Towards Performance Evaluation of Semantic Databases Management Systems

Bery Mbaiossoum[1,2], Ladjel Bellatreche[1], and Stéphane Jean[1]

[1] LIAS/ISAE-ENSMA - University of Poitiers 86960, Futuroscope Cedex, France
{mbaiossb, bellatreche, jean}@ensma.fr
[2] University de N'Djamena, Chad Republic

Abstract. The spectacular use of ontologies generates a big amount of semantic instances. To facilitate their management, a new type of databases, called semantic databases ($\mathcal{SDB}$) is launched. Large panoply of these $\mathcal{SDB}$ exists. Three main characteristics may be used to differentiate them: (i) the storage layouts for storing instances and the ontology, (ii) ontology modeling languages, and (iii) the architecture of the target database management system (DBMS) supporting them. During the deployment phase, the database administrator (DBA) is faced to a choice problem (which $\mathcal{SDB}$ she/he needs to choose). In this paper, we first present in details the causes of this diversity. Based on this analysis, a generic formalization of $\mathcal{SDB}$ is given. To facilitate the task of the DBA, mathematical cost models are presented to evaluate the performance of each type of $\mathcal{SDB}$. Finally, two types of intensive experiments are conducted by considering six $\mathcal{SDB}$, both issued from industry and academic communities; one based on our mathematical cost models and another based on the studied semantic DBMS cost models.

Keywords: Semantic Databases, Cost Models, Query Performance

1 Introduction

The database is one of the robust technologies. Along its history, we remark that when a new data model (e.g., the relational, object or XML data model) is considered; the database technology offers *storage*, *querying* and *management* solutions to deal with these new data. These solutions were directly implemented in academic and industrial DBMS. In the last decades, the semantic technology got a lot of attention from the research community, where semantic (ontological) data models were proposed. This phenomenon generates a large amount of semantic data that require efficient solutions to store and manage them. In order to honour its tradition and to keep its marketplace, the database technology responded to this need and offered *persistent solutions*. As a consequence, a new type of databases is created, called *semantic databases* ($\mathcal{SDB}$), to store both data and ontologies in the same repository. A large panoply of $\mathcal{SDB}$ exists. OntoDB [1], Jena [2] or Sesame [3] are examples of academician $\mathcal{SDB}$. Oracle [4] and IBM [5, 6] are example of solutions coming from industry.

Contrary to traditional databases, $\mathcal{SDB}$ bring new dimensions: (1) *the diversity of ontology formalisms*: each $\mathcal{SDB}$ uses a particular formalism to define its ontologies (e.g., OWL [7] or PLIB [8]), (2) *the diversity of storage layouts*: in a $\mathcal{SDB}$, several storage layouts (horizontal, vertical, binary) are used to store ontologies and their data, and (3) *the diversity of architectures*: three main architectures of DBMS managing $\mathcal{SDB}$ are distinguished according to the number of schemes used to store ontologies, data and eventually the ontology formalism used to define them. These dimensions complicates the deployment phase of $\mathcal{SDB}$ as each dimension may impact positively or negatively the performance of the target applications. Thus evaluating different $\mathcal{SDB}$ becomes a crucial issue for DBA. To do so, two directions are possible: (i) the use of mathematical cost models and (ii) the real deployment of $\mathcal{SDB}$ on several DBMS. Most of studies have been concentrated on the physical phase of performance of $\mathcal{SDB}$, where algorithms for selecting optimization structures such as materialized views [9] are proposed. These studies assume that a fixed $\mathcal{SDB}$ is used and do not consider the dimensions that we discussed. Thus the proposed cost models do not cover all possible combinations of deployment, they only consider one $\mathcal{SDB}$ with a fixed ontology formalism model, storage layout and architecture.

In this paper, we propose a formalization of the notion of $\mathcal{SDB}$. The different components of this formal model are illustrated. This formalization allows us to compare the existing $\mathcal{SDB}$. Cost models depending on the different types of $\mathcal{SDB}$ are proposed and used to compare the existing $\mathcal{SDB}$. To the best of our knowledge, our proposal is the first one dealing with the development of cost models considering the large diversity of $\mathcal{SDB}$. To validate this cost model, several experiments are run on the LUBM benchmark.

This paper consists of six sections. Section 2 defines basic notions related to ontologies and presents the diversity of $\mathcal{SDB}$. Section 3 introduces a formalization of $\mathcal{SDB}$ and the section 4 defines our cost model. We present in section 5 the performance evaluations of several $\mathcal{SDB}$. Finally, section 6 concludes the paper.

2 Background: the diversity of $\mathcal{SDB}$

A variety of $\mathcal{SDB}$ have been proposed in the last decade. Some $\mathcal{SDB}$ only consider the management of ontology instances represented as RDF data while others also support the management of ontologies and ontology formalisms inside the database. Moreover the management of these data is based on different storage layouts. In this section, we introduce basic notions about ontology and detail the different storage layouts and architectures of $\mathcal{SDB}$.

2.1 Ontology definition and formalism

An ontology is a consensual model defined to explicit the semantics of a domain by a set of concepts (class or property) that can be referenced by universal identifiers (e.g., URI). Two main types of concepts in a conceptual ontology are distinguished: *primitive* and *defined concepts*. *Primitive concepts* (or canonical

concepts) represent concepts that can not be defined by a complete axiomatic definition. They define the border of the domain conceptualized by an ontology. *Defined concepts* (or non canonical concepts) are defined by a complete axiomatic definition expressed in terms of other concepts (either primitive or defined concepts). These concepts are the basis of inference mechanisms like automatic classification.

Several formalisms (or languages) have been proposed for defining ontologies. They differ on their descriptive and deductive capabilities. Some ontology languages focuses mainly on the definition of primitive concepts. Thus they are used to define ontologies that are consensual and enhanced conceptual models. RDF Schema and PLIB are two examples of such languages. RDF Schema is a language defined for the Semantic Web. It extends the RDF model to support the definition of classes and properties. The PLIB (Parts Library) formalism is specialized in the definition of ontology for the engineering domain which often requires a precise description (e.g., value scaling and context explication) of primitive concepts. OWL Lite, DL and Full are ontology models with more deductive capabilities. They support the definition of defined concepts with various constructors such as restrictions (e.g, the definition of the *Man* class as all persons whose gender is male) or Boolean expression (e.g., the definition of the *Human* class as the union of the *Man* and *Woman* classes).

2.2 Storage Layouts used in $\mathcal{SDB}$

Three main storage layouts are used for representing ontologies in databases [10]: vertical, binary and horizontal. These storage layouts are detailed below and illustrated on a subset of the LUBM ontology in the Figure 1.

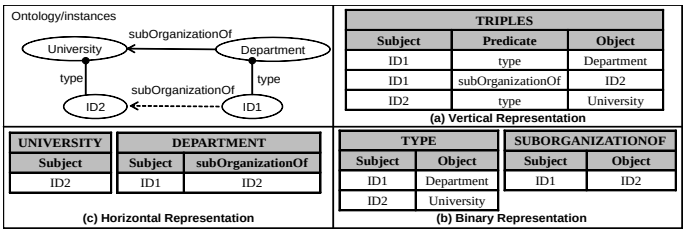

Fig. 1. Main storage layouts used by $\mathcal{SDB}$

– *vertical storage layout:* it consists of a single triples table with three columns `(subject, predicate, object)`. Since URI are long strings, additional tables may be used to store only integer identifier in the triple table. Thus this storage layout is a direct translation of the RDF model. It can be used to store the different abstraction layers related to the management of ontologies. The main drawback of this storage layout is that most queries require

a lot of self-join operations on the triple table [10].The $\mathcal{SDB}$ Sesame (one version of it) and Oracle use this storage layout.

- *binary storage layout:* it consists of decomposing the triple table into a set of 2-columns tables `(subject, object)`, one for each predicate. In some implementations, the inheritance of classes and class membership are represented in a different way (e.g., the class membership can be represented by a unary table for each class or the inheritance using the table inheritance of PostgreSQL). Compared to the vertical storage layout, this storage layout results in smaller tables but queries can still require many joins for queries involving many properties [1]. The $\mathcal{SDB}$ SOR uses the binary storage layout for the representation of ontologies and their instances.
- *horizontal storage layout:* it consists of a set of usual relational tables. For storing ontologies, this storage layout consists of a relational schema defined according to the ontology formalism supported. For managing ontology instances, a table $C(p_1, \ldots, p_n)$ is created for each class `C` where $p_1, \ldots, p_n$ are the set of single-valued properties used at least by one instance of the class. Multi-valued properties are represented by a two-column table such as in the binary representation or by using the array datatype available in relational-object DBMS. Since all instances do not necessarily have a value for all properties of the table, this representation can be sparse which can impose performance overhead. The $\mathcal{SDB}$ OntoDB uses the horizontal storage layout for storing instances, ontologies and the ontology formalism.

2.3 Architectures of $\mathcal{SDB}$

According to the abstraction layers managed (instances, ontologies and ontology formalism) and the storage layouts used, $\mathcal{SDB}$ are decomposed into one or several schemes leading to different architectures illustrated Figure 2.

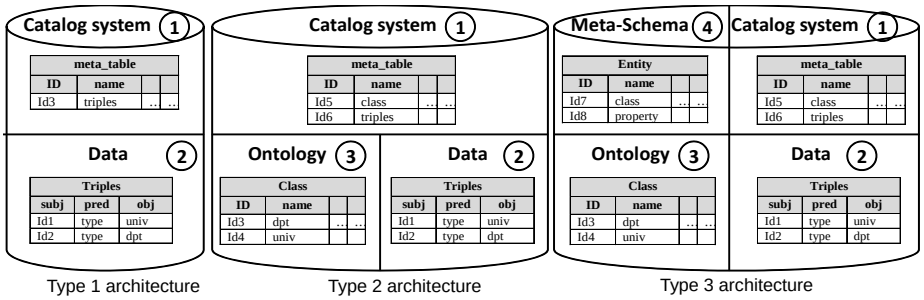

Fig. 2. Different Architectures of $\mathcal{SDB}$

- *Type 1 architecture:* some $\mathcal{SDB}$ like Oracle or Jena use only one schema to store all information. As a consequence, these $\mathcal{SDB}$ have two parts like classical database: the data and system catalog parts.
- *Type 2 architecture:* other $\mathcal{SDB}$ like IBM SOR [5] have chosen to separate the storage of ontology instances from the storage of the ontology (classes and properties) in two different schemes. This separation leads to $\mathcal{SDB}$ composed of three parts: data, ontology and system catalog.
- *Type 3 architecture:* $\mathcal{SDB}$ like OntoDB [1] have introduced a fourth part to store the ontology formalism used. This part, called the *metaschema* in OntoDB, is a specialized system catalog for the ontology part. With this part, users can modify the ontology formalism used in the $\mathcal{SDB}$.

3 Formalization of $\mathcal{SDB}$ and comparison features

3.1 Formalization of $\mathcal{SDB}$

In the previous sections we have seen that $\mathcal{SDB}$ presents an important diversity in terms of architecture, storage layouts and ontology formalisms supported. To provide a general view of $\mathcal{SDB}$ that can be used as a basis for the physical design of $\mathcal{SDB}$, we propose the following formalization:
$\mathcal{SDB} : < MO, I, Sch, Pop, SM_{MO}, SM_{Inst}, Ar >$, where :

- MO represents the ontology (model part). It is formalized by the following 5-tuple: $< C, P, Applic, Ref, Formalism >$ where :
  - C represents the classes of the ontology.
  - P represents the properties of the ontology.
  - $Applic : C \rightarrow P^2$ is a function returning all properties whose domain is a given class.
  - $Ref : C \rightarrow (operator, exp(C))$ is a function that maps each class to an operator (inclusion or equivalence) and an expression of other classes. It is used to represent the relationship between classes (subsumption, Boolean operators, etc.).
  - Formalism is the ontology formalism used to define the ontology.

  For example, an OWL ontology is defined by: $< Classes, Properties, Applic, descriptionlogicoperators, OWL >$ where $descriptionlogicoperators$ is the set of description logic operators supported by the given version of OWL.
- I: represents the set of ontology instances.
- $Sch : C \rightarrow P^2$ is a function that maps each class to the set of properties valued by at least one instance of this class.
- $Pop : E \rightarrow 2^I$ is a function that associates a class to its instances.
- SM_{MO} is the storage layout used to represent the ontology (vertical, horizontal or binary).
- SM_{Inst} is the storage layout used to represent the ontology instances.
- Ar is the $\mathcal{SDB}$ architecture ($Type_1$, $Type_2$, $Type_3$).

According to this formalization, the $\mathcal{SDB}$ Oracle is represented by: $SDB_{Oracle} :< MO :< Classes, Properties, Applic, Operators\ (RDFS, OWLSIF$ or $OWLPrime), (RDFS$ or $OWL)>, RDFInstances, \phi, tablesRDF\_link$ and RDF_values giving instances of each class, $Vertical, Vertical, Type_1>$
This model gives a general view on the diversity of $\mathcal{SDB}$. In the next section, we gives a more precise comparison of $\mathcal{SDB}$ by defining some key futures of these systems.

3.2 Key features of $\mathcal{SDB}$

To deepen our study, we have compared six $\mathcal{SDB}$: three coming from research (OntoDB [1], Sesame [3] and Jena [2]) and three coming from industry (Oracle [4], DB2RDF [6] and IBM SOR [5]). From our study, we have identified a set of key features of $\mathcal{SDB}$. These features include :

- *Natural language support:* this feature consists in using the linguistic information (sometimes in several natural languages) available in an ontology.
- *Query language:* the query language supported by the $\mathcal{SDB}$.
- *Version and evolution of ontologies:* this feature consists in providing a way of modifying an ontology while keeping track of the modification done.
- *Inference support:* this feature consists in supporting the inference rule defined for the ontology formalism supported.
- *User-defined rules:* this feature consists in allowing users to define their own derivation rules.

Table 1 shows these key features of $\mathcal{SDB}$ (in addition to the previous identified criteria) and their availability in the $\mathcal{SDB}$ studied.

Table 1. Comparative study of $\mathcal{SDB}$ (V : vertical, H: horizontal, B: binary, H: hybrid)

Features	**Oracle**	**SOR**	**OntoDb**	**Jena**	**Sesame**	**Db2rdf**
Formalism supported	RDF, OWL	OWL	PLIB	RDF, OWL	RDF, OWL	RDF
Natural Language Support	yes	yes	yes	no	no	no
Query Languages	sql, sparql	sparql	ontoql, sparql	rql, Rdql, sparql	serql, sparql	sql, sparql
Version and evolution	no	no	yes	no	no	no
Inference support	yes	yes	no	yes	yes	no
User-defined rules	yes	no	no	yes	yes	no
Ontology Storage layout	V	H	H	V, H	V, H	V
instances Storage layout	V	B	H	H	V, B	V
Architecture	$Type_1$	$Type_2$	$Type_3$	$Type_1$	$Type_1$	$Type_1$
Underlying DBMS	Oracle	DB2, Derby	Postgres	Oracle, Postgres, Mysql	Oracle, Postgres, Mysql	DB2

The performance of current $\mathcal{SDB}$ is also an important feature. We propose a theoretical comparison through a cost model (Section 4) and an empirical comparison of six $\mathcal{SDB}$ to validate our cost model (section 5).

4 Cost Model

The cost model is an important component of a query optimizer. It can be used for important tasks such as selecting the best query plan or using adaptive optimization techniques. In this section we propose a cost model for $\mathcal{SDB}$ that takes into account their diversity.

4.1 Assumptions

Following assumptions of classical cost models such as those made in System R, we assume that (1) computing costs are lower than disk access costs, (2) statistics about the ontology and their instances are available (e.g., the number of instances by class) and (3) the distribution of values is uniform and attributes are independent of each other.

We also need to make specific assumption for $\mathcal{SDB}$. Indeed logical inference plays an important role in $\mathcal{SDB}$. A $\mathcal{SDB}$ is said to be *saturated* if it contains initial instances and inferred instances, otherwise it is called *unsaturated*. Some $\mathcal{SDB}$ are saturated during the data loading phase either automatically (e.g., IMB SOR) or on demand (e.g., Jena or Sesame). Other $\mathcal{SDB}$ (e.g., Oracle) can be saturated at any time on demand. Our cost function relies on the saturated $\mathcal{SDB}$ (it does not take into account the cost of logical inference).

4.2 Parameters of the Cost Function

As in traditional databases, the main parameters to be taken into account in the cost function of $\mathcal{SDB}$ are: the cost of disk access, the cost of storing intermediate files and the computing cost. Let q be a query and sdb a semantic database against which q will be executed and $cost$ the cost function. In terms of architecture model, we can see that compared to a traditional cost model, the cost model in $\mathcal{SDB}$ is *increased*, resulting from the access cost to different parts of the architecture:

$$\begin{aligned}
&Type1 : cost(q, sdb) = cost(syscatalog) + cost(data)\\
&Type2 : cost(q, sdb) = cost(syscatalog) + cost(data) + cost(ontology)\\
&Type3 : cost(q, sdb) = cost(syscatalog) + cost(data) + cost(ontology)\\
&\qquad\qquad + cost(ontology\ meta - schema)
\end{aligned}$$

The cost to access the system catalog ($cost(syscatalog)$) is part of the cost model of classical databases. It is considered negligible because the system catalog can be kept in memory. Thus, the query cost function depends on the architecture and the storage model of $\mathcal{SDB}$.

We assume that the meta-schema and the ontology-schema are small enough to be also placed in memory. Hence in all architectures the cost can be reduced to the cost of data access, expressed as the number of inputs/outputs

($Cost(q, sdb) = cost(data)$). The cost of queries execution is heavily influenced by the operations done in the query, which are mainly projection, selection and join. Our cost function focuses on these three operations.

4.3 Our Queries Template

We consider queries expressed according to template below. These queries can be expressed in most semantic query languages like SPARQL, OntoQL, etc. If C_1, ..., C_n are ontology classes and p_{11}, ..., p_{nn} properties, the considered query pattern is the following:
$(?id_1, type, C_1)\ (?id_1, p_{11}, ?val_{11})\ \cdots\ (?id_1, p_{n1}, ?val_{n1})\ [FILTER()]$
$(?id_2, type, C_2)\ (?id_2, p_{12}, ?val_{12})\ \cdots\ (?id_2, p_{n2}, ?val_{n2})\ [FILTER()]$
...
$(?id_n, type, C_n)\ (?id_n, p_{1n}, ?val_{1n})\ \cdots\ (?id_n, p_{nn}, ?val_{nn})\ [FILTER()]$

4.4 Cost of Selections and Projections

In our template, a selection is a query that involves a single class. It has the following form: $(?id, type, C)(?id, p_1, ?val_1) \ldots (?id, p_n, ?val_n)[FILTER()]$. We distinguish *single-triple* selections, which are queries consisting of a single triple pattern, and *multi-triples* selections, consisting of more than one triple pattern (all concerning a single class). In the vertical and binary representations only *single-triple* selections are interpreted as selections, because *multi-triples* selections involve joins. We define the cost function as in relational databases. For *single-triple* selection, our function is equal to the number of pages of the table involved in the query:

- vertical layout: $Cost\ (q, sdb) = |T|$, where $|T|$ is the number of pages of the table T. If an index is defined on the triples table, $cost(q, sdb) = P(index) + sel(t) * |T|$, where *P(index)* is the cost of index scanning and *sel(t)* is the selectivity of the triple pattern t as defined in [11].
- binary layout: the selection is done on the *property tables*. $Cost(q, sdb) = |Tp|$ where Tp is the *property table* of the property of the query triple pattern. With an index on the selection predicate, $cost(q, sdb) = P(index) + sel * |Tp|$, where *sel* is the selectivity of the index.
- horizontal layout: the selection targets the tables of classes domain of the property of the query triple pattern. $Cost(q, sdb) = \sum_{Tcp \in dom(p)}(|Tcp|)$, where Tcp are the tables corresponding to the classes domain of the property of the query triple pattern. If there is an index defined on the selection predicate, $cost(q, sdb) = \sum_{Tcp \in dom(p)}(P(index) + sel * |Tcp|)$ where *sel* is the index selectivity.

Multi-triples selection queries are translated into joins in vertical and binary layouts. In the horizontal layout, the cost function is the same as the one defined for *single-triple* selection. A projection is a free filter selection having a list of attributes whose size is less than or equal to the number of properties of the class on which it is defined. Its cost is defined in the same way as the selection.

4.5 Cost of Joins

Join operations are done in queries that involve at least two triple patterns. In the horizontal layout these triple patterns must belong to at least two classes from different hierarchies. In the vertical layout, we note that self-join of triples table can be done in two ways: (1) a naive join of two tables *i.e.,* a Cartesian product followed by a selection. We call this a *classic join*. (2) a selection for each triple pattern followed by joins on selection results. We call that a *delayed join*. We consider separately these two approaches of self-join of triples table. Our cost function for other approaches is the same as in relational databases, and depends on the join algorithm. Only the cost of hash join is presented in this paper (Tab. 2 where V, B and H represent respectively the Vertical, Binary and Horizontal layouts).

Table 2. Cost of joins (dom(p) : domain of p).

Hash join index	*delayed join*	*classic join*
V: join(T,T), T triples table	$2 * \|T\| + 4 * (\|t1\| + \|t2\|)$	$6 * \|T\|$
B: join(T_1, T_2), T_i Tables of prop.	not applicable	$3 * (\|T_1\| + \|T_2\|)$
H: join(T_1, T_2), T_i tables of classes	not applicable	$\sum_{T \in dom(p)} 3(\|T_1\| + \|T_2\|)$

4.6 Application of Cost Model

To illustrate our cost function, we considered queries 2, 4 and 6 of the LUBM benchmark. We have translated these queries according to the different storage layouts. Then we have made an evaluation of their cost using statistics of the Lubm01 dataset (given in the section 5). Figure 3 presents the results obtained. It shows that processing a self-join query on the triples table with a *classic join* requires more I/O than with a *delayed join*. We observe that for a single-triple selection, the binary layout requires less I/O, so it is likely to be more efficient than other storage layouts. In other types of queries (join queries and multi-triples selections), the horizontal layout provides the best results. This theoretical results have been partially confirmed by our experimental results presented in the next section. Indeed as we will see, OntoDB (horizontal layout) provide the best query response time followed by Oracle (vertical layout). However for some $\mathcal{SDB}$ such as Jena, Sesame and DB2RDF, the evaluation results do not clearly confirm these theoretical results. We believe this is due to the specific optimization done in these systems.

5 Performance Evaluation

We have run experiments to evaluate the data loading and query response time of the six considered $\mathcal{SDB}$. As OntoDB was originally designed for the PLIB formalism we have developed a module to load OWL ontologies.

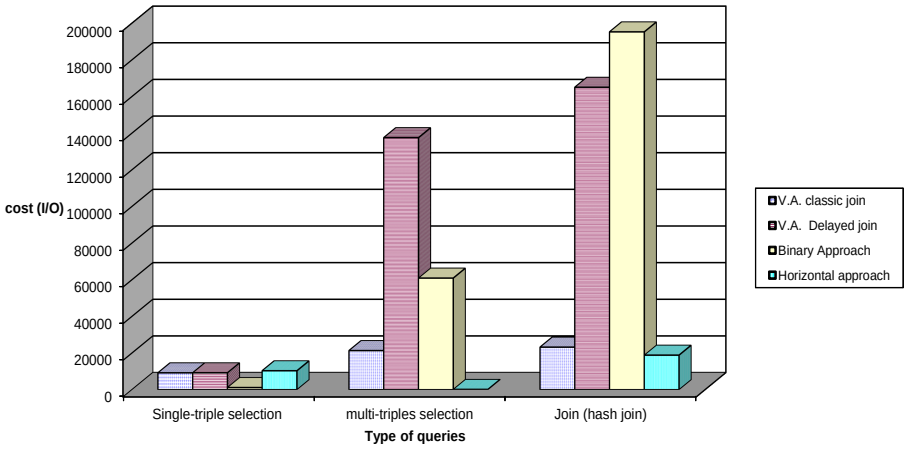

Fig. 3. Queries costs provided by our cost model (V.A. : Vertical approach)

5.1 Dataset used in experiments

We used the benchmark of Lehigh University (denoted LUBM) to generate five datasets with respectively 1, 5, 10, 50 and 100 universities (denoted respectively Lubm01, Lubm05, Lubm10, Lubm50 and Lubm100). The number of instances and triples generated are presented in Table 3. Our experiments were conducted on a 3.10 GHZ Intel Xeon DELL personal computer with 4GB of RAM and 500GB of hard disk. We have used the following DBMS: Oracle 11g, IBM DB2 9.7 for SOR and DB2RDF and PosgresSQL 8.2 for OntoDB, Jena and Sesame.

Table 3. Generated datasets

Dataset	Lubm01	Lubm05	Lubm10	Lubm50	Lubm100
# instances	82415	516116	1052895	5507426	11096694
# triples	100.851	625.103	1.273.108	6654856	13405675

5.2 Performance in Terms of Data Loading Time

We loaded and measured the loading time four times for each dataset on each $\mathcal{SDB}$ and took the average time. Results are given in Table 4. Since Sesame can be used as a $\mathcal{SDB}$ with vertical or binary approach, we tested these two approaches. In the following, we call *SesameSdbI* and *SesameSdbII* the Sesame system implemented respectively with the vertical and binary layout. For Oracle we used the rule base *owlprime*.

Table 4. Summary of loading time (in sec).

Dataset	Lubm01	Lubm05	Lubm10	Lubm50	Lubm100
Oracle	12	55	116	1562	39216
DB2RDF	11	53	109	2322	22605
Jena	25	188	558	40862	147109
SesameSdbI	16	158	391	23424	65127
SesameSdbII	27	231	522	33521	260724
OntoDB	15975	72699	146023	–	–
IBM SOR	90	590	1147	–	–

Interpretation of Results. In terms of loading time, as we can see in Table 4, Oracle and DB2RDF provide the best results. This can be explained by the specific and optimized loading mechanisms that have been build on the Oracle and DB2 DBMS. Compared to DB2RDF, SOR is slower as it takes a lot of time to make the inferences on the ontology and instances and we were not able to load LUBM50 and LUBM100. For the other $\mathcal{SDB}$, the results of the two versions of Sesame and Jena are similar with a slight advantage for the vertical layout of Sesame (the triple table which is a direct translation of the RDF model). OntoDB is slower that the other $\mathcal{SDB}$. This overhead is most probably due to our added loading module that imposes a new layer on top of the loading engine. Like with SOR, we were not able to load LUBM50 and LUBM100.

5.3 Performance in Terms of Queries Processing Time

We measured the query response time of queries on each $\mathcal{SDB}$ four times and we kept the average time. We used the 14 queries of the LUBM benchmark, adjusted to be accepted by our $\mathcal{SDB}$. The datasets used are Lubm01 and Lubm100. The results obtained led to the histograms presented Figure 4 and Figure 5. For readability we present the results of the queries that have a long response time in Table 5. As we were not able to load LUBM100 on OntoDB and SOR, the query response times for these dataset are not shown.

Table 5. Query response time on LUBM100 (Q2,Q6,Q7,Q9,Q14) (in sec)

	Q2	Q6	Q7	Q9	Q14
Oracle	240,75	1743	125,76	1973,32	473,23
DB2RDF	95552,5	12815,5	330454,67	692432	6477
Jena	372760,33	147696	224287,33	3538480	163798
SesameSDB1	32920,66	29922,66	21907,66	110498	16707,33
SesameSDB2	115819	70458	2266	242991	64107

Interpretation of Results. Regarding the queries response times, OntoDB reacts better than the other $\mathcal{SDB}$ for most queries executed on LUBM01. In-

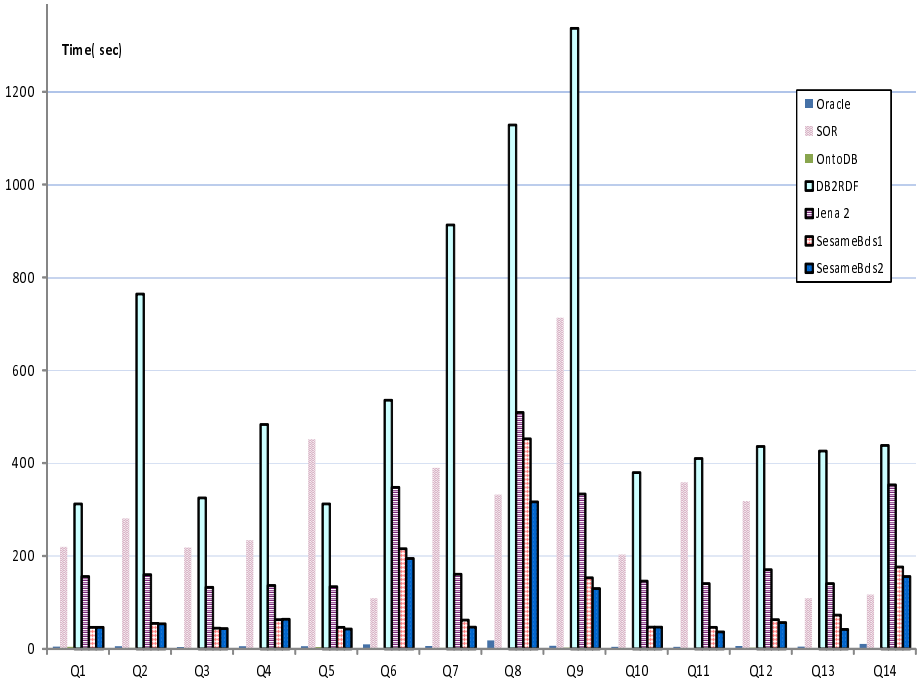

Fig. 4. Queries response time on LUBM01

deed, since OntoDB uses the horizontal layout and that queries involve several property, it performs less join operations than other $\mathcal{SDB}$. Indeed all queries on properties of a same class can be reduced to selection operations on table corresponding to this class, which is not the case when we use an other layout. Query 4 of the benchmark LUBM is a good illustration. This query is made of five triples patterns having all the same domain (*Professor*). This query does not need a join operation in OntoDB, but requires at least four joins in other systems. If the query involves less property (e.g., query 6 of LUBM is a simple selection query on a single property), the query response time is close to the $\mathcal{SDB}$ that use a binary layout as this layout also requires a single scan of a unique property table. For this query, the horizontal layout is the worse as it requires a scan of the whole triple table (Oracle, DB2RDF). We notice that even if Oracle uses an horizontal layout, the query response time are really close to OntoDB and better than the other $\mathcal{SDB}$. These good results are certainly due to the specific optimization techniques set up in this system. Indeed, several materialized view are used to reduce the need to scan the whole triple table. Considering the Sesame $\mathcal{SDB}$, we note that the binary layout implementation of Sesame (sesameBdsII) outperforms slightly the vertical layout (sesameBdsI). For the $\mathcal{SDB}$ based on DB2, the poor performance of DB2RDF can be explained by the fact that it uses a horizontal layout with a lot of tables linked to the triples table (and so

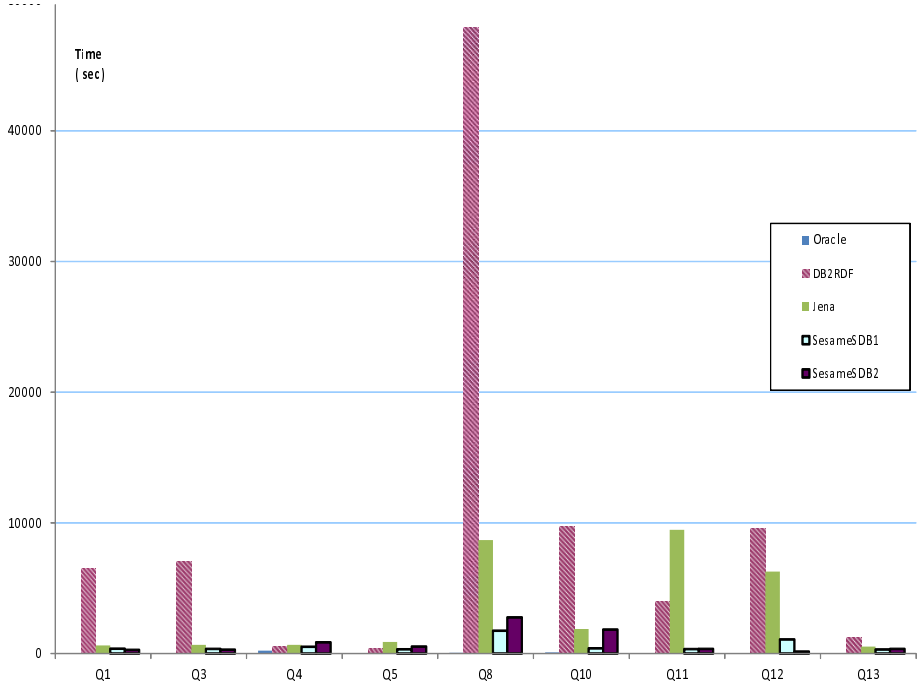

Fig. 5. Queries response time on LUBM100 (Q1,Q3,Q4,Q5,Q8,Q10,Q11,Q12,Q13)

need many joins in addition to the self-joins of the triple table). The result of SOR for LUBM01 are slightly better than DB2RDF but worst than the other $\mathcal{SDB}$. Like for the loading part, this result is due to inference done by SOR during the query processing. Indeed for the LUBM query 12, other $\mathcal{SDB}$ return an empty result since there is no explicit statement of an instance of the *Chair* class. On the contrary SOR returns several results thanks to a deductive process (an instance of *Professor* is also an instance of *Chair* if it is at the *headOf* a *Department*). If we add a rule reflecting this assertion to other $\mathcal{SDB}$ having an inference engine they would also provide answers (but with a worse execution time). Considering the deductive process, Jena and Sesame use inference engines based on a central memory management or file system, but they do not work on data stored in databases yet. It is also possible to use an inference engine during the data loading phase and to store all inferred data in the database. But if we do that the loading time will be worst.

6 Conclusion

In recent years, ontologies have been increasingly used in various domains. Therefore a strong need to manage these ontologies in databases has been felt. As a consequence, both academics and industrialists have proposed persistence solu-

tions based on existing DBMS. Unlike traditional DBMS, $\mathcal{SDB}$ are diverse in terms of ontology formalisms supported, storage layouts and architectures used. To facilitate the understanding of this diversity, we have studied six $\mathcal{SDB}$ and proposed a model that captures this diversity. Considering the performance of $\mathcal{SDB}$, we have conducted a study both theoretically by the definition of a cost model and empirically by measuring the data loading time and query processing time on the LUBM benchmark. The results show that our cost model predict the performance obtained for the $\mathcal{SDB}$ that do not use specific optimizations. Regarding the performances, we first note the effectiveness of industrial semantic databases in terms of data loading time. For the query response time, the results are different. The $\mathcal{SDB}$ that uses an horizontal layout give good results for most queries but the completeness of the inference process has to be taken into account. As a further step in our study of $\mathcal{SDB}$, we plan to modify the data and queries used in our experiment to determine under what conditions a storage layout and/or an architecture is better than an other. The application of our cost model for specific query optimization problem in $\mathcal{SDB}$ (e.g. materialized view selection) is also an important perspective for future work.

References

1. Dehainsala, H., Pierra, G., Bellatreche, L.: Ontodb: An ontology-based database for data intensive applications. In: Proceedings of the 12th International Conference on Database Systems for Advanced Applications (DASFAA'07). (2007) 497–508
2. Wilkinson, K., Sayers, C., Kuno, H., Reynolds, D.: Efficient rdf storage and retrieval in jena2. HP Laboratories Technical Report HPL-2003-266 (2003) 131–150
3. Broekstra, J., Kampman, A., van Harmelen, F.: Sesame: A Generic Architecture for Storing and Querying RDF and RDF Schema. In: Proceedings of the 1st International Semantic Web Conference (ISWC'02). (2002) 54–68
4. Wu, Z., Eadon, G., Das, S., Chong, E.I., Kolovski, V., Annamalai, M., Srinivasan, J.: Implementing an Inference Engine for RDFS/OWL Constructs and User-Defined Rules in Oracle. In: Proceedings of the 24th International Conference on Data Engineering (ICDE'08). (2008) 1239–1248
5. Lu, J., Ma, L., Zhang, L., Brunner, J.S., Wang, C., Pan, Y., Yu, Y.: Sor: a practical system for ontology storage, reasoning and search. In: Proceedings of the 33rd international conference on Very large data bases (VLDB'07). (2007) 1402–1405
6. IBM: Rdf application development for ibm data servers (2012)
7. Dean, M., Schreiber, G.: OWL Web Ontology Language Reference. World Wide Web Consortium. (2004) http://www.w3.org/TR/owl-ref.
8. Pierra, G.: Context representation in domain ontologies and its use for semantic integration of data. Journal Of Data Semantics (JoDS) **10** (2008) 174–211
9. Goasdoué, F., Karanasos, K., Leblay, J., Manolescu, I.: View Selection in Semantic Web Databases. Proceedings of the VLDB Endowment **5**(2) (2011) 97–108
10. Abadi, D.J., Marcus, A., Madden, S.R., Hollenbach, K.: Scalable Semantic Web Data Management Using Vertical Partitioning. In: Proceedings of the 33rd International Conference on Very Large Data Bases (VLDB'07). (2007) 411–422
11. Stocker, M., Seaborne, A., Bernstein, A., Kiefer, C., Reynolds, D.: Sparql basic graph pattern optimization using selectivity estimation. In: Proceedings of the 17th international conference on World Wide Web (WWW'08). (2008) 595–604