Chapter 1
Relational Technologies, Metadata and RDF

Yannis Velegrakis

Abstract Metadata plays an important role in successfully understanding and querying data on the web. A number of metadata management solutions have already been developed but each is tailored to specific kinds of metadata. The Resource Description Framework (RDF) is a generic, flexible and powerful model which is becoming the de-facto standard for metadata representation on the Web. Its adoption has created an exponential growth of the amount of available RDF data calling for efficient management solutions. Instead of designing such solutions from scratch, it is possible to invest on existing relational technologies by exploiting their long presence and maturity. Relational technologies can offer efficient storage and high performance querying at relatively low cost. Unfortunately, the principles of the relational model are fundamentally different from those of RDF. This difference means that specialized storage and querying schemes need to be put in place in order to use relational technologies for RDF data. In this work we provide a comprehensive description of these relational RDF storage schemes and discuss their advantages and limitations. We believe that through carefully designed schemes, it is possible to achieve sophisticated high performance systems that support the full power of RDF and bring one step closer the materialization of the Semantic Web vision.

1.1 Introduction

Recent years have shown a tremendous proliferation of systems that make available on the web data and information from almost every field of human activity, i.e., from corporate environments and scientific domains to personal media and social activities. Interaction with these systems is becoming increasingly complex, mainly due

Yannis Velegrakis
University of Trento, Via Sommarive 14, 38100 Trento, Italy.
e-mail: velgias@disi.unitn.eu

Metadata	Approach
Annotations [21]	Atomic value annotations attached to a block of values within a tuple. They accompany the values as retrieved. Relational algebra query language.
Provenance [11]	Atomic data values carry their provenance, which propagates with them as they are retrieved. Query language supports predicates on provenance.
Quality Parameters [28]	Data values are associated with quality parameters (accuracy, freshness, etc.). SQL is extended to retrieve data using these parameters.
Schema & Mappings [41, 47]	Explicit modeling of schema and mapping information, and associations of it with portions of the data. SQL extension to retrieve data and metadata that satisfy certain metadata properties.
Security [7]	Credential-based access control. System reads complex security profiles and returns data results accordingly.
Super-imposed Information [26]	Loosely-coupled model of information elements, marks and links used to represent superimposed information. It has no specific schema, but is in relational model and can be queried using SQL.
Time [13]	Creation and modification time is recorded with the data and used in query answering. Query language supports predicates on time values.

Fig. 1.1: Metadata management approaches in relational and semi-structured data systems

to the fact that their internal data has dramatically increased in size, structural complexity, semantic heterogeneity and interaction intricacy. To cope with this issue, and successfully query, discover, retrieve, integrate and maintain this data, metadata plays an important role. The term metadata is used to refer to any secondary piece of information that is separate in some way from the primary data. In corporate environments, for instance, metadata regarding data quality [28, 44, 46] can help in detecting erroneous, inaccurate, out-of-date, or incomplete values and can have significant impact on the quality of query results [16]. In scientific domains, whenever data is collected from various sources, cleansed, integrated and analyzed to produce new forms of data [36], any provenance [8], superimposed information [26], different forms of annotations [11, 21] or information on the performed transformations [4, 41], can be important in order to allow users to apply their own judgment to assess the credibility of the query results.

The relational model provides a clear separation between data values and metadata information. The metadata of a relational database concerns the structure of the data (i.e., the schema), constraints that may exist on the data, statistics that are of use by query optimizers, and information about time and permissions on the various schema components. Additional types of meta-data cannot be easily incorporated in this model, and query languages, such as SQL, have no provision for meta-data

based queries, i.e., queries involving meta-data terms. Over the years, there have been numerous research efforts to cope with this limitation. Table 1.1 illustrates some of these efforts along-side the way they have approached the problem. A common denominator of these efforts is the extension of the data model with special structures to store the meta-data of interest and the extension of the query language with additional operators that are specifically designed for a specific kind of meta-data.

A closer look of the works in Table 1.1 can also reveal the high heterogeneity of the kinds of metadata that have been considered and the high degree of specialization of each solution towards the specific meta-data it is targeting. In particular, it can be observed that some metadata is expressed as single atomic values, e.g., the creation time of an element in a database [13], while others have a more complex structure, e.g., the schema mapping [41] or security [7] specification. Furthermore, metadata may be associated either to individual data values [11, 13, 28] or to groups of values, i.e., set of attributes in a tuple [21]. As far as it concerns the query language, it can be noticed that there is a clear distinction between data and metadata. Either there are queries that return only metadata information, i.e., the relations of a database schema, or data-only queries, or queries returning data accompanied by their associated meta-data. However, in many real-world scenarios, the distinction between data and meta-data is blurred. The same piece of information may be seen by some as data and by others as metadata which means that a clear distinction between the kind of arguments (data or metadata) that a query operator can accept may not be desired. It can also be observed that each solution is not directly applicable to other forms of meta-data, at least not without some major modifications. Past attempts on building generic metadata stores (e.g., [6, 23]) have employed complex modeling tools for this purpose: [23] explicitly represented the various artifacts using Telos [30], while the Microsoft Repository [6] employed data repositories (intended for shared databases of engineering artifacts). A simple, elegant approach to uniformly model and query data, arbitrary metadata and their association has been elusive.

In parallel to the database community, the Web community faced a similar need. More than ever before, a model for the uniform representation and querying of metadata on the Web had been apparent. The model had to be machine readable, flexible and easily integrateable with existing web technologies and tools. This led to the introduction of RDF that is currently emerging as the dominant standard for representing interlinked meta-data on the Web. It is a representation language based on directed labeled graphs in which nodes are called resources and edges are called properties. The graph structure is expressed through a series of triples, each representing an edge between two resources (the first and third component of the triple) with the second component of the triple being the identifier of the edge. RDF has an XML syntax, which means that it is humanly and machine readable. It allows the representation of many different kinds of meta-data with highly heterogeneous structures. Its wide adoption by organizations and individuals as the format for web metadata publishing, brings the web one step closer to the realization of the Semantic Web vision [37].

The increased amount of RDF data has naturally called for efficient storage and querying solutions, that has led to the development of a number of different systems, typically referred to as *triple stores* [3,5,10,12,22,33]. Unfortunately, the flexibility that RDF offered, came at a price. Its triple-based nature required special storage, indexing, retrieval and update techniques. Early RDF systems stored triples in giant three-column tables that were efficient for simple triple-based queries, especially in the presence of the right index structures, but had serious scalability issues in complex graph-based queries.

In order to avoid building new native RDF solutions from scratch, it seemed logical to invest on the many years of research and development that has taken place in the area of relational data management. Relational systems have matured enough and are prevalent, thus they can be easily adopted in real world application scenarios. Many popular systems such as Jena [12], Oracle [14], Sesame [10] and 3store [22], are based on this idea. Of course, the tabular nature of the relational model cannot directly serve the graph-based query expressions and data traversal functionality that RDF requires. As such, special schemas need to be designed in order to alleviate that issue and improve query performance.

In this work we provide an overview of the ways that the database and the web communities have dealt with the issue of metadata management. Section 1.2 is an introduction to the relational model. Section 1.3 presents a rough categorization of the metadata management techniques in the relational data management community. A quick overview of RDF and its basic modeling principles is made in Section 1.4. Finally, Section 1.5 presents different modeling schemes for storing RDF data in relational data management systems.

1.2 The Relational Model

Assume an infinite set of attribute names $\mathcal{L}$ and a set of atomic types, e.g., **Integer**, **String**, etc., with pairwise disjoint domains. The name of an atomic type is used to represent both the type and its domain.

A relational database (or relational instance) [15] consists of a set of relations (or relational tables). A relational table is a named set of tuples. A tuple is an ordered set of $\langle a, v\rangle$ pairs, called attributes, with $a \in \mathcal{L}$ and $v \in$ **Integer** $\cup$ **String** $\cup \ldots$ The cardinality of a tuple is the number of attributes it contains. All the tuples in the same relation must have the same cardinality, and all the attributes in the same position of each tuple must have the same attribute name and values of the same type. The schema of a relational table, denoted as $R(A_1{:}T_1, A_2{:}T_2, \ldots, A_n{:}T_n)$, describes the structure of its tuples. In particular, R is the name of the table, n is the cardinality of its tuples, and $A_k{:}T_k$ is the name and type of the attributes in the kth position of each tuple in the table. The schema of a relational database is the set of the schemas of its relational tables.

A number of different languages have been proposed over the last three decades for querying relational data. The most popular is without a doubt SQL [19]. An SQL

query is formed by a select, a from and a where clause. The from clause consists of a set of variables, each associated to a relational table. Each variable is bound to the tuples of the table with which it is associated. The where clause contains a set of conditions. Given a binding of the variables of a query, if the conditions in the where clause are satisfied by the tuples to which the variables are bound, the binding is said to be a *true* binding. For each true binding, a tuple is generated in the answer set according to the expressions in the select clause. Each select clause expression is defined by the following grammar:

$$exp ::= constant \mid variable.attributeName \mid f(exp_1, \ldots, exp_n)$$

Since the output of an SQL query is a set of homogeneous tuples, i.e., a relational table, queries can be composed to form more complex queries. SQL supports a number of additional constructs for grouping, sorting and set operations and others, but we will not elaborate further on those since they are out of the scope of the current work.

The tabular form of the relational model makes it ideal for representation of sets of data with homogeneous structures and the appropriate index structures can provide efficient tuple selection operations based on attribute values.

1.3 Modeling Metadata in Relational Systems

There have been numerous efforts for storing metadata in relational databases. These efforts boil down to three main categories. The first is the use of separate specialized structures that represent the metadata. The second is the use of intermixed structures in which the same table contains both data and metadata. The third kind is the use of intensional associations that allows metadata to be associated to data without the need for the latter to be aware of that.

1.3.1 Separate Structures

This approach involved specially designed tables whose semantics are known in advance and are not considered part of the database schema or instance. They contain meta-information about the stored data, and they can be queried using the relational query language supported by the system. However, they cannot be used for associating generic kinds of metadata to the data values. A classical example of this case are the catalog tables of the relational DBMS.

(a)

Restaurants

Name	Name_c	City	City_c
Chinatown		N.Y.	*in USA*
Pizza Roma	*very nice*	L.A.	
Pizza Roma	*expensive*	L.A.	
Pizza Roma	*from citySearch*	L.A.	*from citySearch*

(b)

Restaurants

Name	City
Chinatown	N.Y.
Pizza Roma	L.A.

Restaurants_c

Key	Name_c	City_c
Chinatown		*in USA*
Pizza Roma	*very nice*	
Pizza Roma	*expensive*	
Pizza Roma	*from citySearch*	*from citySearch*

(c)

Restaurants

Name	City	Name_c	City_c	value
Chinatown	N.Y.		1	*in USA*
Pizza Roma	L.A.	1		*very nice*
Pizza Roma	L.A.	1		*expensive*
Pizza Roma	L.A.	1	1	*from citySearch*

Fig. 1.2: Schemes for storing annotations in relational data

1.3.2 Intermixed Context

In the case in which different kinds of metadata need to be associated to specific data values, a different mechanism is needed. Data annotations are classical examples of such metadata. An annotation may vary from security information to data quality parameters, with the simplest and most prevalent kind of annotation being the user comments.

A possible scheme for data annotations is used in the DBNotes system [8]. For every attribute A, in a data table, a second attribute A_c is introduced to keep the annotation of the value in A. An example of this scheme is illustrated in Figure 1.2(a). Attributes Name_c and City_c contain the annotations of the values in attributes Name and City, respectively. A limitation of the scheme is that in the case in which a value has more than one annotation, the whole tuple needs to be repeated, once for every annotation. This is the case of the `Pizza Roma` restaurant in the example. Note how the tuple [`Pizza Roma`, `L.A.`] has to be repeated three times to accommodate the three different annotations of the value `Pizza Roma`. Furthermore, if an annotation is referring to more than one attributes of a tuple, the annotation has to be repeated for every attribute. For instance, the annotation `from citySearch` in the Restaurants table is repeated in both columns Name_c and City_c.

Restaurants

Name	Type	City	Price
Chinatown	Chinese	N.Y.	8
Pizza Roma	Italian	L.A.	30
The India	Indian	San Jose	25
Yo-Min	Chinese	N.Y.	22
Spicy Home	Indian	L.A.	25
Noodle plate	Chinese	N.Y.	15
Curry	Indian	San Jose	20

Comments

References	Comment	Date	Author
select * from Restaurants where city="L.A."	It has 7% tax	03/06	John
select * from Restaurants where city="N.Y."	Tip is 15%	06/06	Mary
select city from Restaurants where city="L.A."	Is in the US	01/04	Kathy
select * from Comments where author="John"	Not to trust	06/05	Nick

Fig. 1.3: A database instance with intensional associations

To avoid the redundancy a variation of the described scheme can be used. The variation is based on the existence of keys or of row identifiers. In particular, the scheme assumes that for every table T there is a table T_c that stores the annotations. The attributes of the table T_c are basically the attributes A_c of the previous scheme, with one additional attribute, used to reference the key (or row id) of the respective data tuple. This variation is illustrated in Figure 1.2(b). Tuple repetition may still be required in case of multiple annotations on a value, but in this case the wasted space is only for the repetition of the row identifier (or the key) and not for the whole data tuple values.

In certain practical scenarios, an annotation may need to be assigned to more than one tuple attributes as a block and not to each attribute individually. The Mondrian system [21] uses a different variation of the previous scheme to achieve this. It assumes a column A_c of type **bit** for every attribute A of the data table, and a column value, in each data table. When an annotation needs to be placed on a group of attributes in a tuple, the annotation value is inserted in the value column, and the bit columns of the respective attributes are set to 1. An example is illustrated in Figure 1.2(c) in which the annotation `from citySearch` is assigned to both values `Pizza Roma` and `L.A.`

1.3.3 Intensional Associations

The two schemes presented have three main limitations. First, they require explicit association of every metadata entry with its respective data entries. This is not practical in cases in which multiple data elements share the same metadata information. For instance, assume the existence of a table Restaurants with information about restaurants as illustrated in Figure 1.3, and a user that needs to annotate all the restaurants in N.Y. city with some comment about them. After finding all the tuples in the Restaurant table for which city="N.Y.", an explicit association will have to be made between each such tuple and the the respective comment. The second limitation is that future data cannot be handled automatically. For instance, assume that ten new New York restaurants are inserted in the Restaurants table and that the comment the user needs to add is generic and applies to every New York restaurant. An association between the user comment and each of these new restaurants will again have to be explicitly made. The third limitation of the schemes described in the previous subsections is that the data table is fully aware of the existence of the metadata. Any metadata value change to be implemented requires access to the table that the data values are also stored. This is something that may not always be possible since owners of data and meta-data may be different entities with different privileges.

To overcome these limitations, an instensional association framework [34] can be put in place. The idea is to replace the traditional value-based association between tables with associations based on queries. An example is illustrated in Figure 1.3. Table Comments contains comments that various users have made over time. Column References is of a special type that contains, instead of a regular atomic value referring to the key of the data table, a query expression. The evaluation of this query expression determines the data values that the respective comment is about. For instance, the first tuple in the Comments table is about all the restaurants in Los Angeles (L.A.). Note that one metadata tuple is enough to cover all the restaurants on which the comment applies. In real systems where the same metadata value may have to be assigned to multiple data tuples, this scheme can lead to significant saving in terms of space. Furthermore, assuming that a new restaurant opens in L.A. and the respective tuple is inserted in the Restaurants table. The tuple will be automatically associated to the first entry in the Comments table, since it will satisfy the specifications of the query in the References column of the metadata tuple.

By using queries as attributes one can assign metadata to data without any modification on the data tables. Furthermore, the select clause of the query can be used to assign the metadata information to a subset of the columns of the data tuple, as is the case with the third tuple in the Comments table in Figure 1.3. An additional feature is that the queries used as values can reference tuples in any table in the database, even in their own. One such example, is the fourth tuple of the Comments table that represents a comment that applies on the first tuple of the same table. The modeling of this scheme allows for a uniform management of data and metadata and facilitates the construction of metadata hierarchies, i.e., metadata assigned to other metadata.

Of course, the presence of queries as values require modifications of the query evaluation engine. Some first steps have already been done towards evaluation techniques [32, 39], and index structures [38].

1.4 RDF

RDF (Resource Description Framework) [22] has been introduced as a mean to represent metadata on the web. It views web data as a set of *resources*, that may also be related to each other. Any web entity is considered a resource, uniquely identified by its *Unique Resource Identifier (URI)*. Information about web entities is expressed through *RDF statements*. An *RDF statement* specifies a relationship between two resources. It is a triple of the form $\langle subject, predicate, object \rangle$. The *subject* is a URI representing a web artifact, *statement* is a label, and *object* is either another URI or a literal. The information, for instance, that W3C is the owner of the web page http://www.w3.org/RDF, can be expressed through the statement

$\langle$http://www.w3.org/RDF, "owner", http://www.w3.org$\rangle$

assuming that the URI of the W3C is http://www.w3.org.

Web metadata may not always be about web entities but about other metadata. To facilitate this kind of information, statements themselves are considered resources, thus, they can be used in statements like any other resource.

By representing every statement as an edge that connects the resources that appear in its subject and object and is labeled with its predicate, one can generate a graph representation of any RDF data. Since statements are also resources, an edge can connect not only nodes but also edges, thus, the generated graph structure is actually a hyper-graph.

A set of RDF triple statements form an *RDF base*. More formally, we assume the existence of an infinite set of *resources* $\mathcal{U}$, each with a *unique resource identifier* (URI), an infinite set of labels $\mathcal{A}$, and an infinite set of *literals* $\mathcal{L}$. Each literal can be considered a resource having itself as its actual URI. A *property* is an association between two resources which is also a resource. A property is represented by a triple $\langle s, p, o \rangle$, where p is the URI of the property while s and o the URIs of the resources it associates. The URI p of a property $\langle s, p, o \rangle$ is denoted by $URI(p)$ or simply p.

Definition 1.1. A *RDF base* Σ is a tuple $\langle I, P \rangle$, where $I \subseteq \mathcal{U}$ is a set of individuals, $P \subseteq I \times \mathcal{U} \times I$ is a set of properties and $I \cap \{URI(p) \mid p \in P\} = \emptyset$.

To describe and/or control the structure or RDF data, W3C has introduced *RDF Schema* (RDFS) [42]. RDFS is a set of constructs that allows the definition of schematic, i.e., typing, information about RDF data. It defines classes as a way to group together RDF resources that have common structures, differentiates between a property and a non-property resources, allows the definition of inheritance and enables the definition of constraints on the resources that a property is allowed to associate.

RDFS assumes that set $\mathcal{U}$ contains the following special resources:[1] **rdfs:Literal**, **rdfs:Property**, **rdfs:Class**, **rdf:Thing**, **rdfs:type**, **rdfs:domain**, **rdfs:range**, the **rdfs:subClassOf** and the **rdfs:subPropertyOf**.

Typing information is modeled through the **rdfs:type** property. All the resources associated to a resource C through the **rdfs:type** property are said to be *instances* of C. Every resource is an instance of the resource **rdf:Thing**. Each resource C that can have instances is itself an instance of the resource **rdfs:Class**. All such instances are referred to as *classes*. Resources **rdfs:Property**, **rdfs:Literal**, **rdfs:Class** and **rdf:Thing** are classes. The instances of the class **rdfs:Literal** are all the literal values. Every instance of a class that is not an instance of class **rdfs:Property** is referred to as *individual*. A partial order can be defined among classes, through properties that are instances of **rdfs:subClassOf**. If a class is a subclass of another, then the set of instances of the first is a subset of the set of instance of the second. The subclass relationship forms a lattice hierarchy among the classes, with class **rdf:Thing** being at the top of the lattice.

Every property resource is an instance of a *property class*. Property classes are used to restrict the resources properties can associate. A *property class* is an instance of class **rdfs:Property** and is associated through properties **rdfs:domain** and **rdfs:range** to two other classes. When a property is an instance of a property class, the resources it associates must be instances of the domain and range classes of its property class.

A partial order sub-property relationship can be defined between the property classes, similarly to the subclass relationship that can be defined among classes. The sub-property relationship is defined through the **rdfs:subPropertyOf**. When a property is a sub-property of another, then the domain and range classes of the first must be subclasses of the domain and range of the second.

Definition 1.2. A *RDF/RDFS base* Σ is a tuple $\langle I, P, C, P_c, \tau, \tau_c, \dot{\preceq}_c, \dot{\preceq}_p \rangle$, where $I \subseteq \mathcal{U}$ is a set of individuals, and P is a set of properties with $P \subseteq I \times \mathcal{U} \times I$ and $I \cap \{URI(p) \mid p \in P\} = \emptyset$. C is a set of classes that includes **rdfs:Class**, P_c is a set of property classes that includes **rdfs:Property**, $\tau | I \rightarrow C$ and $\tau_p | P \rightarrow P_c$ are typing function, $\dot{\preceq}_c$ is a partial order relationship[2] on C with **rdfs:Class** as the root, and $\dot{\preceq}_p$ is a partial order relationship[3] on P with **rdfs:Property** as root.

1.5 Using Relational Systems for RDF Storage

The current RDF storage and retrieval landscape reminisces XML. Although there are proposals for native RDF storage, a great majority has opted towards the use

[1] The names of the properties are their URIs. For simplicity, name-spaces have been omitted from the discussion.

[2] Representing the subclass relationships

[3] Representing the sub-property relationships

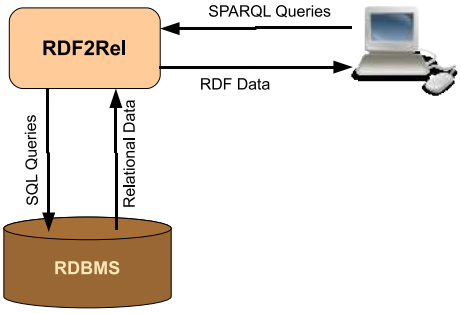

Fig. 1.4: The generic architecture of systems that employee relational data management solutions for storing, querying and retrieving RDF data.

of relational data management systems. This trend comes at no surprise. Development of new solutions from scratch requires significant amount of effort, time and money. Relational technologies, on the other hand, have been around for more than three decades, have great achievements to demonstrate and are dominating the market. They can offer numerous of-the-shelf solutions with great query performance and good scalability. Unfortunately, using a relational system for that purpose is not a straight forward task, mainly due to the different foundational principles between the RDF and the relational model. Special schema design and RDF-tailored query answering techniques need to be put in place for such a coupling to work. The following sections provide an overview of the different schemes that can be used for that purpose. Many of them are already adopted by major RDF stores, such as Jena [12], Oracle [14], RDFStore [3], Sesame [10], 3Store [22], DLDB [33] or Hexastore [45]. A typical architecture consists of a relational system with an additional layer that stands between the relational repository and the user or application interface. The layer is aware of the storage scheme used in the relational repository and is responsible for translating the RDF queries posed by users or applications to queries on the relational structures. The architecture is graphically depicted in Figure 1.4.

The length of the URIs is one of the first issues faced in relational RDF storage. A URI typically consists of a web address followed by some directory path and the name of the intended resource. Although storing and retrieving long strings is not an issue in modern DBMS, it is becoming an issue in indexing. Many indexing mechanisms use a maximum of 256 or 512 characters as key values, thus, only the first 256 or 512 characters of the URIs will be used in the index. Truncating URIs makes the index useless since the truncated URIs may become non-unique. To avoid this issue, URIs can be stored in a reverse form, so that any possible truncation occurs only on its head. Alternatively, URIs can be mapped to unique system generated identifiers, preferably of numerical nature. Comparisons on numerical values are more efficient than string comparisons, thus, such a mapping offers significant performance improvements. In what follows, we assume that such a mapping is always possible and we will treat URIs and their respective system generated identifiers as synonyms.

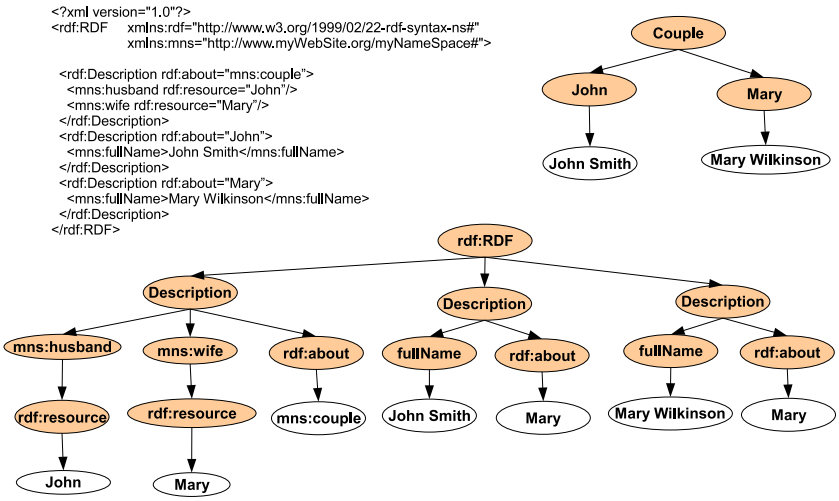

Fig. 1.5: XML syntax (top left) of some RDF structure (top right) and the tree representation of its XML syntax (bottom)

A similar technique based on hash encoding is used by 3Store [22]. In particular, for every URI or atomic value that exists in the database, a hash key is computed and used instead of a URI or a literal. The selection of the right hash functions can significantly reduce the time required to compare two values/URIs, and can lead to considerable query performance improvements.

1.5.1 Storing RDF as XML

Since RDF has an XML syntax, any XML storage mechanism can be used to store RDF data. Currently, there is a rich literature addressing the issue of storing and querying XML data in relational databases. Several mapping strategies have been proposed, and many systems have already been developed, such as, Stored [18], Edge [20], Interval [17], XRel [49], XPERANTO [40], or LegoDB [9]. Furthermore, most major commercial relational DBMSs, e.g., IBM DB2 [48], Oracle [24] and Microsoft SQL Server [29], are currently supporting the shredding, storing, querying and retrieving of XML documents.

There are two main drawbacks in following such a strategy. The first is the mismatch between the syntactic variations of the XML representation and the RDF data model. In particular, the tree representation of the XML syntax of some RDF structure may involve additional elements that may not correspond to any of the RDF structures. For instance, Figure 1.5 illustrates an RDF model represening a couple where John Smith is the husbant and Mary Wilkinson the wife. The top left-hand-side of the figure is the XML syntax of the RDF model, while the top right-hand side

is its graph representation. The lower part of the figure is the tree representation of the XML data of the top left-hand side. It can easily be noticed the differences between the two graph representations, which means that queries on the RDF structure will have to be translated accordingly if the RDF structures are stored according to the XML representation.

A second drawback of the strategy is the mismatch between the XML and the RDF query patterns. XML queries are based on the tree structure of XML and typically involve root-to-element paths or subtree selections. RDF resources, on the other hand, lack such a hierarchy. They are structured as directed graphs, and naturally, RDF queries typically involve random graph traversals.

1.5.2 Vertical Table

Since an RDF base is expressed as a set of triple statements of the form ⟨subject, predicate, object⟩, a straight-forward solution is to use a 3-attribute table for storing the statements. This schema is referred to as the *vertical table* scheme. Each tuple in a vertical table represents a triple statement, with its three attributes corresponding to the subject, the predicate and the object part of the triple, respectively. Figure 1.6 illustrates a vertical table that models a part of an RDF base.

Assume that a user is interested in finding whether `Calvanese` has any publications and poses the SPARQL query:

select ?subject **where** { ?subject `author` `Calvanese` }

To retrieve the answer, the SPARQL query is translated to the folowing SQL expression:

select subject
from Statements
where predicate="`author`" **and** object="`Calvanese`"

which can be efficiently answered in the presence of indexes on the attributes of the table Statements.

On the other hand, to answer queries like, for instance, the one that looks for the journals in which `Mecca` and `Atzeni` have sent a publication together, requires multiple self-joins over the Statements table which leads to significant performance issues. The particular query, for instance, can be answered through the relational query:

select s3.object **from** Statements s1, Statements s2, Statements s3
where s1.predicate="`author`" **and** s1.object="`Mecca`" **and**
s2.predicate="`author`" **and** s1.object="`Atzeni`" **and**
s3.predicate="`journal`" **and** s1.subject=s2.subject **and**
s1.subject=s3.subject

The advantage of the vertical table scheme is that it is suitable for storing highly heterogeneous data. It can easily model resources with any number of properties.

Statements

subject	predicate	object
1	type	article
1	title	LAURIN
1	author	Catarci
1	author	Santucci
1	author	Calvanese
1	journal	WWW
1	year	2001
1	file	CSC01.pdf
2	type	book
2	author	Codd
2	title	The Relational Model
2	year	1990
2	publisher	Addison-Wesley
3	type	book
3	title	Principles of DB Systems
3	author	Vianu
3	author	Hull
3	author	Abiteboul
3	year	1995
3	publisher	Addison-Wesley
4	type	article
4	title	Web-Based Data
4	author	Atzeni
4	author	Merialdo
4	author	Mecca
4	journal	IEEE Int. Comp.
4	file	AMM02.pdf

Fig. 1.6: A vertical table.

The scheme facilitates statement-based queries, i.e., queries consisting of a triple statement that lacks one or two parts and returns the resources of the triples complementing the missing parts, like, the example query about `Calvanese` above. This is because statement-based queries get translated to selectivity relational queries which can be efficiently answered in the presence of the right index structures. On the other hand, due to the joins that need to be performed, data browsing and path queries on the RDF structures are costly to implement.

In the presence of schema information, the vertical table can be used to answer schema queries, i.e., queries related to RDF metadata. The scheme stores the schema information in triples as it does with the rest of the data [27]. Thus, it is easy, for instance, to answer whether a class is a direct subclass of another. It will be hard, however, to answer whether a class is a subclass of another in the general case, since the system will have to search and combine many triple statements.

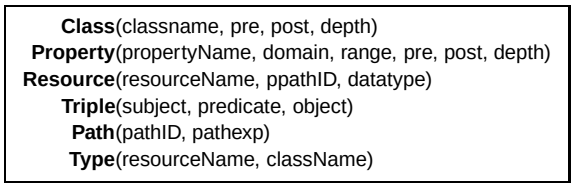

Fig. 1.7: Relational schema for a graph-based RDF storage

1.5.3 Graph-Based Storage

Since the RDF model is basically a directed graph, a large number of queries are about detecting subgraphs satisfying some path expressions. The vertical table scheme has poor performance for this type pof queries because they require multiple join operations.

To overcome this problem, the design of the relational tables can be based on the paths that are most likely to be used [27]. To cover all the possible cases, one can extract for each resource *e* all the path expressions to every other resource reachable from *e* and explicitly store them. This precomputation avoids expensive joins during query answering and improves significantly the query response time. An important requirement, however, is that the data contains no cyclic paths.

To reduce the size of the tables, an idea is to use different tables to represents different parts on the RDF graph [27]. In particular, one can extract different subgraphs based on the class hierarchy, the properties, etc., and store each one of them in its own table. That way, depending on their kind, queries will be directed and answered on the respective tables. The extraction of the subgraphs can be based on the special RDF properties. For instance, the *class*, *inheritance*, *property*, *type*, *domain-range* and generic subgraphs can be extracted from the RDF graph based on the properties rdf:subclassOf, rdfs:subPropertyOf, rdf:type, rdfs:domain/range and rdfs:seeAlso/isDefinedBy relationships. Clearly, the structure of each such graph is much less complex that the structure of the whole RDF graph which leads to better response times. Furthermore, based on the characteristics of each such graph, different techniques can be used for the representation of the different graphs in the relational tables.

A possible schema of the relational tables for a graph-based RDF storage is illustrated in Figure 1.7. Relations Class and Property are used to store the classes and properties of the RDF Schema. Attributes pre, pos and depth represent the Li and Moon [25] encoding of each class/property in the RDF graph. The Li and Moon encoding is a numbering scheme that has been extensively used in the XML literature to check ancestor/decedent relationships. It assigns to each node three numbers. The first and the second, represent the pre-order and the post-order position of each node in the XML tree. The third determines the depth of each node, computed as the length of the path from the root to the specific node. Trying to apply the specific numbering scheme on an RDF graph gives rise to two main issues. First, since RDF graphs are directed acyclic graphs and not trees, it is not clear what node should

serve as the root from which the numbering should start. Second, multiple different paths may have a common start and end nodes, and choosing one over another may have conncequences. To overcome these issues, nodes with in-degree equal to 0 can be characterized as *"roots"* and paths can be computed from each such node to any other. Furthermore, in the case of multiple paths to a node, multiple copies of that node can be created, one for each different path.

In the relational schema of Figure 1.7, table Triple is used to model all the RDF statements and is used to efficiently answer predicate queries. Table Type is used to record for every resource instance the RDF schema class it belongs. The attributes of these two tables are self-explanatory.

Materializing the different paths from the roots to the nodes is also an issue, since the paths can be of variable size. To avoid expensive join operations, every path can be encoded as a string of labels and resource names separated through some special character, i.e., "/". The role of table Path and Resource is to materialize this encoding. The former encodes every path along with a unique identifier. The latter, associates every node, i.e., resource, in the RDF graph with its paths in the Path table.

Finding the resources reachable through a path can be answered by selecting from the Path table those tuples for which the pathexp attribute is equal to the path specified in the query. This selection can be performed fast if a Hash or B+ tree index is available. Since the paths recorded in pathexp are all paths starting from a root, the specific selection can work only if the path specified in the query also starts from a root. If it starts from a non-root node, then the table Path needs to be searched for the tuples with a pathexpr attribute value ending with the requested path. Such an operation cannot exploit the index, and is based on sequential scan. To avoid sequential scan, paths can be stored in pathexp in reverse order, i.e., starting from the ending node and ending with the root. That way, the index can be exploited to improve searching for paths ending to a node.

1.5.4 Graph Schema - Vertical Data

An approach similar to the graph-based storage has been used in RDFSuite [3], but with further improvements and features. The general idea is to exploit to the maximum the RDF Schema information whenever this is available. Two sets of relational tables are needed. The first is used to represent the RDF Schema information and the second the actual RDF data.

To represent the RDF schema information, and in particular the class and property hierarchies, four tables are used, namely the Class, Property, SubClass and SubProperty. The Class and Property tables hold the classes and properties defined in the RDF Schema, respectively. The SubClass and the SubProperty tables, store the `rdfs:subClass` and `rdfs:subProperty` relationships that exist between the classes and the properties, respectively. A special table Type is used in which the build-in classes defined in RDF Schema [42], i.e, `rdfs:Class` or `rdfs:Property`, and the literals, i.e., `string` or `integer`, are hard-coded into its contents. The

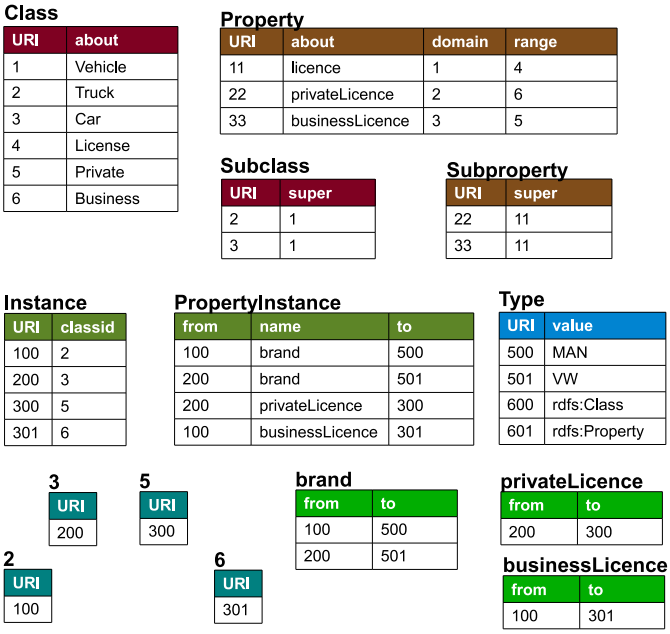

Fig. 1.8: A Graph Schema Vertical Data encoding example.

schema of the tables along with a small fraction of the table contents are illustrated in Figure 1.8. Note that table Class is a unary table, while tables SubClass and SubProperty are binary since they store the relationship between the classes and properties that exist in the tables Class and Property, respectively. The Property table on the other hand is a ternary relationship since it provides for every property, the domain and range classes. B^+-tree indexes are required on every attribute in these tables to achieve satisfactory performance.

For representing the RDF data, an idea similar to the vertical table scheme can be used. In particular, a table Instance with two attributes is required. The first attribute of a tuple in that table keeps the URI of an instance resource and the second the URI of the class that the instance belongs. A similar approach is used to represent the properties of the instances, through a Property table with three columns: two for keeping the URIs of the instances that the property relates, and one for storing the name of the property. A fraction of the Instance and PropertyInstance tables is illustrated in Figure 1.8. As before, B^+-tree indexes are constructed for every attribute in these tables.

The Instance and the PropertyInstance tables may get too large and affect performance. An alternative modeling is to use one table for the instances of each different class. The name of each such table is the name of the class, and the tuples it contains are the URIs of the instances of the respective class. The tables in Figure 1.8 that have a number as a name are examples of this variation. The same applies for

the property instances, but this time the table is labeled with the name of the property and it must have two columns that record the two resources that the property instance associates.

Having separate tables for instances and properties of different classes may lead to more efficient query answering since fewer tuples may have to be scanned. However, this option is not applicable in cases that the number of classes are extremely large, since this will require the definition of an extremely large number of tables. Although the number of tables supported by the majority of modern relational systems is large enough, there are RDF bases that require the creation of more than 300.000 tables, which even if supported by the relational system, it will have serious performance and management issues.

1.5.5 Property Table

An alternative to the vertical table schema that aims at reducing the number of joins required during query answering is to cluster the properties of the resources and find groups of entities that share the same set of properties. The common properties can then be modeled as attributes of the same table, referred to as a *property table*. This modeling makes explicit the association of the different properties, eliminating the need for joins. A property table consists of N attributes. The first functions as a key and is typically a URI. The remaining $N-1$ attributes represent a set of $N-1$ properties that are commonly found to all the resources, or at least the majority of them. For instance, by studying the RDF data of Figure 1.6, one can notice that properties type and title appear to all the four resources, while the property year appears in the majority of them, i.e., it appears in all resources apart from 4. Thus, we can construct a relational table with four attributes, the first corresponding to the resource URI and the rest to the attributes type, title and year. The table is illustrated on the left-hand side of Figure 1.9.

Note that resource 4 in Figure 1.6 has no year property, and this is reflected in the Property table of Figure 1.9 through a NULL value on the respective attribute. The property table could have also included attributes journal and file for storing the data of the respective properties. However, since these properties appear to only few resources, they would have required the respective columns to be padded with a lot of nulls, resulting to an unjustified waste of space. To avoid this, a different table, referred to as the *excess table*, is instead introduced. Its schema and functionality is the same as the table in Figure 1.6. The context of this table are shown on the right-hand side of Figure 1.9.

A requirement of the property table scheme is that the properties used in the property table are not multi-valued. In the RDF example of Figure 1.6, certain resources have more than one author. Since the relational model is in First Normal Form [2], such multi-values attributes cannot be modeled in the property table, and unavoidably will be included in the excess table.

Property Table

URI	type	title	year
1	article	LAURIN	2001
2	book	The Relational Model	1990
3	book	Principles of DB Systems	1995
4	article	Web-Based Data	NULL

Excess Table

URI	predicate	object
1	author	Catarci
1	author	Santucci
1	author	Calvanese
1	journal	WWW
1	file	CSC01.pdf
2	author	Codd
2	publisher	Addison-Wesley
3	author	Vianu
3	author	Hull
3	author	Abiteboul
3	publisher	Addison-Wesley
4	author	Atzeni
4	author	Merialdo
4	author	Mecca
4	journal	IEEE Int. Comp.
4	file	AMM02.pdf

Fig. 1.9: A property table scheme.

A variation of the property table scheme is one that considers multiple property tables. Howe many such tables and with what attributes is something that is determined by the property clustering technique. For instance, by noticing that although there is no journal property for every resource in the RDF data, for those resources that exist, property file is also present. Thus, a second property table can be created for only the properties journal and file.

The advantage of the property table scheme is that queries involving popular attributes in the cluster, can be answered without joins. For instance, asking for the article with the title "LAURIN" published in 2001, can be answered with a selection query on the property table. Of course, queries involving attributes that have not been modeled in the property table and are located in the excess table will unavoidably require joins.

The property table scheme makes no use of schema information, thus it can be used for schema-less data. However, the existence of RDF Schema can offer new opportunities for optimizing the encoding. In particular, since classes are used to cluster together resources with the same structure, one could use the classes as a guide for generating multiple property tables, for instance, one for each class. The properties encoded in each such table will be the properties of the respective class, and the table will be referred to as a *class property table*. For example, assuming that for the RDF data of Figure 1.6 there are two main classes, one for articles and one for books. The data can then be represented into two class property tables as illustrated in Figure 1.10. Note that even in this case, the multi-valued properties continue to be an issue and still require an excess table in order to be stored. On the other hand, all the other properties will be accommodated to one or more property tables. The case in which a property is modeled in more than one property tables, is

Article Property Table

URI	type	title	year	file	journal
1	article	LAURIN	2001	CSC01.pdf	WWW
4	article	Web-Based Data	NULL	AMM02.pdf	IEEE Int. Comp.

Book Property Table

URI	type	title	year	publisher
2	book	The Relational Model	1990	Addison-Wesley
3	book	Principles of DB Systems	1995	Addison-Wesley

Excess Table

URI	predicate	object
1	author	Catarci
1	author	Santucci
1	author	Calvanese
2	author	Codd
3	author	Vianu
3	author	Hull
3	author	Abiteboul
4	author	Atzeni
4	author	Merialdo
4	author	Mecca

Fig. 1.10: A property table scheme with class property tables.

the one in which the property is shared by more than one classes. For instance, the property title appears in both property tables of Figure 1.10. This is a fundamental difference between the property tables based on the RDF Schema and the property tables based on property clustering. In the latter case, no property gets repeated in more than one table.

Although the property table scheme may be proved to be efficient for many applications, there are cases in which it may under-perform. One of these cases is the one in which the data from different class property tables needs to be combined. For instance, assume that one is interested in finding the years in which publications have taken place. In the case of class property tables, this will translate to the union query that selects the years from the two class property tables. The scheme performs well if the data is highly structured data, i.e., conforms to some schema. This minimizes, and in the best case eliminates, the number of properties that are recorded in the excess table. However, we should not forget that one of the main reasons of the RDF popularity is its ability to model highly heterogeneous data, which means that a large majority of the data of interest is of such nature.

1.5.6 Vertical Partitioning

The property table clusters together properties that are common to a group of resources, but those properties that cannot be accommodated in any cluster will have all to be stored in the excess table. A different scheme that aims at tackling this limitation is the vertical partitioning. The idea of the vertical partitioning scheme [1] is similar to the idea of column-store in databases. It groups together properties of the same type and store each such group in a separate table. These tables can be linked together based on the resource URIs. More specifically, all the subject-predicate-object triples that have the same predicate value form a group stored under a two-column relational table named after the name of the predicate. The first column is

type

URI	value
1	article
2	book
3	book
4	article

title

URI	value
1	LAURIN
2	The Relational Model
3	Principles of DB Systems
4	Web-Based Data

year

URI	value
1	2001
2	1990
3	1995

author

URI	value
1	Catarci
1	Santucci
1	Calvanese
2	Codd
3	Vianu
3	Hull
3	Abiteboul
4	Atzeni
4	Merialdo
4	Mecca

journal

URI	value
1	WWW
4	IEEE Int. Comp.

publisher

URI	value
2	Addison-Wesley
3	Addison-Wesley

file

URI	value
1	CSC01.pdf
4	AMM02.pdf

Fig. 1.11: A vertical partitioning scheme example.

the URI of the resource, i.e., the subject, and the second is the value of the property, i.e., the object. In total, there will be k such tables, where k is the number of different properties that the RDF data contains. A vertical partitioning example for the RDF data of Figure 1.6 is illustrated in Figure 1.11.

URI attributes cannot be keys for the tables in the vertical partitioning scheme, unless the represented property is not a multi-valued property. However, since joins are based on the URIs, the existence of indexes on all the URI attributes is required. Furthermore, indexes can also be constructed on the value attribute for facilitating selection based on property values. If the tuples in each table are sorted by the URI attribute, then joins between two tables can be performed in an efficient manner using merge-joins [35]. This also means efficient handling of multi-valued properties since all the entries of a multi-valued property will be stored consecutively in the respective property tables.

The defragmentation of the set of triples into sets that are stored into separate tables can significantly improve query answering for queries that require searching to only few properties, since the data access is needed to only the tables of interest.

Despite its advantages, the vertical partitioning scheme is not free of limitations. The most critical one is that queries with conditions on several properties will require joins of multiple tables, which although can be speed-up with the right indexing and join technique, it is not as efficient as sequential access to tables that contain all the attributes of interest. Another limitation is that the names of the properties are not recorded as data, but as meta-data in the names of the individual tables. This means that the data can be access only if the names of the respective properties are known. Thus, queries requiring the discovery of all the properties that a specific resource may have, cannot be directly answered. A solution is to use the meta-data information offered by relational DBMSs, in particular the catalog tables, in order to obtain a list of the available tables. Even with such a knowledge though, to find all the properties that a particular resource may have, will require a separate query to be sent to every table in the database in order to discover whether it contains a tuple for the specific resource of interest.

1.5.7 Smart Indexing

In recent years, a new query model has emerged on the web. Users may not have complete knowledge of the data they are querying since it is highly heterogeneous and its structure difficult to communicated. In such an environment, queries are mainly of exploratory nature and, in some cases, underspecified. For this kind of queries, schema-based approaches like the vertical partitioning or the property tables may not be the preferable solution.

For efficiently answering queries that look for properties of a given resource, the triple nature of RDF can be exploited to build specialized indexes [45]. Since RDF data is described by a list of triples of the form ⟨*subject*, *predicate*, *object*⟩, there are 6 different ways one can retrieve that data, each corresponding to one of the 3! different ways that the components *subject*, *predicate* and *object*, can be combined. For example, one combination is to first provide a *subject*, retrieve all the *predicate*s of that *subject*, and for each such *predicate* retrieve the *object*s of the triples that have the specific *subject* and *predicate*. Given the six different ways the three components can be combined, there are six different indexes that can be constructed, one for each combination. Each such index, consists of a list of lists of resources. A graphical illustration of such a structure is depicted in Figure 1.12. Assume that this figure is the index for the combination *subject-predicate-object*. Since the *subject* is first, the first horizontal list has as many elements as the different *subject* values that can be found in the RDF data triples. Each element corresponds to one such value *s* and points to another list. Since *predicate* is the second component in the *subject-predicate-object* combination, the pointed list consists of as many elements as the number of different *predicate* values that are found in the RDF data triples with *s* as a *subject*. Each of these elements corresponds to one *predicate* value *p* and points also to a list. That list contains all the *object* values that can be found in the RDF data triples that have *s* as a *subject* and *p* as a *predicate*. This kind of lists are those depicted vertically in Figure 1.12.

An index structure like the one described above is constructed for each of the six different combinations of *subject*, *predicate* and *object*, in order to cover all the possible orders they may be queried. Each index has all the information that can be found in the list of triples, thus no additional storage is required. Since each index hold all the triple information, one would expect the total required space to be six times the size of the set of triples. This is not actually true. The reason is that the lists of the third component in a combination are the same independently of the order of the first two components, thus, there is no need for storing them twice. For instance, the lists for the *object* values in the combination *predicate-subject-object* is the same as the list of *object* values in the combination *subject-predicate-object*.

The materialization of a specific combination of the index structure in a relational system requires a number of two-column and a number of single-column tables. In particular, a two-column table is needed to model the list for the first component of the combination, i.e., the top horizontal list in Figure 1.12. The first column of the table contains the values of the component that the list represents, and the second column contains the reference to the table that models the respective list of the

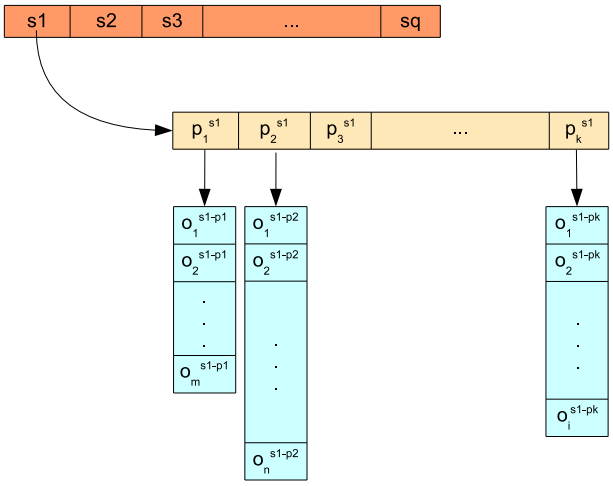

Fig. 1.12: One of the six triple-based indexing structures.

second component. The latter has also a two-column structure. The first column contains the values of the second component in the combination. In each tuple of that table, the second attribute is a reference pointer to a table materializing the list with the respective values of the third component, i.e., the materialization of the vertical lists depicted in Figure 1.12.

Among the advantages of this scheme is its natural support for multi-valued properties, the lack of any need for nulls, and the reduced number of I/O operations to access the data of interest. The main limitation, however, is the space. Every value in the triples is indexed twice. For instance, an *object* value of a triple is indexed by the *subject-predicate-object* and by the *predicate-subject-object* combination structure.

The RDF-3X [31] system is following a similar approach for the management of RDF data. RDF-3X is not based on relational technology. It has been built from scratch, and is tailored specifically to RDF data. It stores RDF data as a list of triples, as in the case of a vertical table (ref. Section 1.5.2), and builds on top of it a series of specialized indexes for the various query operators. The list of triples, however, can easily be stored in a relational database if needed. Like hexastore, it builds indexes for all the six different permutations of the three dimensions that constitute an RDF triple, but it goes beyond this by constructed additional indexes oven aggregation operators on them. The constructed indexes can be compressed in an efficient way and can lead into a space requirement that is less than the one of the actual triple data. Query answering in RDF-3X is also based on merge-join operators performed over sorted index lists.

1.6 Conclusion

The goal of this chapter was to explore the links between metadata management, RDF and relational technologies. The first part introduced relational systems and presented the main directions that have been followed in storing and querying metadata in such systems. The second part introduced RDF which is the emerging standard for metadata representation on the web. It presented the rational of its introduction, the reasons for its success and its main modeling principles. It was recognized that building native RDF solutions from scratch requires a lot of effort and under the current rate that the web evolves, any delay is a luxury the community cannot afford. Efficient and effective solutions are needed right now. The exploitation of the relational technology appears as a promising option. Relational systems have been around for decades. They are mature enough, easily accessing and offer great performance and scalability. However, the different principles between the RDF and the relational model makes the use of relational systems for RDF storage and retrieval a complicated task. We presented alternative schemes that have been proposed in the scientific literature, and we have described the advantages and disadvantages of each one. From the descriptions it is becoming clear that there is no such thing as golden rule or best solution. Each technique is best suited for certain cases. The decision on which technique one could use, highly depends on the characteristics of the RDF data to be stored and the kind of queries that are to be asked.

References

1. Abadi, D.J., Marcus, A., Madden, S., Hollenbach, K.J.: Scalable Semantic Web Data Management Using Vertical Partitioning. In: Proceedings of the International Conference on Very Large Data Bases (VLDB), pp. 411–422 (2007)
2. Abiteboul, S., Hull, R., Vianu, V.: Foundations of Databases. Addison-Wesley (1995)
3. Alexaki, S., Christophides, V., Karvounarakis, G., Plexousakis, D.: On Storing Voluminous RDF Descriptions: The Case of Web Portal Catalogs. In: Proceedings of the International Workshop on the Web and Databases (WebDB), pp. 43–48 (2001)
4. Alexe, B., Tan, W.C., Velegrakis, Y.: STBenchmark: towards a benchmark for mapping systems. Proceedings of VLDB Journal **1**(1), 230–244 (2008)
5. Beckett, D.: The design and implementation of the Redland RDF application framework. Computer Networks **39**(5), 577–588 (2002)
6. Bernstein, P.A.: Repositories and Object Oriented Databases. ACM SIGMOD Record **27**(1), 88–96 (1998)
7. Bertino, E., Castano, S., Ferrari, E.: On specifying security policies for web documents with an XML-based language. In: Proceedings of the Symposium on Access control Models and Technologies (SACMAT), pp. 57–65 (2001)
8. Bhagwat, D., Chiticariu, L., Tan, W.C., Vijayvargiya, G.: An Annotation Management System for Relational Databases. In: Proceedings of the International Conference on Very Large Data Bases (VLDB), pp. 900–911 (2004)
9. Bohannon, P., Freire, J., Roy, P., Siméon, J.: From XML Schema to Relations: A Cost-Based Approach to XML Storage. In: Proceedings of the International Conference on Data Engineering (ICDE), pp. 64– (2002)

10. Broekstra, J., Kampman, A., van Harmelen, F.: Sesame: An Architecture for Storin gand Querying RDF Data and Schema Information. In: Proceedings of the Spinning the Semantic Web Conference, pp. 197–222 (2003)
11. Buneman, P., Khanna, S., Tan, W.: On Propagation and Deletion of Annotations Through Views. In: Proceedings of the Symposium on Principles of Database Systems (PODS), (2002)
12. Carroll, J.J., Dickinson, I., Dollin, C., Reynolds, D., Seaborne, A., Wilkinson, K.: Jena: implementing the semantic web recommendations. In: Proceedings of the International World Wide Web Conference (WWW), pp. 74–83 (2004)
13. Chawathe, S., Abiteboul, S., Widom, J.: Representing and Querying Changes in Semistructured Data. In: Proceedings of the International Conference on Data Engineering (ICDE), pp. 4–19 (1998)
14. Chong, E.I., Das, S., Eadon, G., Srinivasan, J.: An Efficient SQL-based RDF Querying Scheme. In: Proceedings of the International Conference on Very Large Data Bases (VLDB), pp. 1216–1227 (2005)
15. Codd, E.F.: A Relational Model of Data for Large Shared Data Banks. Communications of ACM **13**(6), 377–387 (1970)
16. Dasu, T., Johnson, T.: Exploratory Data Mining and Data Cleaning. Wiley Publishers (2003)
17. DeHaan, D., Toman, D., Consens, M.P., Özsu, M.T.: A Comprehensive XQuery to SQL Translation using Dynamic Interval Encoding. In: Proceedings of the ACM International Conference on Management of Data (SIGMOD), pp. 623–634 (2003)
18. Deutsch, A., Fernández, M.F., Suciu, D.: Storing Semistructured Data with STORED. In: Proceedings of the ACM International Conference on Management of Data (SIGMOD), pp. 431–442 (1999)
19. Eisenberg, A., Melton, J., Kulkarni, K.G., Michels, J.E., Zemke, F.: SQL: 2003 has been published. ACM SIGMOD Record **33**(1), 119–126 (2004)
20. Florescu, D., Kossmann, D.: A performance evaluation of alternative mapping schemes for storing XML in a relational database. Tech. rep., INRIA (1999)
21. Geerts, F., Kementsietsidis, A., Milano, D.: MONDRIAN: Annotating and querying databases through colors and blocks. In: Proceedings of the International Conference on Data Engineering (ICDE) (2006)
22. Harris, S., Shadbolt, N.: SPARQL Query Processing with Conventional Relational Database Systems. In: Proceedings of the International Conference on Web Information Systems Engineering (WISE), pp. 235–244 (2005)
23. Jarke, M., Gallersdorfer, R., Jeusfeld, M.A., Staudt, M.: ConceptBase - A Deductive Object Base for Meta Data Management. Journal of Intelligent Information Systems **4**(2), 167–192 (1995)
24. Krishnamurthy, R., Kaushik, R., Naughton, J.F.: XML-SQL Query Translation Literature: The State of the Art and Open Problems. In: Proceedings of the XML Database Symposium (XSym) , pp. 1–18 (2003)
25. Li, Q., Moon, B.: Indexing and Querying XML Data for Regular Path Expressions. In: Proceedings of the International Conference on Very Large Data Bases (VLDB), pp. 361–370 (2001)
26. Maier, D., Delcambre, L.M.L.: Superimposed Information for the Internet. In: International Workshop on the Web and Databases (WebDB), pp. 1–9 (1999)
27. Matono, A., Amagasa, T., Yoshikawa, M., Uemura, S.: A Path-based Relational RDF Database. In: Proceedings of the Australasian Database Conference (ADC), pp. 95–103 (2005)
28. Mihaila, G., Raschid, L., Vidal, M.E.: Querying “Quality of Data” Metadata. In: Proceedings of the IEEE META-DATA Conference (1999)
29. Microsoft support for XML. Http://msdn.microsoft.com/sqlxml
30. Mylopoulos, J., Borgida, A., Jarke, M., Koubarakis, M.: Telos: Representing Knowledge About Information Systems. ACM Transactions on Database Systems (TODS) **8**(4), 325–362 (1990)
31. Neumann, T., Weikum, G.: RDF-3X: a RISC-style engine for RDF. Proceedings of VLDB Journal **1**(1), 647–659 (2008)

32. Neven, F., Bussche, J.V., Gucht, D.V., Vossen, G.: Typed Query Languages for Databases Containing Queries. In: Proceedings of the Symposium on Principles of Database Systems (PODS), pp. 189–196 (1998)
33. Pan, Z., Heflin, J.: DLDB: Extending Relational Databases to Support Semantic Web Queries. In: Proceedings of the International Workshop on Practical and Scalable Semantic Systems (PSSS), (2003)
34. Presa, A., Velegrakis, Y., Rizzolo, F., Bykau, S.: Modelling Associations through Intensional Attributes. In: Proceedings of the International Conference on Conceptual Modeling (ER), (2009)
35. Ramakrishnan, R., Gehrke, J.: Database Management Systems. McGraw-Hill (2007)
36. Rose, R., Frew, J.: Lineage Retrieval for Scientific Data Processing: A Survey. ACM Computing Surveys **37**(1), 1–28 (2005)
37. Shadbolt, N., Berners-Lee, T., Hall, W.: The Semantic Web Revisited. IEEE Intelligent Systems **21**(3), 96–101 (2006)
38. Srivastava, D., Velegrakis, Y.: Intensional Associations between Data and Metadata. In: Proceedings of the ACM International Conference on Management of Data (SIGMOD), pp. 401–412 (2007)
39. Stonebraker, M., Anton, J., Hanson, E.N.: Extending a Database System with Procedures. ACM Transactions on Database Systems (TODS) **12**(3), 350–376 (1987)
40. Tatarinov, I., Viglas, S., Beyer, K.S., Shanmugasundaram, J., Shekita, E.J., Zhang, C.: Storing and querying ordered XML using a relational database system. In: Proceedings of the ACM International Conference on Management of Data (SIGMOD), pp. 204–215 (2002)
41. Velegrakis, Y., Miller, R.J., Mylopoulos, J.: Representing and Querying Data Transformations. In: Proceedings of the International Conference on Data Engineering (ICDE), pp. 81–92 (2005)
42. W3C: RDF vocabulary description language 1.0: RDF Schema (2004). Http://www.w3.org/TR/rdf-schema/
43. W3C: Resource Description Framework (RDF) (2004). Http://www.w3.org/TR/rdf-concepts/
44. Wang, R., Reddy, M.P., Kon, H.B.: Toward Quality Data: an Attribute-based Approach **13**(3-4), 349–372 (1995)
45. Weiss, C., Karras, P., Bernstein, A.: Hexastore: sextuple indexing for semantic web data management. Proceedings of VLDB Journal **1**(1), 1008–1019 (2008)
46. Widom, J.: Trio: A System for Integrated Management of Data, Accuracy, and Lineage. In: Proceedings of the Biennial Conference on Innovative Data Systems Research (CIDR), pp. 262–276 (2005)
47. Wyss, C.M., Robertson, E.L.: Relational languages for Metadata Integration. ACM Transactions on Database Systems (TODS) **30**(2), 624–660 (2005)
48. IBM DB2 XML Extender. Http://www4.ibm.com/software/data/db2/extenders/xmlext.html
49. Yoshikawa, M., Amagasa, T., Shimura, T., Uemura, S.: XRel: a path-based approach to storage and retrieval of XML documents using relational databases. ACM Transactions on Internet Technologies **1**(1), 110–141 (2001)