A Protégé 4 Backend for Native OWL Persistence

Jörg Henß Joachim Kleb Stephan Grimm

Fraunhofer IITB
Karlsruhe Germany
henss@iitb.fraunhofer.de

FZI Research Center
for Information Technologies
at the University of Karlsruhe
{kleb, grimm}@fzi.de

Abstract

We present a persistence layer for native storage and manipulation of OWL ontologies on top of the OWL API and an associated integration of the first version of this OWL persistence layer into the Protégé ontology engineering environment. This allows for an efficient handling of large ontologies within the Protégé 4 environment even if they do not fit in main memory. The approach is based on a direct mapping from native OWL constructs to database entries by utilising a framework for object-relational mappings.

1 Motivation

There are numerous reasons that demand a scalable persistence layer for Protégé, e.g. the capability to process large ontologies. Former versions of Protégé (3.x) come along with persistence solutions that enable the database storage of ontologies. By evolving from version 3 to 4 - and thus from the former frame based architecture to an architecture that supports the Web Ontology Language (OWL) inherently - a gap emerges concerning the persistence of ontologies. Following the decision to abandon the frame based approach, this calls for a redesign of the former database storage format.

In our approach to an OWL persistence layer, we utilise the OWL API[1] [2] object model for the storage of native OWL language constructs to derive an appropriate database schema. In particular we focus on the axiomatic view given through the OWL API object model. We use an object-relational (O/R) mapping to realise native OWL persistence as a database backend for the OWL API. Since Protégé 4 builds on the OWL API, this persistence layer can readily be used as a database storage solution for Protégé. The reuse of the available store implementations for Protégé 3.x was not possible, as these implement the CLOS Meta-Object Protocol[2] [4, Ch. 5/6]. Also the use of available triple stores, like jena[3] or sesame[4], seemed not appropriate as they share the same drawbacks as the CLOS model on the database layer. In particular, the persistence solutions for those models are realised by the use of a single table, which results in a major performance loss.

During the realisation, a major design issue was the non-invasive implementation by avoiding changes on the OWL API elements, hence ensuring full compatibility. Moreover the approach provides a plug-in for Protégé and thus an additional opportunity for ontology persistence without changing the Protégé core code.

In the following Section we explain the advantages of a native OWL persistence layer and the reasons for the chosen database schema. In Sec. 3 the integration of the database back-end in Protégé 4 is explained. Section 4 entails the next development steps of the persistence solution and the conclusion.

2 Native OWL Persistence Layer

Native OWL persistence refers to a direct way of representing OWL language constructs in an underlying storage layer one-to-one. This is in contrast to triple-based storage solutions, where an OWL ontology is represented at the more fine-grained level of triples. Avoiding the conversion to the triple structure is

[1]http://owlapi.sourceforge.net/
[2]http://protege.stanford.edu/doc/design/jdbc_backend.html
[3]http://jena.sourceforge.net/
[4]http://www.openrdf.org/

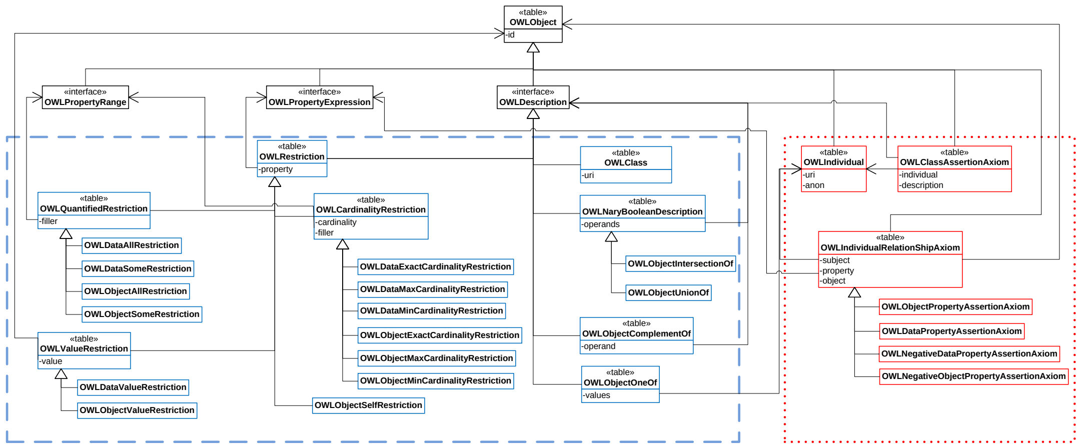

Figure 1: UML structure showing complex classes (blue dashed area) and ABox (red dotted area) axioms. Classes stereotyped with ≪*table*≫ contribute a table to our schema.

a major advantage, as this allows for faster storage and retrieval of OWL ontologies, saving processing time for conversion. Moreover querying the ontology can be speed up, as less (self-)joins on large triple structures are required. Furthermore complex OWL expressions, like cardinalities[5], can be stored in a more compressed way. To achieve this nativeness, we build the representation of an OWL ontology in our persistence layer on the OWL API object model as the basis for a mapping to database tables.

We propose the use of an object-relational (O/R) mapping approach for native persistence of OWL ontologies, by mapping the OWL axioms of an ontology directly to a database schema and thus providing an axiomatic view on the database level. Derived from that, entities included in these axioms have to be persisted as well. By using cascaded insertions, a feature most common O/R mapping[6] tools include, all entities and axioms referenced by an inserted ontology axiom become equally persisted, this reflects the axiomatic view on the ontology. Relationships between objects in an ontology are persisted as foreign key references. We also gain several benefits from the usage of a relational database management system (RDBMS), eg. as stated in [5] we can use the build-in support for transactions, access control, logging and recovery. Beyond that, the query optimisation techniques of modern RDBMSs can be used to optimise query performance, and thus provide better scalability compared to specialised RDF stores [10].

Schema Representation through O/R-Mapping We use the OWL API object model as conceptual basis of our implementation. This close integration allows the user to directly interact with the persisted ontology via the OWL API in the same way as with in-memory ontologies. The solution has been seamlessly integrated in the OWL API (refer to Sect. 3). Ontology modularisation can be achieved by using the OWL import mechanism, that even allows mixing in-memory and persisted ontologies in our implementation.

The creation of the O/R mapping of OWL API objects to database tables, constituting our database schema, was done in a bottom-up approach. The axiom types, respectively objects which are mapped, are directly derived from the OWL API object model. Furthermore we added support for persistence of SWRL rules in the same manner, as the OWL API supports these likewise.

The concrete classes for axioms and entities of the OWL API object model, as well as their corresponding interfaces and abstract classes, constitute a class hierarchy. Hence our O/R mapping has to be hierarchic as well. In order to realise this hierarchic mapping, we use the "One Class One Table" pattern mixed with the "One Inheritance Tree One Table" pattern [3]. The resulting database schema consists of 56 tables, representing the hierarchic structure used. We tried to minimise the overall number of tables, by coalescing class tables, sharing the same set of fields and the same parent class, in a single table. The majority of tables in our schema is required for modelling TBox axioms, e.g. cardinality, range and domain restrictions or complex class descriptions, cf. Fig. 1. The ABox of the ontology constitutes a smaller fraction of our schema. It consists mainly of the entity table for the individuals, named *OWLIndividual*, storing the URIs, a type-of relation (*OWLClassAssertionAxiom*) asserting classes to individuals and a

[5]E.g cardinality restrictions are stored as 4 triples using a state of the art triple store
[6]Our implementation is based on hibernate (`http://www.hibernate.org`)

table representing the relationships of an individual (*OWLIndividualRelationshipAxiom*). Figure 1 shows a simplified schema for these axioms (red subgraph). Interestingly, the latter has three fields (subject, property and object) resembling a triple structure as proposed in [1].

For proper support of inheritance it is necessary to have a single table containing the primary key of all objects of the ontology and their types (*OWLObject*). In consequence this table grows very fast, leading to slower insertion performance. Vertical partitioning could be a way to reduce this performance loss. Additionally to the OWL API object model hierarchy, we introduce an extra table containing meta information about the ontology, e.g. its URI. Occurrences of redundancies in the persisted information are very rare.

Related Approaches Compared to other database persistence solutions for OWL ontologies, like IBM SOR [5] or Owlgres [9], it is remarkable that our approach yields a database schema quite similar to these approaches. Though we focus on a direct manipulation in contrast to those systems being focused on reasoning and querying tasks. Since our implementation already supports OWL 2 [7], as used in the OWL API, we have a slightly higher count of axiom types and tables. Furthermore we did no conversion of semantic equivalent axioms[7], supporting the concept of the OWL API as ontology manipulation interface. We expect a similar performance of our suggested schema, when used for those tasks[8]. An advantage of our solution - compared to the basic database persistence of triples - is the reduction of joins in case of queries and a higher selectivity on tables.

The choice of an O/R mapping as intermediate layer allows us to vary several design issues, in particular concerning details of the database schema[9] and caching, at deploy or run time. It should be mentioned, that our database schema is only one of several possible schemata derivable from the OWL API, because it is possible to refactor the database schema, e.g. choosing another pattern for inheritance mapping [3]. This can be done without changing the implemented queries, as our O/R mapping layer supports implicit inheritance. This ability makes our system more flexible than other solutions and allows for performance tuning without changing the implementation.

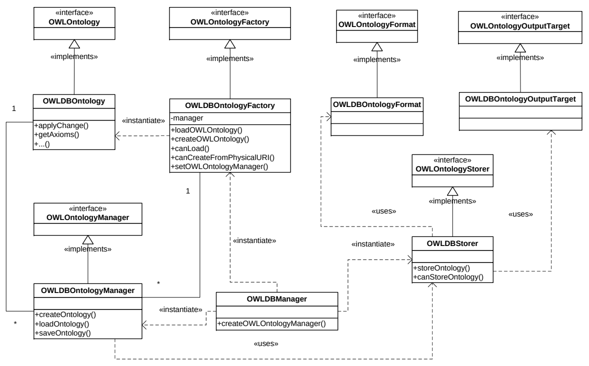

Figure 2: OWL API architecture

3 Integration into Protégé via OWL API

Our persistence solution is based on the object structure given through the OWL API, which it extends by the possibility of database storage. For this extension, we take into account the design issues of the OWL API following the design decisions of the predefined interface declarations. Figure 2 shows an overview of the involved OWL API classes and interfaces. `OWLDBOntology` implements the `OWLOntology`-Interface and offers a database-based realisation of an ontology. An object of this class can be created via the `OWLDBFactory`. Thereby the `OWLDBOntologyManager` offers access to the specific database settings. Prior to our modification, the OWL API only supported file-based formats as turtle, n-triple, rdf/xml, owl/xml, etc. According to the design pattern we implemented the additional format (`OWLDBOntologyFormat`) and thus enable the user to distinguish between the different persistence formats during runtime. The `OWLDBStorer` allows for the persistence of arbitrary ontologies in a database, while the `OWLDBManager`

[7]E.g. equivalent class can be modelled as mutual subclass

[8]It is very likely that support for SPARQL-DL [8] will be integrated into the OWL API

[9]E.g. the naming of tables

enables the convenient management of the persisted OWL ontologies. Through these implementations we are able to offer a seamless support of the database persistence and manipulation in Protégé.

We extended the Protégé-Open dialogue in order to offer an additional item “Open OWL Ontology from Database”. In a descending dialogue - similar to the already known dialogues in Protégé 3 - the user specifies the necessary database settings, as user-name, password, etc. Afterwards the ontology is loaded from the persistence store and displayed in Protégé. We also enabled the creation of a database ontology within Protégé.

The current implementation of Protégé 4 abolished the concept of project files. However there are several reasons to continue this concept. Apart of the storage of the GUI layout, the database settings could be stored, and reused to load the ontology next time Protégé is started.

Considering the collaborative development - already available in Protégé 3 via RMI - the support of transactional changes could be an advantage. For example, by using a RDBMS we also gain the transaction management capabilities and thus are able to realise the lost update problem.

4 Conclusion and Outlook

We presented our solution to enable database persistence for Protégé 4. Based on the OWL API we implemented a consequential extension for database storage that can be reused in Protégé. We concentrated on the advantages of an OWL-focused database schema for native owl constructs. The prototype of our application is available for the community at `http://www.fzi.de/downloads/ipe/owldb.zip`.

A further optimisation of our database schema by analysing frequent api-calls within usage scenarios seems to be possible. Furthermore it has to be investigated, if our implementation is suitable for rule based reasoning as proposed in [5], e.g. using the rules defined in OWL 2 RL [6].

Acknowledgements

Research was funded by the German Ministry of Economics BMWi as part of the THESEUS Project (01MQ07019).

References

[1] S. Auer and Z. G. Ives. Integrating ontologies and relational data. Technical Report MS-CIS-07-24, University of Pennsylvania Department of Computer and Information Science Technical, 11 2007.

[2] M. Horridge, S. Bechhofer, and O. Noppens. Igniting the OWL 1.1 Touch Paper: The OWL API. In C. Golbreich, A. Kalyanpur, and B. Parsia, editors, *OWLED*, volume 258 of *CEUR Workshop Proceedings*. CEUR-WS.org, 2007.

[3] W. Keller. Mapping objects to tables - a pattern language. In *Proc. Of European Conference on Pattern Languages of Programming Conference (EuroPLOP)97*, 1997.

[4] G. Kiczales. *The Art of the Metaobject Protocol*. The MIT Press, July 1991.

[5] J. Lu, L. Ma, L. Zhang, J.-S. Brunner, C. Wang, Y. Pan, and Y. Yu. SOR: A Practical System for Ontology Storage, Reasoning and Search. In *VLDB ’07: Proceedings of the 33rd international conference on Very large data bases*, pages 1402–1405. VLDB Endowment, 2007.

[6] B. Motik, B. C. Grau, I. Horrocks, Z. Wu, A. Fokoue, and C. Lutz. Owl 2 web ontology language: Profiles. World Wide Web Consortium, Working Draft WD-owl2-profiles-20081202, December 2008.

[7] B. Motik, P. F. Patel-Schneider, and B. C. Grau. Owl 2 web ontology language: Direct semantics. World Wide Web Consortium, Working Draft WD-owl2-semantics-20081202, December 2008.

[8] E. Sirin and B. Parsia. Sparql-dl: Sparql query for owl-dl. In C. Golbreich, A. Kalyanpur, and B. Parsia, editors, *OWLED*, volume 258 of *CEUR Workshop Proceedings*. CEUR-WS.org, 2007.

[9] M. Stocker and M. Smith. Owlgres: A scalable owl reasoner. In C. Dolbear, A. Ruttenberg, and U. Sattler, editors, *OWLED*, volume 432 of *CEUR Workshop Proceedings*. CEUR-WS.org, 2008.

[10] J. Zhou, L. Ma, Q. Liu, L. Zhang, Y. Yu, and Y. Pan. Minerva: A scalable owl ontology storage and inference system. In *ASWC*, pages 429–443, 2006.