that it allows this task to be executed instantaneously by manipulating the partition structure, rather than physically moving large amounts of data. Moreover this mechanism reduces significantly the number of joins and unions that otherwise would make much more complex the query process, and in particular performs faster joins. Finally the three levels we consider define also the typology of user that can access to the system, from less expert (i.e. the conceptual level) to more expert (i.e physical level).

Other advantages of the proposed idea are listed in the following.

In RDF documents a property/predicate may appear more than once with the same subject resource but different object resource (this is called multi-valued attribute). The management of multi-valued attributes is easily implemented in our approach. At conceptual level, each object value corresponds to a new instance of the property/predicate construct. At logical level, a new instance of a construct corresponds to the addition of
a row in the construct table.

In our approach, as stated in previous sections, blank nodes are managed like other resources. A blank node differs from a common resource in the value of the URI attribute. The blank node has a fake URI equal to the internal OID representation.

Let us consider a variant of the example of Section 2, where we represent the children of a person with an RDF container (namely a Seq container). This situation is depicted in Figure 8. With our model, we can address RDF collections exploiting the constructs Container, SimpleElement and ResourceElement. In particular, we represent a collection with an instance of Resource construct (a blank node) and each element of the collection with an instance of SimpleElement or ResourceElement construct (depending on the type of elements). An instance of Predicate construct links the container with the proper resource. The logical representation is presented in the right side of Figure 8 (where we omit not relevant constructs). We define a new instance of a construct for each element of an RDF document. Therefore, every construct instance always has a value for each field (i.e. OID, properties and possibly references).

Like Abadi et al., we have a fixed structure that does not require any kind of clustering. In other words, we can refine our model, adding new constructs, but there is not the need to change already defined constructs (i.e. the number of fields of a construct will not change).

5 Experimental Results

In this section, a plenty of experiments have been done to evaluate the performance of our framework. We illustrate the RDF Benchmark chosen (i.e. a public available dataset and a set of seven representative queries). Then we compare performance and scalability of our approach with Triple and Vertical Partitioning. We will call our approach Semantic Web Information Management (SWIM).

5.1 RDF Benchmark

We used the public available Barton Libraries dataset\(^5\), provided by the Simile Project\(^6\). This dataset is a collection of RDF documents (i.e. around 11,000) containing records (formatted according to the RDF data model specifications) acquired from a dump of the MIT Libraries Barton catalog. This collection of files was derived from diverse sources. Therefore the structure of the data is quite irregular and often presents RDF malformed URIs. We converted the Barton dataset from RDF/XML syntax to triples using the Jena parser and then we made a soft cleaning of data (i.e eliminating duplicate triples and malformed URIs). The parser phase resulted a total of 60,578,683 triples. As documented by Abadi et al. [1], the dataset contains 260 predicates and 23 properties. Many of them are multi-valued: they appear more than once for a given subject. Respect to Abadi et al. we maintained triples with particularly long literal values (as the property abstract in the dataset) and triples with properties or predicate with few occurrences (as the predicate _1953). This is because we want to prove our experiments with the real unstructured nature of Semantic Web data.


Our experiments want to compare the SWIM methodology with the common Triple storage technique and in particular with the Vertical Partitioning approach, which revealed itself one of the most performing approach. Therefore we implemented the seven representative queries used by Abadi et al. The full queries are briefly described at a high level here. We illustrate the complexity of each one (also by using a graphical notation as shown in Figure 9) and later we will discuss how these queries are executed by the different approaches.

**Query 1 (Q1)** This query counts the number of different resources that are object of the predicate `type`. This requires a scan for the triples with predicate `type` and a counting for the different objects of these triples. In this case, in each triple, the object resource is the variable to be processed.

**Query 2 (Q2)** This query counts the occurrences of all properties or predicates coming out from resources that are subject of triples with predicate `type` and object a resource named `Text`. In this case the property or predicate for each triple is the variable to be processed.

**Query 3 (Q3)** Following the query Q2, Q3 counts the occurrences of all properties or predicates (and corresponding objects) coming out from resources that are subject of triples with predicate `type` and object a resource named `Text`. The complexity of this query is similar to the previous.

**Query 4 (Q4)** Starting from the counting of all properties (predicates)-object from Q3, this query recalculates these counts where the subject of triples, with predicate `type` and object a resource named `Text`, has also an outcoming property `language` with value `fre` (i.e. French). This query is thus similar to Q3, but has a higher level of result selectivity comporting a more larger space of searching.
Query 5 (Q5) Here the query results a subject $s$ and an object $o$ such that $s$ has an outcoming predicate \textit{origin} with object a resource named \textit{DLC} and an outcoming predicate \textit{records} having an object that is also subject of a triple with predicate \textit{type} and object $o$. Also this query has a high level of selectivity where resources are the variables to process.

Query 6 (Q6) This query combines Q2 and Q5. It returns all the predicates or properties coming out from a resource or subject of a triple with predicate \textit{type} and object a resource named \textit{Text} or subject of a triple with predicate \textit{records} having an object that is also subject of a triple with predicate \textit{type} and object \textit{Text}. This query presents a relevant complexity due to the high selectivity and huge amount of data to process in the dataset. Also in this case in each triple predicate and properties are variables to process.

Query 7 (Q7) This final query presents a selection of three types of resources, respectively $r_1, r_2$ and $r_3$ such that $r_1$ is the subject as of a triple with predicate \textit{point} and object a resource named \textit{end} as of a triple with predicate \textit{type} and object $r_2$ as of a triple with predicate \textit{encoding} and object $r_3$. In this case resources are variables to process.

5.2 Platform Environment

Our benchmarking system is a dual quad core 2.66GHz Intel Xeon, running Debian, with 8 Gbytes of memory, 6 MB cache, and a 2-disk 1Tbyte striped RAID array.

We implemented our experiments using Postgres. As Beckmann et al. [4] experimentally showed, it is relevantly more efficient respect with commercial database products. We used the default Postgres configuration: we preferred to no calibrate the system for specific needs.

Our Postgres implementation of the Triple storage provides a table containing three columns: \textit{subject}, \textit{property} and \textit{object}. We defined a primary key on the columns triple (subject, property, object) and used B+ tree indexes. One clustered on (subject, property, object), three unclustered respectively on subject, predicate and object\(^7\).

We implemented the Vertical Partitioning storage using one table per property. Each table contains two columns: \textit{subject} and \textit{object}. The primary key is the columns couple (subject, object). We used two indexes: a clustered B+ tree index on subject, and an unclustered B+ tree index on object.

The SWIM storage exploits the native partitioning technique of Postgres. We used three tables for \textit{Resource}, \textit{Predicate} and \textit{Property} with the relational schema described in Section 4.3. They represent the parent tables partitioned respect to the ranges described in the Section 4.4. On the Resource table (and relative partitions), we used a constraint of unique value on the \textit{URI} attribute. On the Property and Predicate tables we used an unclustered B+ tree index on the \textit{Name} attribute and for each partitions we used clustered B+ tree indexes on the \textit{SubjectResourceOID} and \textit{ResourceOID} attributes, and unclustered B+ tree indexes on the \textit{ObjectResourceOID} and \textit{Value} attributes. Each partition has a check constraint and a rule to redirect data inserted into the parent table.

In all implementations we translated textual data by hash coding to gain more performance.

\(^7\)These indexes were experimentally determined to achieve the best performing results.
5.3 Evaluating Results

Performance Results The seven queries have been tested on the three implementations and the resulting query execution times are shown in Figure 10. Each number is the average of three runs of the query. Internal database cache and operating system cache can cause false measurements then before each query execution we have restarted the database and cleaned any cache.

In the following we will explain the obtained results for the execution of each query.

Q1 The Triple approach presents a relevant response time, compared to the other approaches, due to several self-joins needed. Vertical Partitioning and SWIM, instead, have comparable results. In the Vertical approach only the \textit{type} table is accessed, performing a count operation on the different values of the \textit{object} attribute. To answer this query, SWIM has to access the join between the \textit{predicate} and \textit{resource} tables. Not the whole tables are involved in the join operation thanks to the partitioning technique. It automatically redirects to the partitions corresponding to the \textit{type} predicate and the \textit{Text} resource.

Q2 The Triple approach needs to extract all the subjects from the triple with property \textit{type} and object \textit{Text}. Then a self-join on the subject is performed to extract all other properties and count them.

The Vertical accesses the table \textit{type} selecting the subject for all the tuples with object \textit{Text}. Then a scan for all the other tables is performed to join on the subject and count. This is a typical case where the property is unknown then the Vertical Partitioning needs to scan all the database tables that refer to properties. This is due to the fact that the name of the table is the property, therefore a query to the database metadata is needed. Finally all the results are collected by a union operation.

The SWIM approach queries the predicate table and, relying the partitioning mechanism, accesses directly the \textit{type} partition to select the subjects for the object \textit{Text}. Then the previous result is used in a nested query to extract all the properties and predicates.
with the corresponding subjects. The partitioning techniques avoid to scan the whole tables (millions of tuples) and access only the portions of interest. Moreover we avoid the huge amount of unions executed by the Vertical.

**Q3** Since the query Q3 is similar to Q2, we don’t illustrate all the details. In this case the Vertical approach presents similar response time. The additional operation on the objects costs a 25% more to the Triple approach. The execution time of SWIM increases consistently due to the necessity of join with the resource table. However, also in this case, the partitioning mechanism supports the performance of this operation.

**Q4** Respect to Q3, Q4 introduces a higher level of selectivity. This allows the tree approaches to exploit the indexing mechanisms and reduce the response time. We benefit more than the others exploiting the combination between indexing and partitioning.

**Q5** In this case, Triple needs to perform three heavy self-joins. Since the properties are known, Vertical obtains the best result. It directly accesses the specific tables selecting subjects and objects. SWIM presents a comparable time but higher due to the join operations with the resource table.

**Q6** This query has the highest complexity due to the explicit union operation between the results of two subqueries. Triple dramatically suffers in this case. In both subqueries the property is unknown then Vertical needs to apply each to every table and make union of the results. This is the case where SWIM can exploit at best its logical representation combined with the physical optimization.

**Q7** Like Q1 and Q5, properties are known and the resources related to the properties should be selected. Again, in this case, Vertical is the most performing due to the aforementioned advantages.

Summarizing, the Triple approach presents the worst results, due to its unprofitable storage model. Vertical obtains best results in the queries where the properties are known (i.e. Q1, Q5, Q7) because the corresponding tables are accessed directly. The results of SWIM are better in the other cases due to its internal data organization (conceptual, logical, physical). However, in Q1, Q5, Q7 SWIM response times are comparable with Vertical.

**Scalability Results** To measure the scalability of each approach, we have performed an incremental import of data. Each Barton file have been imported separately, each of which containing around 5500 triples. The import of a single file corresponds to add new knowledge to the RDF storage. We have measured the time needed to insert the new triples in each approach and the results are presented in Figure 11. It can be seen that SWIM scales almost linearly and the results are better than the other approaches. This is due to the aforementioned peculiarity of its internal organization.

Query scalability is tested using the seven queries. After the import of each Barton file, a query is executed for all the approaches, measuring the query answer time. For instance Figure 12 shows the scalability performance of Q2. Also in this case (and similarly in the other queries), SWIM grows with a linear factor.
6 Conclusions and Future Works

Due to the growing importance of Semantic Web, several applications that use RDF data have been developed. This large amount of data must be rapidly accessed in order to be effectively used. Storing and maintaining RDF data represent crucial activities to achieve this complex purpose. The classical “triple-store” approaches are not good enough because most of the queries require a high number of self-joins on the triples table. In order to overcome these problems we proposed a model-based approach to store and maintain large amount of RDF data and showed that it achieves performance results similar to other well-known approaches. Moreover, the scalability tests performed have shown the quality of the overall approach.

The expressive power of the proposed model depends on the number of constructs we have introduced. If the need to represent new concepts arises, we have just to add proper constructs to the model. In particular, the goal of our current work involves the
management of both RDF (data level) and RDFS (schema level) via an extended model. Another issue we are studying regards the possibility to integrate the management of OWL from a data management perspective.

References


