A Database Backend for OWL

Jörg Henß[1], Joachim Kleb[2], Stephan Grimm[2] and Jürgen Bock[2]

[1] Fraunhofer IITB
Fraunhoferstr. 1
76131 Karlsruhe, Germany
{henss}@kit.edu
[2] FZI Research Center for Information Technology
at the University of Karlsruhe
Haid-und-Neu-Str. 10-14
76131 Karlsruhe, Germany
{surname}@fzi.de

Abstract. Most Semantic Web applications are build on top of technology based on the Semantic Web layer cake and the W3C ontology languages RDF(S) and OWL. However RDF(S) embodies a graph abstraction model and thus is represented by triple-based artifacts. Using OWL as a language for Semantic Web knowledge-bases, this abstraction no longer holds. OWL is build up on an axiomatic model representation. Consequential storage systems focusing on the triple-based representation of ontologies seem to be no longer adequate as persistence layer for OWL ontologies. Our proposed system allows for a native mapping of OWL constructs to a database-schema without an unnecessary complex transformation in triples. Our Evaluation shows that our system performs comparable to current OWL storage systems.

1 Motivation

The requirements for Semantic Web applications often include the need to store large amounts of data, in particular when interconnected data is considered[3]. In such situations the accessible data is enormous, even if only parts of the data are accessed. This is also a common problem for semantic applications focusing on specific (closed) domains, e.g. within the medical sector. Two prominent ontologies are the FMA[4] and SNOMED[5] ontologies that include thousands of statements.

Considering the large quantity of data, the common in-memory processing patterns of most OWL-based Semantic Web applications seem to be inappropriate. Hence a storage system should be used when large ontologies are accessed.

Database systems offer several advantages that can improve the functionality of applications besides the scalable management, e.g. transaction based interaction, revision- and multi-user functionality.

[3] E.g. the linking-open-data-cloud http://linkeddata.org/
[4] http://sig.biostr.washington.edu/projects/fm/AboutFM.html
[5] http://www.ihtsdo.org/

Proceedings of OWL: Experiences and Directions 2009 (OWLED 2009),
Rinke Hoekstra and Peter F. Patel-Schneider, editors. http://www.webont.org/owled/2009

Regarding persistence layers for ontologies mostly triple-based systems have been proposed until now. However leaving the RDF(S) based triple view and moving forward to OWL as an object abstraction this schema seems no longer appropriate. In particular as this representation form turns out to be unnecessary complex and cumbersome.

In using an object-relational mapping we propose a more native way to design a persistence layer for OWL as an extension to the in-memory based OWL-API [1]. This approach focuses on the manipulation of ontologies in the first place (not reasoning) and has been validated through a comparison of our system to state-of-the-art competitors in this area.

The paper is structured as follows: Section 2 describes our approach for OWL-ontology persistence in detail and compares it to the triple-based storage model. Section 3 evaluates our approach compared to other systems focusing on the design similarities and differences. A summary and outlook is given in Sect. 4.

2 A Native Approach to Ontology Persistence

Native OWL persistence refers to a direct representation of OWL language constructs in an underlying storage layer one-to-one. Hence we abandoned the conventional way of representing an OWL ontology by subject-predicate-object triples in favour of an object centred way.

The triple-based representation pattern has been the foundation of ontology persistence design for a long time. It emerged from the RDF(S) graph abstraction but has some major drawbacks. In particular the storage of all triples in one table results in a huge and inefficient table including thousands or millions of triples. Furthermore several database indices, three to five are used commonly, on this table are required to avoid full table scans while querying. This table design results in the necessity of self-joins in order to retrieve associated data and thus in a further decrease of the overall performance of such stores.

OWL itself is expressible as RDF(S) triples and thus can be stored in triple-stores. This conversion to triple-form includes further drawbacks. For instance, complex OWL expressions, like cardinality, need at least four triples for storage and thus enlarge a persisted model if written in triples.

In order to avoid this conversion, a native representation of OWL in the form of axioms and objects is indispensable. This includes a suitable table design adapted to the features of OWL and avoiding the drawbacks of RDF(S)-stores, as the amount of self-joins.

The ability to interact with ontologies necessitates an interaction layer given by an API. Current RDF-store implementations, like Jena, come along with an API including an RDF-abstraction for the access of the triple store. Jena, for example, via its API provides access to an internal graph-model of nodes and their relations in order to handle RDF(S) ontologies. There also exists an OWL model for Jena, but it is based on the graph-model as well. In our approach we decided to use the OWL-API[6] [1]. This API has been implemented in the Java

[6] `http://owlapi.sourceforge.net/`

programming language and already supports the features of the upcoming OWL 2 standard. It has additionally been used in the Protégé[7] ontology editor. The axiom-centred abstraction as well as the object abstraction for classes, properties and individuals had been the major decision criteria for this API.

2.1 Object-relational Mapping

Starting from the OWL object abstraction given by the OWL-API we implemented a mapping from the API to the database layer by considering different mapping strategies[8]. The details of an object-relational mapping include the design of the object-to-table transfer. Here we considered the design patterns given by W. Keller [2]. We concentrated on his proposed patterns for mapping inheritance, as the API utilises Java-Interfaces and thus includes multiple inheritance paths for objects. Keller suggests three variants for this kind of mapping: (a) *one inheritance tree one table*, (b) *one class one table* and (c) *one inheritance path one table*.

Option (a) stores all objects of an inheritance tree in the same table. Thus the table provides columns for all possible fields of objects in this tree. As one object holds only an excerpt of fields, many columns include a null value. This results in a sparse table with a small degree of utilisation. Option (b) stores each object in a separate table. Each inherited field is stored in the table of the according super-class and is accessible via a foreign key. The direct fields are stored in the table of the class. Option (c) is a direct variant of (b), that also stores the inherited fields in the table of the actual class. The choice of an appropriate option has to consider the criteria of write-, read- and query-time as well as space consumption and flexibility. We decided for a mixed form between variant (a) and (b) in order to aggregate the advantages of a high-performing write-, read- and query-time given through variant (a) and a minimisation of space and a high flexibility through option (b). We tried to keep the overall number of tables as small as possible by representing class tables according to variant (b). We shared the same set of fields for several classes as well as the same parent class in a single table.

The majority of tables in our schema represents the mapping of complex inclusion axioms, *e.g.* range, domain and cardinality restrictions as well as class descriptions. The remaining axioms constitutes only a small fraction to the overall schema.

Essentially three tables are involved: a table containing the URIs of the individuals, a table holding class assertions of the individuals and a table containing the relationships of individuals. On closer inspection the latter has a significant similarity to a triple structure (object-property-subject, cf. [3]).

We introduced an additional table containing general information of the ontology, such as its URI. Also the primary key of all objects and their types are inserted in a single table, for a smooth handling of the inheritance dependencies.

[7] `http://protege.stanford.edu/`
[8] In particular we used Hibernate: `http://www.hibernate.org/`

We focused on the *direct manipulation* of ontology elements. Hence the resulting schema might differ from a schema optimised for a reasoner implementation. Here the contradiction between a specific knowledge-base representation according the algorithms of particular reasoners and a representation enabling direct-manipulation becomes obvious.

3 Evaluation

Performance and scalability are two major criteria for describing the quality of a persistence system. In particular this holds for systems that allow for the access of knowledge-bases that could differ largely in size and complexity.

As we did not yet contemplate reasoning in our implementation, our proposed evaluation differs undoubtedly from evaluation of ontology reasoners. Hence all performed queries handle explicitly stated data and do not depend on reasoning. This is due to the fact that in general each reasoner uses a proprietary ontology model, not related to third-party-APIs, like the OWL-API.

3.1 Queries and Used Systems

We evaluate the quality of our approach against several similar systems based on a theoretical comparison of the used schemas, the architectural differences as well as a practical evaluation according to the respective response times for several query tasks. We selected a mixture of common queries that are likely to occur in a normal interaction with an ontology store. Additionally we include queries that use complex relations or yield large result sets.

In particular we have chosen three systems to compare to our system:

- IBM SOR [5] is an ontology reasoning and storage system. It is primarily focused on rule based reasoning on top of the RDBMS as proposed in the OWL 2 RL profile [6].
- Jena [7] is an open source semantic framework. It uses a graph based structure and has its main focus on RDF(S), but also supports OWL ontologies. Jena SDB is a database persistence backend for the Jena graph.
- Owlgres [8] is an open source OWL reasoning and querying system based on database techniques and designed for conjunctive query answering. It is limited to the DL-Lite [9] fragment, on which OWL 2 QL [6] is based on.

3.2 Persistence Store Designs

Database-based ontology persistence systems can be divided into two groups: *schema-aware* and *schema-unaware* [10]. The systems considered in this paper all belong to the group of schema-unaware systems, which are commonly used and provide maximum flexibility as they use a fixed database schema that has no direct dependencies to the stored ontology.

Jena SDB is a typical representative of the triple based storage systems. All statements are persisted in one large triple table, such that the triple parts

are foreign-key references to nodes. Those nodes are stored in a separate table containing information on the node type and lexical value, e.g. the URI or literal value. This table structure requires many self-joins while querying and has poor insertion speed as several indices are used.

Owlgres uses a more refined database structure aligned to the DL fragment DL-Lite, on which it is based. The used structure has a clear separation between TBox and ABox tables. There exists a table containing all entities referenced in the TBox, e.g. classes or object properties, storing the entities type, lexical value and frequency, that is used for optimising queries on the ABox. Furthermore, Owlgres stores information on role and concept inclusions, e.g. subclassing, in separate tables. The ABox is represented by a table containing all individuals, a table storing the class assertions and two tables containing individual relationships partitioned into data and object properties. The latter two have a triple structure to represent statements. Altogether the store design yields a higher selectivity while querying.

The store design employed by SOR was chosen with respect to the rule-based reasoning. As those rules are derived from the OWL language, the table structure is strongly related to the axiom types and entities of OWL. The schema can be split into TBox and ABox tables as well. SOR supports a broad range of OWL TBox axioms, e.g. restrictions and intersections. An overview of the used tables structure can be found in [11]. The ABox table structure employed by SOR is very similar to Owlgres, though SOR has a more refined way of storing literals.

Our system uses a store design derived from the object model of the OWL-API, thus it is directly related to the OWL axiom types (cf. Sect. 2.1). Actually the derived table structure is quite similar to the structure used by SOR, resulting in likewise advantages. Additionally, our schema contains tables that are attributable to the object relational mapping approach. Our schema can store any kind of OWL axioms without the need to convert them to semantic equivalents, such as SOR, which stores equivalent classes as mutual subclasses.

3.3 Performance Test

In order to get comparable measures for the tested systems, we created a selection of queries and measured the response time. We selected the FMA, wine9 and LUBM ontologies to test on ontologies of different complexity. The statistical facts of those ontologies are shown in Table. 1.

Ontology	Expressivity	Axiom count				
		class	object property	data property	individual	annotation
FMA	ALUIN(D)	82336	246	147	881362	440032
wine9	SHIN(D)	188	25	0	106582	4
LUBM	ALEHI+(D)	50	47	4	8519	76

Table 1. Ontology Statistics

As we intended to test ontology persistence systems and not the underlying databases, we executed all tests using the same RDBMS. We also tried to minimise the effects of reasoning and switched it off where it was possible, without considerably affecting the systems functionality.

Import and Load Time At first an ontology has to be transferred to a database (import) and then reconstructed in order to allow for user access (load). The import process can be time and memory consuming. Especially SOR required large amounts of RAM as it builds up a large in-memory model of the ontology. Table 2 shows the measured times. Notice that our system cannot compete with Owlgres or SDB concerning import-times, but is at the level of SOR, as both systems do not use batch loading.

	FMA				LUBM				wine9			
	OWLDB	SDB	Owlgres	SOR	OWLDB	SDB	Owlgres	SOR	OWLDB	SDB	Owlgres	SOR
Import	21442138	10935259	**2388831**	30176795	46140	39656	48874	**32015**	314368	**61484**	123718	352323
Load	2188	3574	2654	**153**	1843	1813	**1281**	30234	1579	953	**47**	51200

Table 2. Import and load time (ms).

Retrieval Queries We used two kinds of instance retrieval queries to search by an annotation and by a class. These two query types could be answered by all systems in comparable times, though our system had problems answering the annotation query as this is not directly supported by the OWL-API. When querying for individuals of a class, Jena SDB is notably fast, as this hits an index.

	FMA				LUBM				wine9			
	OWLDB	SDB	Owlgres	SOR	OWLDB	SDB	Owlgres	SOR	OWLDB	SDB	Owlgres	SOR
Annotation	7332547	1007	123	**97**	922	**16**	**16**	47	14172	234	**10**	390
Class	633	61	**13**	17	16	16	**10**	47	3610	3828	**125**	360

Table 3. Retrieval Queries (ms).

Assertion Queries This category contains queries for the assertions of a given individual. In particular we query for class and property assertions. It is striking that Owlgres has no possibility to query for the class of an individual, thus we had to iterate the classes to find all assertions. Our system tended to answer these queries very fast.

Axiom Queries and Statistical Queries We also queried for specific types of OWL axioms: subclasses, inverse object properties, transitive object properties, though not all systems supported all of these, e.g. Owlgres does not support transitivity.

	FMA				LUBM				wine9			
	OWLDB	SDB	Owlgres	SOR	OWLDB	SDB	Owlgres	SOR	OWLDB	SDB	Owlgres	SOR
Individual	22	153	346927	**17**	15	31	156	**10**	**63**	**63**	4468	125
Object Property	636	3797	**18**	57	**5**	187	94	15	250	79	**47**	263
Data Property	923	1477	20	**14**	16	47	**3**	16	31	15	**3**	47

Table 4. Individual assertion retrieval time (ms).

	FMA				LUBM				wine9			
	OWLDB	SDB	Owlgres	SOR	OWLDB	SDB	Owlgres	SOR	OWLDB	SDB	Owlgres	SOR
Sub Classes	80872	1106825	**425**	2186	47	203	**3**	31	52	1359	**3**	93
Inverse Properties	79	18	**2**	72	16	15	**2**	31	47	110	**2**	63
Transitive Properties	29	**5**	-	121	**2**	16	-	32	**3**	**3**	-	15
All Classes	56706	312602	159371	**1914**	32	16	**2**	31	31	329	**3**	30
All Individuals	**49919**	1711948	-	219121	31	26218	**14**	32	1265	507524	203	**188**

Table 5. Statistical query retrieval time (ms).

This type of query was especially slow on SDB, while Owlgres benefited from its cached TBox. Furthermore we did statistical queries, e.g. get all classes or get all individuals.

4 Conclusion and Outlook

We presented a novel approach to ontology persistence focused on the OWL language. This approach allows for native storage of language depending constructs. We start from an API reflecting the design of OWL and utilise an object-relational mapping. We create a specific mapping optimised for performance and flexibility issues. Resulting from the carried out evaluation, our system performs comparable to its competitors. It even outperforms those on several queries, despite the fact that we have not yet optimised our system completely.

The evaluation showed the advantages of our system opposed to the others in terms of the architectural design especially when used for ontology editing, though there are still some issues, e.g. to support faster import using batch loading or to support caching strategies. Next steps are the elimination of left over triple structures in our schema, an extension regarding requirements due to OWL-evolution specific tasks as well as the implementation of full-text indices.

Our system is publicly available on *owldb.sourceforge.com*.

Acknowledgment The work presented in this paper has been funded by the German Federal State of Baden-Württemberg and by the German Federal Ministry of Economics and Technology in the THESEUS ("New Technologies for the Internet of Services", `http://theseus-programm.de/`) research programme under grant 01MQ07019.

References

1. Bechhofer, S., Volz, R., Lord, P.: Cooking the Semantic Web with the OWL API. In: Proceedings of the 2nd International Semantic Web Conference (ISWC'03). LNCS, Springer (2003)
2. Keller, W.: Mapping objects to tables - a pattern language. In: Proc. Of European Conference on Pattern Languages of Programming Conference (EuroPLOP). (1997)
3. Auer, S., Ives, Z.G.: Integrating ontologies and relational data. Technical Report MS-CIS-07-24, University of Pennsylvania Department of Computer and Information Science (11 2007)
4. Navathe, S., Ceri, S., Wiederhold, G., Dou, J.: Vertical partitioning algorithms for database design. ACM Trans. Database Syst. **9**(4) (1984) 680–710
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: VLDB '07: Proceedings of the 33rd international conference on Very large data bases, VLDB Endowment (2007) 1402–1405
6. Motik, B., Grau, B.C., Horrocks, I., Wu, Z., Fokoue, A., Lutz, C.: OWL 2 Web Ontology Language: Profiles. World Wide Web Consortium, Working Draft WD-owl2-profiles-20081202 (December 2008)
7. Carroll, J.J., Dickinson, I., Dollin, C., Reynolds, D., Seaborne, A., Wilkinson, K.: Jena: implementing the semantic web recommendations. In: WWW Alt. '04: Proceedings of the 13th international World Wide Web conference on Alternate track papers & posters, New York, NY, USA, ACM (2004) 74–83
8. Stocker, M., Smith, M.: Owlgres: A scalable owl reasoner. In Dolbear, C., Ruttenberg, A., Sattler, U., eds.: OWLED. Volume 432 of CEUR Workshop Proceedings., CEUR-WS.org (2008)
9. Calvanese, D., Giacomo, G.D., Lenzerini, M., Rosati, R., Vetere, G.: Dl-lite: Practical reasoning for rich dls. In: Proc. of the 2004 Description Logic Workshop (DL 2004). Volume 104 of CEUR Electronic Workshop Proceedings, http://ceur-ws.org/. (2004)
10. Pan, Z., Heflin, J.: Dldb: Extending relational databases to support semantic web queries. (2004)
11. Zhou, J., Ma, L., Liu, Q., Zhang, L., Yu, Y., Pan, Y.: Minerva: A scalable owl ontology storage and inference system. In: ASWC. (2006) 429–443