Benchmarking Database Representations of RDF/S Stores

Yannis Theoharis[1,2], Vassilis Christophides[1,2], and Grigoris Karvounarakis[3]

[1] Institute of Computer Science, FORTH, Vassilika Vouton, P.O.Box 1385, GR 71110
[2] Department of Computer Science, University of Crete, P.O.Box 2208, GR 71409, Heraklion, Greece
{theohari, christop}@ics.forth.gr
[3] Department of Computer and Information Science, University of Pennsylvania, 3330 Walnut St., Philadelphia, PA 19104, USA
gkarvoun@cis.upenn.edu

Abstract. In this paper we benchmark three popular database representations of RDF/S schemata and data: (a) a schema-aware (i.e., one table per RDF/S class or property) with explicit (ISA) or implicit (NOISA) storage of subsumption relationships, (b) a schema-oblivious (i.e., a single table with triples of the form ⟨subject-predicate-object⟩), using (ID) or not (URI) identifiers to represent resources and (c) a hybrid of the schema-aware and schema-oblivious representations (i.e., one table per RDF/S meta-class by distinguishing also the range type of properties). Furthermore, we benchmark two common approaches for evaluating taxonomic queries either on-the-fly (ISA, NOISA, Hybrid), or by precomputing the transitive closure of subsumption relationships (MatView, URI, ID). The main conclusion drawn from our experiments is that the evaluation of taxonomic queries is most efficient over RDF/S stores utilizing the Hybrid and MatView representations. Of the rest, schema-aware representations (ISA, NOISA) exhibit overall better performance than URI, which is superior to that of ID, which exhibits the overall worst performance.

1 Introduction

Several RDF stores have been developed during the last five years for supporting real-scale Semantic Web applications. They usually rely on (main-memory) virtual machine implementations or on (object-) relational database technology, while employing a variety of storage schemes. The most popular database representations for shredding RDF/S resource descriptions into relational tables are: the *schema-oblivious* (also called *generic* or *vertical*), the *schema-aware* (also called *specific* or *binary*) and a *hybrid* representation, combining features of the previous two. In *schema-oblivious*, a single table is used for storing both RDF/S schemata and resource descriptions under the form of triples (⟨subject-predicate-object⟩). In *schema-aware*, each property (or class) defined in an RDF/S schema is represented by a separate table. In hybrid one table per RDF/S meta-class is created, namely, for class and property instances with different range values (i.e., resource, string, integer, etc.). Several variations (e.g., with explicit or implicit database representation of subsumption relationships, use of resource URIs vs IDs, etc.) of these three core storage schemes have also been implemented in existing RDF stores [2,15,24,4,16,19,22,13] (see [20] for an extensive survey). In terms of inferring triples from schema information there exist two approaches: either to precompute them (at compile-time) or to compute them on demand (at run-time). The *schema-oblivious* (URI and ID), as well as, approaches using materialized views (MatView) adopt the former approach, while *schema-aware* (ISA and NOISA) and Hybrid adopt the latter. On demand computations can be performed either in main memory (as in ISA) or in secondary memory (as in NOISA and Hybrid). All these representations have pros and cons for different Semantic Web application scenarios, and, thus, benchmarking their performance is an important, but also a challenging task.

In this paper, we focus on the efficient evaluation of *taxonomic* RDF/S queries, retrieving the proper or transitive instances of a particular class or property. A key point affecting the performance of such queries is the representation of subsumption relationships and thus, the cost of traversing persistent class (or property) hierarchies. For this reason, we have developed a synthetic RDF/S generator, which takes as input the size of the subsumption hierarchies, the number of classified resources, as well as their distribution under classes or properties at various levels in the hierarchy and produces RDF/S schemas and resource descriptions that match these specifications. Then, we have conducted extensive experiments on the aforementioned RDF/S storage schemes on top of the object-relational DBMS PostgreSQL. The main conclusion drawn from these experiments is that the evaluation of taxonomic queries is most efficient over RDF/S stores utilizing the Hybrid and MatView representations. This result is especially interesting in the case of Hybrid which is also optimal in terms of storage space requirements, in contrast with MatView which relies on Transitive Closure (TC) precomputation over the database instances, that incurs a huge storage overhead. Of the rest, *schema-aware* representations (ISA, NOISA) exhibit as expected overall better performance than URI, which is superior to that of ID, that exhibits the worst performance.

Experimental results reported in [1] and [2] also highlight the performance gains of the *schema-aware* representation compared to the *schema-oblivious* one. The main reason is that in the former, tuples contain only the property values involved in a given query, while in the latter, tuples contain both property names and values and, thus, imply an additional filtering phase on the property name on a significantly larger table (i.e., extra overhead for schema filtering in all queries) to locate the tuples actually involved in a query. However, a comparative evaluation of taxonomic queries against different database representations of subsumption relationships is not provided in any of these studies. Furthermore, the statistical analysis presented in [14], [21] highlights the structural characteristics of RDF schemata employed by popular or emerging SW applications. However, these studies do not benchmark intensional (i.e., schema) or extensional (i.e., data) queries formulated against secondary memory-based RDF/S stores. Moreover, in [5] an extensive benchmarking of intensional taxonomic queries has been presented for various families of encodings, using real data from the Open Directory Portal as a testbed. In this paper, we take one step further, by evaluating both intensional and extensional taxonomic queries against various synthetic RDF/S schemata and resources descriptions, corresponding to different Semantic Web application needs. The goal of the experiments reported in [8] was to evaluate the trade-off between the materialization of the TC, including triples that are inferred by the schema, and its run-time computation using a DL (Description Logic) reasoner. Their main conclusion was that the use of materialized views in a database managed by a DL reasoner leads to increasing result completeness, while the query response time is considerably low. Furthermore, authors in [23] worked on the problem of incremental maintenance of materialized ontologies using logic reasoners by taking into account the RDF/S model semantics [11]. They also noted the trade-off between inferencing time, storage space and access time. Compared to these studies we provide precise formulas to estimate the storage overhead of the materialized approach.

The remainder of this paper is organized as follows: Section 2 surveys the main storage schemes adopted by existing RDF/S stores. Moreover, we illustrate the translation of taxonomic queries against each of the three possible RDF/S relational representations. Section 3 introduces our synthetic RDF/S generator based on different distribution modes of resources under the classes of a schema. Section 4 presents the results of our experimental evaluation using the qualitative and quantitative parameters considered by our RDF/S generator. Finally, Section 5 concludes our paper and discusses possible future directions in RDF/S benchmarking.

2 RDF/S Storage Schemes

The three widely used storage schemes for shredding RDF/S resource descriptions into relational tables are:

Schema-oblivious (also called *generic* or *vertical*): One ternary relation is used to store any RDF/S schema or resource description graph. This table contains triples of the form ⟨subject-predicate-object⟩ where attribute `subject` represents a resource that is the source of a property, whose name is given in attribute `predicate`, while attribute `object` represents a destination resource or a literal value for this property (see Figure 1). Different properties of a specific resource are tied together using the same subject URI.

Schema-aware (also called *specific* or *binary*): Unlike the previous representation one table per RDF/S schema property or class is used (see Figure 2).

Hybrid: In this representation (see Figure 3), there is a ternary relation for every different property range type and a binary relation for all class instances (as in *schema-aware*). On the other hand, property (class) instances with range values of the same type are stored in the same relation, distinguished by the property (class) id (as in *schema-oblivious*).

Triples

Subject (resource URI)	Predicate (property name)	Object (property value)

Fig. 1. Schema-oblivious representation

Property$_1$

Subject (resource URI)	Object (property value)

...

Property$_n$

Subject (resource URI)	Object (property value)

Class$_1$

Subject (resource URI)

...

Class$_m$

Subject (resource URI)

Fig. 2. Schema-aware representation

Properties with range Resource

Subject (resource URI)	Predicate (property name)	Object (property value)

...

Properties with range integer

Subject (resource URI)	Predicate (property name)	Object (property value)

Class Instances

Subject (resource URI)	Object (classid)

Fig. 3. Hybrid representation

Schema evolution is straightforward in the *schema-oblivious* approach, whereas the addition (deletion) of a new property requires the addition (deletion) of a table in the *schema-aware* approach. On the other hand, the former approach disregards type information, since all property values are usually stored as VARCHARs (i.e., strings) in the `object` attribute, whereas the latter entails a significant overhead when managing a potentially large number of tables (for voluminous RDF/S schemata). In Hybrid, schema evolution can be easily supported (as in *schema-oblivious*), while preserving type information (as in *schema-aware*).

The main variations of the *schema-aware* scheme concern the representation of subsumption relationships of classes and properties, defined in one or more RDF/S schemata. The first, called ISA, exploits the object-relational features of SQL99 [18] for representing subsumption relationships using sub-table definitions (see subsection 2.1). The second, called NOISA, ignores this feature and stores RDF/S data using a standard relational representation as depicted in Figure 2. Furthermore, two variations of the *schema-oblivious* scheme have been proposed, which differ in the way they store resources' URIs. The former, called URI, stores the URIs in the table holding the triples (usually repeating the same URI, e.g., in multiple triples that refer to properties of the same resource), while the latter, called ID, relies on integer identifiers to represent resources and properties in the triple table and stores them only once in a separate table (called "instance"). It should be stressed that the redundancy in the URI representation incurs a significant storage overhead. On the other hand, the ID representation suffers

Table 1. RDF/S storage schemes and systems

RDF/S Stores	**Schema-aware**			**Hybrid**	**Schema-oblivious**	
	ISA	NOISA	MatView	Hybrid	URI	ID
RDFSuite[2]	X	X		X		
Jena[15,24]					X[24]	X[15]
Sesame[4]	X					X
DLDB[16]			X			
RStar[13]						X
KAON[22]						X
PARKA[19]		X				
3Store[9]						X

from the need of an additional join operation at the end of every query, in order to retrieve the actual resource URIs.

Except from the triples which are explicitly defined in an RDF graph, many other can be inferred by the semantics of the schema (see RDF/S model semantics [11]). Two main approaches have been proposed to address this issue: the *a priori* (at compile-time) materialization in the persistent store or the *a posteriori* (at run-time) computation of the inferred triples. The former approach avoids to recomputing TCs for every query, but incurs a storage overhead and makes data updates harder, while the latter has less storage requirements, although its scalability is limited by the main memory space that is required for the run-time TC computations.

Existing RDF/S stores employing either URI or ID, the two *schema-oblivious* variations, usually adopt the former (materialized) approach,[1] while [16] proposes to store the precomputed triples as materialized views: for each class or property, a table holds its proper instances while a materialized view holds both proper and transitive instances. In order to create this view, the table with the proper instances is "unioned" with the views of its direct subclasses. Henceforth, we call this storage scheme MatView. On the other hand, RDF/S stores employing one of the two variations of the *schema-aware*, ISA and NOISA, as well as the Hybrid usually adopt the former (virtual) approach. It is worth noticing that both Hybrid and NOISA employ an internal encoding of subsumption relationships using interval-based labels of persistent classes and properties [5]. This encoding ensures an efficient evaluation of taxonomic queries in secondary storage, by transforming costly TC computations into appropriate range queries and reduces main memory requirements of the TC computation.

Table 1 summarizes the storage schemes implemented by existing RDF/S stores. Other approaches exist, but they are beyond the scope of our paper. For example, [6] focuses on how to derive an efficient *schema-aware* representation without any a priori knowledge of the employed RDF/S schemata. Although quite interesting, this work leads to more complex implementations of declarative query language interpreters running on the top of application-specific RDF/S

[1] Although SQL99 [18] defines a syntax for expressing transitive joins the existing implementations are not efficient [17].

stores. Furthermore, in [7], the authors employ the *schema-oblivious* approach for building persistent Semantic Web applications on top of existing RDF/S stores. Finally, native stores like Redland [3] or YARS [10] employ lower level database techniques to manage RDF/S data such as Hash Tables and B$^+$-trees, but do not provide full-fledged database functionality.

2.1 Translation of Taxonomic RDF/S Queries

In this subsection, we present the translation of the core RDF/S taxonomic queries into SQL over the relational schemas considered by the *schema-aware* (ISA and NOISA), the Hybrid and the *schema-oblivious* (URI and ID) representations illustrated in Figures 1, 2 and 3.

Consider, for instance, the (binary) tree-shaped class hierarchy of Figure 4. The label of each class is composed of two integers: the *end* number denotes the unique classid obtained when traversing the hierarchy in post-order, while the *start* is the *end* number of the leftmost descendant of the class. Then, to find all subclasses of the Root we simply need to issue a query with the filtering condition $1 \leq start, end \leq 7$ (i.e., returning the classes whose label is included in the interval of the Root). The labels of classes and properties, as well as their subsumption relationships are stored into two auxiliary tables called `SubClass` and `SubProperty`. In this context, each extensional taxonomic query issued against the NOISA and Hybrid representations (i.e., find all transitive instances of the Root) also implies an intensional query involving a range condition on class or property labels.

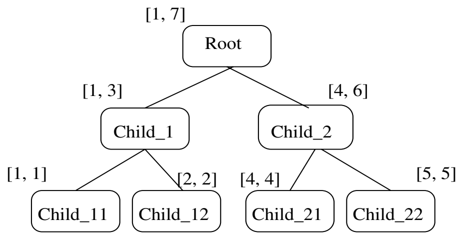

Fig. 4. Example of a labeled RDF/S schema

In the following we show the SQL translation of taxonomic queries at Root level (given its label) over our testbed representations:

- **Schema-aware** NOISA: the SQL translation of our example taxonomic query in this representation is performed in two phases. First we need to find all the subclasses of the Root class:

```
select  S.end
from    SubClass S
where   S.start >= RootStart and S.end <= RootEnd
```

  Next, we need to scan sequentially all the tables holding the instances of the previously retrieved subclasses, in the order determined by the query plan:

```
(select URI from Child_11) UNION ALL (select URI from Child_12) UNION ALL
(select URI from Child_1)  UNION ALL (select URI from Child_21) UNION ALL
(select URI from Child_22) UNION ALL (select URI from Child_2)  UNION ALL
(select URI from Root)
```

Note that, internally, the unique classid (i.e., the post order number) is used as table name (rather than a string as depicted in Figure 2) and no elimination of duplicates is required (`UNION ALL`) to assemble resource URIs.

– **Schema-aware** ISA: the SQL translation of our example taxonomic query in this representation is left entirely to the internal implementation of the PostgreSQL table inheritance feature:

```
select URI from Root;
```

where all the tables of the involved subclasses are sequentially scanned. As a matter of fact, PostgreSQL relies on a special catalog table, called *pg_inherits*, to store the subsumption relationships of tables defined in an object-relational schema. This table holds a unique id for each sub-table, along with the id of its parent table and the number of table occurrences in the hierarchy (in case of multiple inheritance). Then, a C program uses this information to compute the sub-tables involved in an SQL query: first, the tableID is inserted into an empty list; then, the direct children of this table are found, by performing a selection on *pg_inherits*, and their tableIDs are appended to the list. This process is repeated recursively for each new tableID that is appended to the list, until a fixpoint is reached. After that, PostgreSQL scans all tables in the order in which they appear in the list.

– **Hybrid**: the SQL translation of our example taxonomic query in this representation is simpler, since it requires only one phase for both schema filtering and instance scanning:

```
select  I.URI
from    ClassInstances I
where   I.classid >= RootStart and I.classid <= RootEnd;
```

– **Schema-oblivious** URI: the SQL translation of our example taxonomic query in this representation is:

```
select  T.SubjectURI
from    Triples T
where   T.predicate = 'typeof' and T.object = 'Root';
```

– **Schema-oblivious** ID: the SQL translation of our example taxonomic query includes a join operation between the table holding triples and the one holding resources' URIs. Below, *typeofID* stands for the identifier of the property *typeOf*:

```
select  I.URI
from    Triples T, Instance I
where   T.predicate = typeofID and T.ObjectID = RootID and I.ID = T.SubjectID;
```

– MatView**:** the SQL translation of our example taxonomic query involves a simple scan on the materialized view which stores the proper and transitive instances of the Root class.

```
select  MV.URI
from    Mat_View_Root MV;
```

3 Synthetic RDF Data Generation

As we will explain in the sequel, for relatively small schema sizes, *the hierarchy structure (i.e., shape and arity) does not affect the performance of (intensional or extensional) taxonomic queries*; as a matter of fact, *their performance only depends on the number of nodes in the hierarchy.* For this reason, our RDF/S generator produces only binary-tree-shaped subsumption hierarchies rather than more exotic structures of class or property lattices.

More precisely, the three critical parameters of our generator are (a) the depth of the tree; (b) the total number of classified resources; and (c) their distribution mode under nodes at various hierarchy levels. It should be stressed that the tree depth determines the size of an RDFS schema and therefore the number of tables we have to create in the two *schema-aware* representations (i.e., for complete binary trees $2^{depth+1} - 1$ tables). In our benchmark we consider three categories of schemata, namely, *small* (up to 4 levels, i.e., 31 nodes), *medium* (up to 6 levels, i.e., 127 nodes), and *large* (more than 7 levels). In addition, the number of resources that we consider in our experiments is 10,000, 100,000 and 1,000,000.

3.1 Distribution of Resources

In average case analysis, we can consider a uniform distribution of resources under the nodes of the tree-shaped subsumption hierarchy. However, this is not a realistic assumption for real-life Semantic Web applications. For instance, in some SW applications, such as Semantic Web Portals [5], leaf classes are highly populated compared to the intermediate ones while in other applications such as Knowledge Bases (e.g., the IMDB[2] wrapped in RDF/S) some class subtrees are heavier than others in terms of classified resources. Our RDF/S generator relies on the zipfian distribution [25] to simulate the classification of resources in Semantic Web applications.

Definition 1. *The distribution of occurrence probabilities of resources under the schema classes follows the zipfian law:*

$$Zipf(A, i) = A/(i^z * h)$$

where A is the total number of resources to be distributed, i is the rank value given to each class, N is the total number of classes, z is a skew parameter and $h = \sum_{j=1}^{N} 1/j^z$.

After assigning an increasing rank to each class, the probability that a resource is classified under a class according to the zipfian distribution essentially follows a power-law: the number of resources classified under the class with the 1st rank is i^z times larger than the class with the ith rank. When $z = 0$, resources are uniformly classified, while when $z > 0$ some classes are more frequently populated than others. In this work we consider that $z = 1$ and thus, a class with i-th rank, can be populated with $A/(i * h)$ resources. In a nutshell, our generator considers the following resource distribution modes:

[2] Url: www.imdb.com

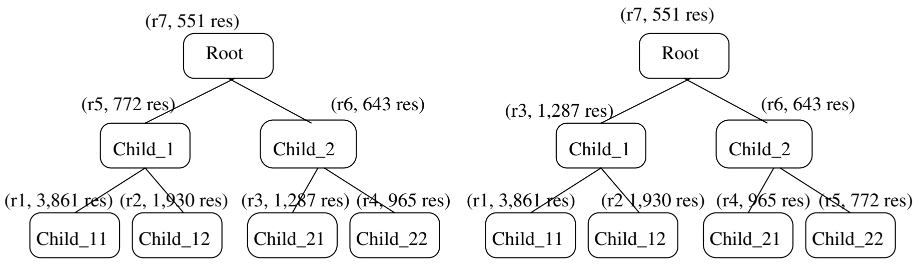

Fig. 5. Zipfian Distribution favouring leaves vs favouring subtrees

- *Uniform distribution of resources to tree classes ($z = 0$)*: in this case, resource distribution is determined only by the total number of schema classes (i.e., the tree depth). For instance, with a uniform classification of 10,000 resources under the seven nodes of our example schema depicted in Figure 4, in the case of ISA and NOISA we need to insert 10,000/7 = 1428 tuples in each class instance table, while in Hybrid 10,000 tuples will be inserted into the single class instance table (Hybrid), 1/7 of which will have the classid of the Root class as the value of the attribute `object`, 1/7 of which will have the classid of the Child_1, etc.
- *Zipfian distribution of resources favouring tree leaves ($z = 1$)*: in this case, lower rank values are given to leaf classes while the Root class has the highest rank. Using this class ranking, the classification of 10,000 resources under our example schema is illustrated in the left part of Figure 5 (for each class its rank value and number of classified resources is shown).
- *Zipfian distribution of resources favouring sub-trees ($z = 1$)*: in this case, lower rank values are given to the classes of a sub-tree. The generator is parameterized to take into account the depth of the root class of a sub-tree. For instance, the lower rank values are given to the classes of the first (leftmost) sub-tree whose root (Child_1) is located at depth 1 of our example schema. Using this class ranking, the classification of 10,000 resources under our example schema is illustrated in the right part of Figure 5 (for each class its rank value and number of classified resources is shown).

4 Experimental Evaluation

In this section, we present a performance evaluation of taxonomic queries issued against six relational representations (ISA, NOISA, MatView, Hybrid, URI and ID), using the synthetic RDF/S schemata and data created by our generator. The objective of our study is to measure the effect of the schema size in intensional taxonomic queries, as well as, the effect of resource number and distribution modes in extensional taxonomic queries. Experiments were carried out on a pc with a Pentium III 1GHz processor and 256MB of main memory, over Suse Linux (v9.2) using PostgreSQL (v7.4.6) with Unicode configuration and 10,000 buffers (8KB each), used for data loading, index creation and querying. Each query was run several times: once, initially, to warm up the database buffers and then nine more times to get the average execution time of a query.

4.1 Physical Database Schema and Size

First, we loaded tree-shaped schemata of variable depth into the database. In ISA and NOISA, tables **SubClass** (or **SubProperty**) were populated with the subsumption relationships of the synthetically generated RDFS schema In these representations, an index on the `uri` attribute was created for each instance table of a specific schema class. To speed-up sequential access, each instance table was clustered according to this index. In Hybrid, a B^+-tree index was created on the `classid` attribute of the single table that holds the instances of all classes. This table was clustered according to `classid`, in order to minimize the I/Os required when fetching the resources that are classified under a specific class.

Then, we generated various datasets according to the distribution modes presented in the previous section and load them into the instance tables of each representation. To compute the physical database size for each representation we consider that the attribute `uri` has the type `VARCHAR(1000)`, while `classid` in Hybrid has the type `int4`. Moreover, we took into account the extra storage cost per tuple due to an internal id of 40 bytes generated by PostgreSQL to identify the physical location of a tuple within its table (block number, tuple index within block). PostgreSQL also incurs an overhead of 4 bytes for the storage of strings. In *schema-oblivious*, the attribute `predicate` has the type `VARCHAR(20)`. Table 2 summarizes the size of the database for the three different numbers of resources, distributed uniformly among the schema classes.

- ISA and NOISA: For each tuple $((1000*1+4)+40)$ Bytes $= 1$KB are needed.
- Hybrid: For each tuple $((1000*1+4)+4+40)$ Bytes $= 1$KB are needed. Also for each entry of the index constructed on `classid`, PostgresSQL holds 8 bytes for the 'row pointer' and 4 bytes for the `int4` type of the search key. Since 12 Bytes are needed per index entry, the expected index size for 10,000 resources is around 12KB. However, PostgreSQL fills, as expected, each index page until the fill-factor of 70%. As a result, for 10,000 resources the index size is approximately $1.3*12$KB $= 15.6$KB.
- Schema-oblivious: For each tuple $(2*(1000*1+4)+(20*1+4))$ Bytes $= 2$KB are needed.

The following lemma gives a precise measure of the storage overhead of TC precomputations, in *schema-oblivious* and MatView.

Lemma 1. *Consider a complete-binary tree shaped RDFS schema and uniform resource distribution. Let d be the depth of the tree and A be the number of triples explicitly given. Then the number of total triples (those explicitly given and those inferred due to class or property subsumption) is: $totalTriples(A,d) \simeq d*A$*

Proof. *Let A be the total number of triples. Then, each class has $y = A/2^{d+1}-1$ triples. Computing inferred triples for each class in a bottom-up fashion results in the following total number of triples:*

Table 2. Database size

# of Resources	ISA,NOISA,Hybrid	URI	ID	MatView
10,000	10 MB	$depth * 20$ MB	$\simeq$ 10-14 MB	$depth * 10$ MB
100,000	100 MB	$depth * 200$ MB	$\simeq$ 100-140 MB	$depth * 100$ MB
1,000,000	1 GB	$depth * 2$ GB	$\simeq$ 1-1.4 GB	$depth * 1$ GB

$TA = \sum_{i=0}^{d} 2^i * (2^{d+1-i} - 1) * y = y * (\sum_{i=0}^{d} 2^{d+1} - 2^i) = y * ((d+1) * 2^{d+1} - \sum_{i=0}^{d} 2^i) = y*((d+1)*2^{d+1}-(2^{d+1}-1)) = A*((d+1)*2^{d+1}/(2^{d+1}-1)-1) \simeq d*A$ $\square$

Lemma 1 presumes a complete, binary-tree-shaped schema. It should be also stressed that, a zipfian distribution of resources favouring leaves or subtrees implies that a larger number of resources will be located deeper in the tree. Since MatView, URI and ID duplicate the resources classified under a class in the instance tables of all of its superclasses, the storage overhead in these representations is more significant in the case of the zipfian than in the case of the uniform distribution.

Increasing the number of triples during TC precomputation implies a direct increase of the database size. URI's storage requirements are obviously d times larger than without precomputed TCs. On the other hand, ID hold triples of the type ⟨int4, int4, int4⟩ and resource URIs are only stored once. Hence, the storage overhead in ID is significantly smaller than in URI. More precisely, each triple needs $3 * 4 + 40 = 52$ Bytes (vs 2KB in URI).

Finally, in MatView, each view is of type ⟨resourceURI, id⟩. The aforementioned storage overhead of this representation can be computed in a similar way by changing the meaning of A, from "total number of triples" to "total number of resources".

4.2 The Effect of Schema Size

As we have already explained in Section 2.1, taxonomic queries involve two filtering phases, an intensional (i.e., at the schema) and an extensional (i.e., at the data). During the former, we need to compute all the subclasses of the root class whose transitive instances need to be retrieved. Recall that, in NOISA and Hybrid, this computation is performed by a range query on the classes' interval-based labels, while in ISA a TC is performed internally on the structural information of the inheritance table catalog (*pg_inherits*) maintained by PostgreSQL. During the extensional filtering phase of a taxonomic query, *schema-aware* (both ISA and NOISA) needs to scan a number of (possibly empty) tables, containing the instances of the schema classes, while all the other representations need to scan only one (possibly empty) table, regardless of the number of the schema classes under which resources are classified.

In order to measure the cost of the schema filtering in taxonomic queries, we have executed the same query (i.e., transitive instances of the Root class) against an empty database created according to the six possible representations. As we can see from Figure 6, in *schema-aware* the execution *time of taxonomic queries depends on the size of the schema in terms of number of classes or properties* (and thus on the depth of our complete binary tree) while the execution *time in the other storage schemes is independent from the schema size* (almost 0 seconds). The extra cost of the *schema-aware* is due to the I/O seek time of empty tables.

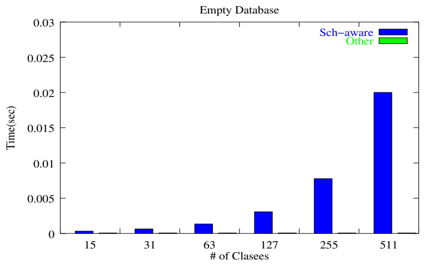

Fig. 6. Querying an empty database

Moreover, since the physical size of a resource's URI in ISA and NOISA is 1KB and each PostgreSQL buffer requires 8 KB (out of which only 8,152 Bytes of them are used - the other 40 Bytes hold block information), only 7 tuples (i.e., 7 resources) can be stored in one block. Thus, the last block of each table may contain from 1 up to 7 resources. This factor incurs an extra storage overhead, which in the worst case (i.e., 511 classes for *depth* = 8) can be up to 4MB.

4.3 The Effect of Resource Distribution Mode

Recall that, taxonomic queries in Hybrid are evaluated in one phase, where both schema and data filtering are performed against the single table used to store all class instances. Then, to find the transitive instances of a specific class (e.g., Root) we only need to perform an index scan on the `classid` (filtering condition on the labels of the descendant classes) and table clustering on this attribute is high beneficial for query performance. Hence, as we can see in Figure 7, the execution *time in* Hybrid *scales linearly with respect to the size of the database (i.e., the number of classified resources)*. Similar behaviour is exhibited by *schema-oblivious* and MatView, also evaluating taxonomic queries in one phase due to the TCs precomputation.

Before further detailing our experimental results, we would like to point out that ISA and NOISA exhibit the same behavior in terms of all the aforementioned factors affecting the evaluation of taxonomic queries. They only differ in the fact that schema filtering (as the first evaluation phase of taxonomic queries in *schema-aware*) in ISA is performed in main-memory by PostgreSQL, while in NOISA it is handled by a separate query. However, for small schema sizes the main-memory and the persistent processing of the schema filtering phase comes with almost the same execution cost. For this reason, both representations gave the same measurements in all experiments and thus we are going to refer to both of them in the figures below as *schema-aware*.

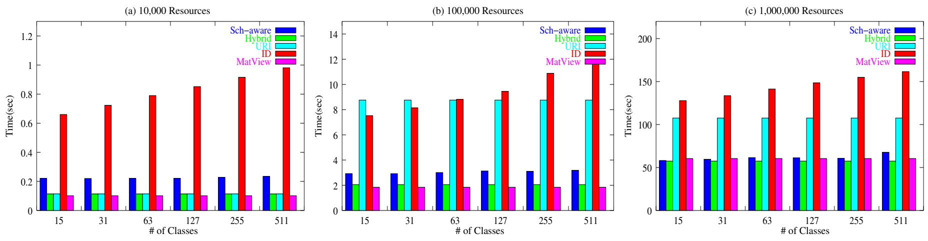

Fig. 7. Root Queries: variable depth and number of uniformly distributed resources

Querying the Root Class: Figure 7 depicts the execution times for a query requesting the transitive instances of the **root** class, in the case of the uniform distribution, over each representation. The performance figures for the two zipfian distributions were very similar to Figure 7, thus we conclude that *the distribution mode doesn't affect execution time of taxonomic queries at the root class.*

It is worth noticing that the schema size affects query evaluation time only in the case of *schema-aware*, due to the storage overhead explained previously. As we can see in Figure 7(a), this overhead, which varies between 0 and 4MB, has a significant effect for 10,000 resources (i.e., of size 10 MB), while Figures 7(b),(c) depict that it is not as important for larger numbers of resources, where the *schema-aware* representations achieve similar performance to that of Hybrid and MatView. Furthermore, comparing Hybrid and URI, we can easily observe that URI exhibits very similar performance to Hybrid in the case of 10,000 resources (Figure 7(a)), while Hybrid clearly outperforms URI in the other two cases (Figures 7(b),(c)). The reason for the latter is that the physical size of triples involved in the query in URI is twice as large as the size of the tuples involved in the **ClassInstances** table of Hybrid, thus additional I/O activity is required for this representation. Regarding ID, in all figures it exhibits the worst performance, because it requires an extra join to retrieve the actual resource URI. This join is very costly, given that the number of triples involved is, on average, *depth* times larger than the actual triples existing in the RDF graph (Lemma 1). Finally, MatView is the only representation, between those who precompute TCs, which achieves good performance, since taxonomic query evaluation only involves a sequential scan over the corresponding view. However, precomputing the TCs (also for URI and ID) both incurs a huge storage overhead and also creates the need for a view-update strategy.

Querying a Middle Level Class: Figure 8 depicts the execution times over each representation, for a query requesting the transitive instances of a **middle** level class, in the case of a zipfian distribution favouring subtrees (note that the y axis is drawn in logarithmic scale). Clearly, for a *small* and *medium* number of resources, Hybrid and MatView exhibit the overall best performance, while for a *large* number of resources *schema-aware* and MatView outperform all other representations. Of the rest ID performs better than URI, but they are both far worse than the previous three representations.

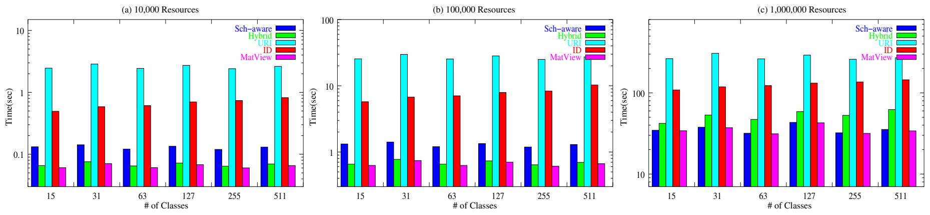

Fig. 8. Middle level queries: Zipfian distribution favouring subtrees

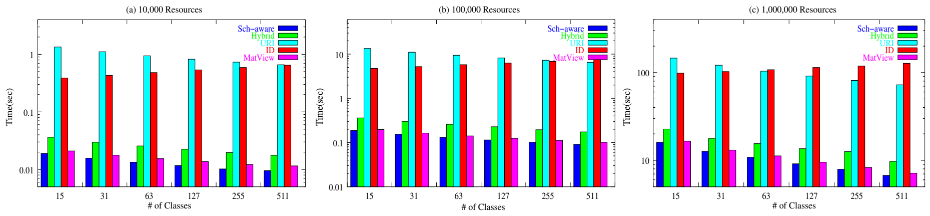

Fig. 9. Leaf level queries: Zipfian distribution favouring leaves

In the case of middle level queries we have to access a smaller number of subclasses, as well as of classified resources, than in the case of querying the root class. In order to measure the effect of distribution modes in query evaluation we need to compute the selectivity of the filtering conditions on the instance tables of each representation. A zipfian distribution favouring subtrees leads to query selectivity of 35% and 45%, while favouring leaves leads to selectivity between 45% and 55%[3]. We should point out that, since in the two zipfians distributions (favouring subtrees or leaves) the subtrees rooted at a **middle** level have different weights (i.e., number of classified resources) we choose in our experiments to query the heaviest subtrees.

The varying selectivity rapidly affects query evaluation time in Hybrid and URI. In the case of Hybrid, an index scan is performed on the **ClassesInstances** table, using the B$^+$-tree index on the attribute `classid` where the selectivity is fairly high, as opposed to the sequential scan required to retrieve the instances of the root class. As one would expect, the higher the selectivity, the higher is the benefit of choosing an index scan. On the other hand, when the selectivity is low, the I/O cost of accessing and using the index may be greater than the benefit; hence, index scan is efficient in the case of a uniform distribution, while it incurs an execution overhead in the case of the two zipfian distributions. This behaviour is reflected in Figure 8(c), where Hybrid exhibits worse performance than *schema-aware* and MatView for a *large* number of resources. Also the index scan on table **Triples** in URI uses a B$^+$-tree index on the `object` attribute. The overhead of accessing the index in this case is even bigger than in Hybrid, since the index size in URI (index on a `VARCHAR(1000)` attribute) is much bigger than

[3] On the other hand, a uniform distribution leads to increased selectivity, starting from 80% for *depth* = 3 and increasing up to 94% for *depth* = 8.

in Hybrid (index on an `int4` attribute). As a result, URI exhibits the overall worst performance. Finally, in the case of ID the query plan produced by PostgreSQL do not use the index on `object`, but a sequential scan on table **Triples**. Hence, what really affects the evaluation of taxonomic queries in ID is not the distribution mode, but the depth of the subsumption hierarchy, since the total number of triples is depth times larger (Lemma 1) than the original one.

Querying Leaves. In this case we have to access only a single class and a smaller number of classified resources. The former implies no additional I/Os for the *schema-aware* representations due to space left at the end of blocks. Hence, *schema-aware* exhibits the same (overall best) behaviour as MatView (Figure 9, note that the y axis is drawn in logarithmic scale). Furthermore, the selectivity is higher than in the case of queries at **middle** level, and ranges between 70% and 85% in the two zipfian distributions.[4] (we queried the **leaf** class with the largest weight). As a result, the perfomance figures of Hybrid converge with those of *schema-aware* and MatView, while URI and ID follow by far.

Due to space limitations, we are not showing the experimental results for the cases of querying middle level and leaf classes, when resources are distributed uniformly. However, the results in those cases illustrate the same trends, with the exception that URI performs better than ID.

5 Summary and Future Work

The main conclusion that can be drawn from our experiments is that Hybrid and MatView achieve the best performance in terms of query execution times of taxonomic queries. Both exhibit very similar performance in the cases of *small*, *medium* and *large* numbers of resources (namely 10,000, 100,000 and 1,000,000, respectively) and queries on the **root** or **leaf** classes, while MatView outperforms Hybrid in the case of queries on **middle** level classes.

However, the performance of MatView relies on the duplication of resources in the instance tables of all superclasses of the class under which they are classified, which incurs a huge storage overhead. Moreover, MatView comes with the additional cost of data updates in materialized views, which can be a decisive factor in applications involving frequent updates (URI and ID also suffer from the same drawbacks). Unlike MatView, Hybrid achieves competitive performance without having to precompute TCs, by taking advantage of the encoding of subsumption hierarchies (the attribute `classid`) that is stored with the data (resource `uri`), enabling to evaluate taxonomic queries in a single phase.

Of the rest, *schema-aware* representations achieve similar performance to Hybrid and MatView for *medium* and *large* number of resources and queries on **root** class. Additionally, *schema-aware* exhibit the overall best performance in the case of taxonomic queries at **leaf** level classes. Furthermore, *schema-aware* is better than URI for *medium* and *large* number of resources and queries on root,

[4] Compared to selectivity ranging between 94% and 99.8% in a uniform distribution.

and clearly for queries at **middle** or **leaf** level classes. Note that, URI is sensitive to the size of the main-memory used for caching: as this size increases, URI's performance improves for larger number of resources. It is worth noticing that queries in our benchmark were executed against databases that only contained resources classified under classes. The addition of property-related triples in a single **Triples** table, in the case of *schema-oblivious* representations (URI, ID) would further degrade the performance of the two *schema-oblivious* representations.

Finally, apart from the case of taxonomic queries at **middle** or **leaf** level classes and zipfian resource distribution where ID outperforms URI, ID exhibits the worst performance, mainly because of the costly join operation it has to perform, and also suffers from the same drawbacks as MatView and URI (although the storage overhead is much smaller than in the case of URI).

It should be stressed that the conclusions drawn from our experiments are also confirmed by the independently conducted benchmarking of XML database implementations [12], where the combination of *document-dependent* partitioning (as in schema-aware) with the use of interval-based encoding for containment joins (similar to taxonomic queries) yields superior performance, compared to document-indepedent (similar to schema-oblivious) approaches. As a next step, we plan to extend our testbed to other categories of queries involving schema and data path expressions, that are translated to SQL queries with joins over the underlying RDBMS. To that end, we need to extend our generator with appropriate distribution modes of properties over (domain or range) classes. Conclusions of [21] could offer a basis for our attempt to model more sophisticated schema structures.

References

1. R. Agrawal, A. Somani, and Y. Xu: Storage and Querying of E-Commerce Data. In Proc. of VLDB 2001.
2. S. Alexaki, V. Christophides, G. Karvounarakis, D. Plexousakis: On Storing Voluminous RDF Descriptions: The case of Web Portal Catalogs. In Proc. of WebDB'01 (co-located with ACM SIGMOD'01).
3. D. Beckett: Redland RDF Application Framework, 2003.
4. J. Broekstra, A. Kampman and F. van Harmelen: Sesame: A generic Architecture for Storing and Querying RDF and RDF Schema. In Proc. of the ISWC'02.
5. V. Christophides, M. Scholl, D. Plexousakis., S. Tourtounis: On Labelling Schemes for the Semantic Web. In Proc. of the 12th Intern. World Wide Web Conference (WWW'03), 2003.
6. L. Ding, K. Wilkinson, C. Sayers, H. Kuno: Application-Specific Schema Design for Storing Large RDF Datasets. In Proc. of the PSSS'03, collocated with ISWC'03.
7. M. Gertz, K.-U. Sattler: A Model and Architecture for Conceptualized Data Annotations. Technical Report CSE-2001-11, Dept. of Computer Science, University of California, Davis, 2001.
8. Y. Guo, J. Heflin, Z. Pan: Benchmarking DAML+OIL Repositories. In Proc. of ISWC'03.

9. S. Harris, and N. Gibbins: 3Store: Efficient Bulk RDF Storage. In Proc. of 1st International Workshop on Practical and Scalable Semantic Web Systems 2003.
10. A. Harth, S. Decker: Yet Another RDF Store: Perfect Index Structures for Storing Semantic Web Data With Contexts. DERI Technical Report, 2004.
11. P. Hayes: RDF Semantics. W3C Working Draft, World-Wide Web Consortium (W3C), 2003.
12. H. Lu, J. X. Yu, G. Wang, S. Zheng, H. Jiang, G. Yu, A. Zhou: "What Makes the Differences: Benchmarking XML Database Implementations", ACM TOIT, Vol.5, No.1, Feb'05, p 154–194.
13. L. Ma, Z. Su, Y. Pan, L. Zhang, T. Liu: RStar: An RDF Storage and Query System for Enterprise Resource Management. In Proc. of the ACM CIKM 2004.
14. A. Magkanaraki et al: Benchmarking RDF schemata for the Semantic Web. In Proc. of the 1st International Semantic Web Conference (ISWC'02), 2002.
15. B. McBride. Jena: Implementing the RDF Model and Syntax Specification. 2001, Technical report Hewlett Packard Laboratories.
16. Z. Pan, J. Heflin: DLDB: Extending Relational Databases to Support Semantic Web Queries. In Proc. of PSSS'03, collocated with ISWC'03.
17. G. Schadow, M. Barnes, and C. McDonald, Representing and querying conceptual graphs with relational database management systems is possible, In Proc. of AMIA Symposium 2001:598-602
18. SQL99 Standard, NCITS/ISO/IEC 9075-1 01-Jan-1999 Information Technology - Database Languages - SQL - Part 1: Framework.
19. K. Stoffel, M. Taylor, J. Hendler: Efficient Management of Very Large Ontologies. In Proc. of American Association for Artificial Intelligence Conference (AAAI'97), 1997.
20. SWAD-Europe Deliverable 10.2: Mapping Semantic Web Data with RDBMSs.
21. C. Tempich, R. Volz: Towards a benchmark for Semantic Web reasoners - an analysis of the DAML ontology library. In Proc. of The 2nd Int. Workshop on Evaluation of Ontology-based Tools, EON2003.
22. R. Volz, D. Oberle, B. Motik, S. Staab: KAON SERVER - A Semantic Web Management System. In Proc. of the Atlantic Web Intelligence Conference (AWIC'03), 2003.
23. R. Volz, S. Staab, B. Motik: Incremental Maintenance of Materialized Ontologies. Proc. of ODBase'03, 2003.
24. K. Wilkinson, C. Sayers, H. A. Kuno, D. Raynolds: Efficient RDF Storage and Retrieval in Jena2. In Proc. of SWDB'03 (co-located with VLDB'03).
25. G. K. Zipf: Human Behaviour and the Principle of Least Effort. Addison-Wesley, Reading, Massachusetts, 1949.