JOPA: Stay Object-Oriented When Persisting Ontologies

Martin Ledvinka(✉) and Petr Křemen

Czech Technical University in Prague, Technická 2, Prague, Czech Republic
{martin.ledvinka,petr.kremen}@fel.cvut.cz

Abstract. Accessing OWL ontologies from IT systems can bring many problems unfamiliar to developers used to the more common relational storage approach. These problems stem from the dynamic nature of ontologies, their open-world character and expressiveness. In this paper, we present the Java OWL Persistence API (JOPA), a persistence layer allowing object-oriented access to semantic web ontologies. It supports features like caching, transactional processing and a semantically clear contract between the ontology and the object model. In addition, we present the OntoDriver, a software layer decoupling storage access from the object-ontological mapping. We provide an in-depth theoretical complexity analysis of our approach in connection with an analysis and practical evaluation of the performance of ontological storage with regards to application access scenario.

Keywords: Ontology · Persistence · Application access · Complexity

1 Introduction

Large expressive ontologies are a powerful tool for knowledge modelling. However, their complexity requires proper design and development of end-user information systems. During information system design, its creators face the challenge of choosing an appropriate software library that is reasonably easy to use and maintain, but that allows exploiting the ontology in its complexity [1].

On one side, information systems, that accept closed-world assumption by their nature, have to deal with distributed and open-world knowledge represented in ontologies. On the other hand, ontological changes often do not affect the information system data model assumptions and thus can be smoothly applied without information system recompilation and redeployment (e.g. taxonomy/ metadata extension). Furthermore, expressive power of semantic web ontologies is significantly higher than that of relational databases.

For example, an ontology specifies that each *Person* has a *name*. Due to the open world assumption, the ontology is consistent even if a particular *Person* does not have recorded his/her *name*. However, a genealogical application accessing the ontology needs the *name* to be known, which causes the application to crash whenever it receives (consistent, but application–incompatible) data from an ontological source, specifying a *Person* without a *name*.

This paper presents a solution for these issues – the Java OWL Persistence API (JOPA), see [2], a persistence layer that allows using the object-oriented paradigm for accessing semantic web ontologies. Comparing to other approaches, it supports validation of the ontological assumptions on the object level ([1,3]), advanced caching, transactions, unification and optimization of repository access through the OntoDriver component, as well as accessing multiple repository contexts at the same time. Additionally, we present a complexity analysis of OntoDriver operations that allows optimizing object-oriented access performance for underlying storage mechanisms. Next, we compare our solution to low level Sesame API in terms of efficiency. Last, we discuss several challenges of ontology access stemming from our experience in real-world application design.

Section 2 shows the relationship of our work to the state-of-art research. Section 3 introduces design and implementation of a prototype system for ontology-based information system access. Section 4 analyses complexity of operations defined in the API for storage access. In Sect. 5 we discuss the practical usage of both JOPA and OntoDriver. We further examine our experience from using JOPA in real-life application and the lessons we learned in Sect. 6. The paper is concluded in Sect. 7.

2 Related Work

Some object-oriented solutions try to approximate ontological OWL reasoning [4] by means of procedural code, like [5], or [6]. However, this significantly limits the expressive power of the ontology and is memory-consuming on the information system side.

There is another research direction, not compromising reasoning completeness, while maintaining its scalability – simplifying programmatic access to semantic web ontologies stored in optimized transactional ontology storages. This is also where our solution lies. Two main existing approaches are presented in the next sections.

2.1 Domain-Independent APIs

Many APIs for programmatic access to ontologies make no assumptions about the particular ontology schema. This paradigm is exploited in frameworks like OWL API [7], Sesame [8] or Jena [9]. These systems are generic, allowing to exploit full range of ontological expressiveness, trading it for verbosity and poor maintainability of the resulting code. Furthermore, using these tools requires software designers to hold deep knowledge of the underlying ontological structures. Comparing to these systems, our solution provides object-ontological mapping that helps software designers in keeping the design readable, consistent and short, see Sect. 5.1.

2.2 Domain-Specific APIs

There are already several established solutions, where the ontology schema is compiled directly into the object model. This paradigm makes use of an object-ontological Mapping (OOM). Representatives of this paradigm are e.g. Empire [10] or AliBaba[1]. Comparing to the former, these systems actually access ontologies in a frame-based (or object-oriented) manner. Object-ontology mappings bind the information system tightly to the particular ontology. This significantly simplifies programmatic access and is less demanding on the developer expertise in semantic web ontologies. However, it also discards most of the benefits of ontologies. The object model becomes as rigid as the model of applications based on relational databases, no difference is made between inferred and asserted knowledge and the application looses access to knowledge not captured in the domain model.

A thorough discussion of these architectures can be found in [2,3]. JOPA, introduced in Sect. 3, aims at taking the best of both types, as can be seen in Fig. 1. It provides compiled object-based mapping of the ontology schema similar to the domain-specific approaches described above, while also enabling access to the dynamically changing aspects of the ontology (see Sect. 3).

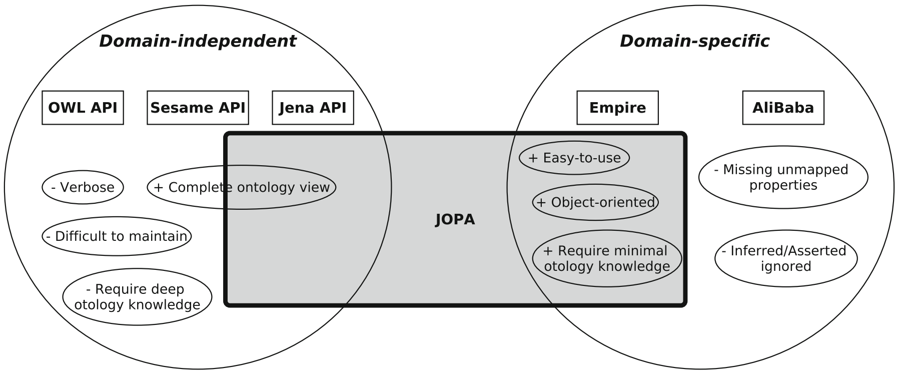


Fig. 1. JOPA compared to domain-independent and domain-specific approaches. The *complete ontology view* is included only partially in JOPA, because although it supports unmapped properties, accessing concepts not mapped by the object-model is limited in JOPA. It would require defining an entity mapping the *owl:Thing* concept, which would bring problems with object identity, as will be discussed in Sect. 6.1.

3 JOPA

JOPA stands for Java OWL Persistence API. It is in essence an API for efficient access to ontologies in Java, designed to resemble its relational-world counterpart Java Persistence API [11].

In this section, we introduce the architecture of JOPA.

[1] https://bitbucket.org/openrdf/alibaba, Accessed 04-08-2015.

Architecture. From the architectural point of view, JOPA is divided into two main parts:

OOM, realizes the object-ontological mapping and works as a persistence provider for the user application. The API resembles JPA 2 [11], but provides additional features specific to ontologies.

OntoDriver, provides access to the underlying storage optimized for the purposes of object-oriented applications. OntoDriver has a generic API which decouples the underlying storage API from JOPA.

Figure 2 shows a possible configuration of an information system using JOPA, together with some insight into the architecture of JOPA. The application object model is defined by means of a set of integrity constraints which guard that the ontological data are usable for the application and vice versa. Thanks to the well-defined API between the OOM part of JOPA and OntoDriver, there can be various implementations of OntoDriver and the user can switch between them (and between the underlying storages) without having to modify the actual application code.

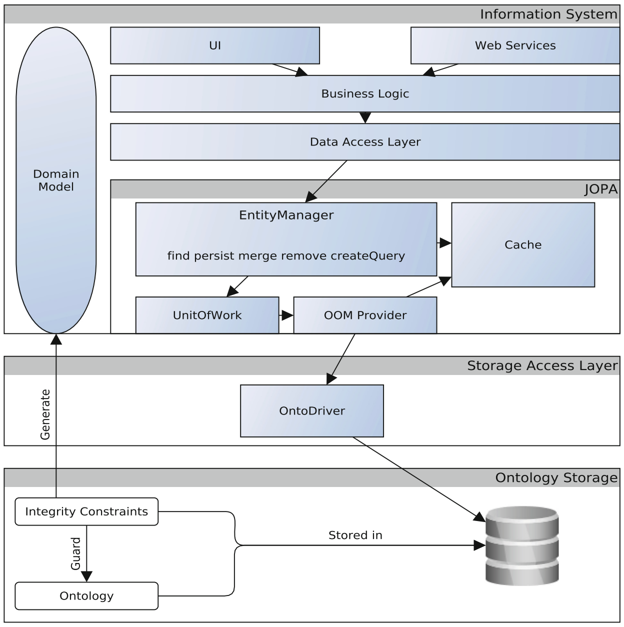


Fig. 2. JOPA Architecture. There are clear borders between the application, storage access layer and the storage itself. It is also visible that the domain model objects can be used throughout the application. Integrity constraints restrict the ontology so that it corresponds to the application's needs and annotations corresponding to these constraints are used to specify the domain model entity classes.

3.1 JOPA OOM

Let us now briefly describe the main features of the object-ontological mapping layer of JOPA.

The OOM layer is mainly represented by the `EntityManager` interface, which corresponds to its JPA 2 counterpart [11] to a large extent. It contains CRUD[2] operations: *find*, *persist*, *merge* and *remove*, but it enhances them with versions supporting context descriptors [2]. It also contains operations for transaction management and cache access.

Mapping. Object-ontological mapping is a mechanism of transforming ontological data, represented by concepts, properties and individuals, into classes, attributes and instances of the object-oriented paradigm and vice versa. This mapping enables one to define and use simple classes (called *entities*) as business objects in the application, a mechanism with which developers are familiar. Such an entity can be seen for example in Listing 1.1.

```
@OWLClass(iri="http://example.org/Student")
class Student {
  @Id(generated = true)
  URI id;
  @DataProperty(iri="http://example.org/name")
  String name;
  @DataProperty(iri="http://example.org/email")
  String email;
  @ObjectProperty(iri="http://example.org/course",
                  fetch = FetchType.LAZY)
  Set<Course> courses;
  @Inferred
  @Types
  Set<String> types;
  @Properties
  Map<String, Set<String>> properties;
}
```

Listing 1.1. Example of a business entity class declaration with JOPA annotations representing the object-ontological mapping.

The mapping is described by a set of integrity constraints, represented as Java annotations in the entity classes. Such integrity constraints define a clear contract between the object model and the ontology. The class declaration in Listing 1.1 reveals the simplicity of the description of the mapping. A class represents an ontological concept, data properties are attributes of primitive types (*String*, *Integer* etc.) and object properties are references to instances of other classes. The `types` field in the example then represents a set of concepts to which an individual mapped to an instance of this class belongs.

[2] Create, Retrieve, Update, Delete.

An ontological individual is fully identified by its IRI[3]. However, this is not the case in object oriented world, where the identity of each instance is tightly coupled with the class it belongs to. Therefore, JOPA places a constraint to the identity of individuals represented in it by requiring them to be explicitly stated to be of the specified ontology concept.

JOPA currently does not support blank nodes and anonymous individuals.

Unmapped Properties. One of the key features that differentiates JOPA from other ontology persistence frameworks like Empire and AliBaba is its ability to provide access to properties not captured by the object model. Such property values are represented by a map where the keys are property IRIs and values are sets of property values. The map is annotated with the `@Properties` annotation. This way the application has, although limited, access to the dynamic part of ontological data without having to adjust the domain model. See the `properties` attribute in Listing 1.1.

Inferred Attributes. Ontologies contain two types of information:

– *Explicit (asserted)*,
– *Implicit (inferred)*.

Inferred information cannot be changed, as it is derived from the asserted knowledge by a *reasoner* and can change only by modification of the explicitly stated information. As a consequence, it is necessary to prevent modification of inferred data. JOPA supports both asserted (in read/write mode) and inferred (read-only) attributes. This support is realized by means of the `@Inferred` and `@Asserted` annotations. The `@Asserted` annotation is optional. Every field not annotated with `@Inferred` is considered asserted and allowed to be modified.

Contexts. Another feature of JOPA is its ability to work with ontologies distributed in several contexts (graphs). When the underlying storage supports this feature, the application is able to specify not only in which context an instance should be searched for, but also contexts for individual attributes of the instance. If the context is not specified, the default one is used.

Transactions and Caching. JOPA supports transactional processing of the ontological data. However, the mechanism is different from standard relational-based persistence, because reasoning makes it more difficult to reflect pending changes to the transaction that produced them. For example, when a property value is changed during a transaction T_1, only T_1 has to be able to see effects of that change even before commit. JOPA itself does not employ any reasoning and offloads this burden to the underlying OntoDriver implementation. The OntoDriver is free to choose any strategy for keeping track of transactional changes. When a business transaction commits, JOPA tells the OntoDriver to make the pending changes persistent in the storage.

[3] Internationalized Resource Identifier.

Since applications often manipulate the same data, it is reasonable to use cache to reduce the necessity to query the storage. JOPA contains a *second-level cache* [11], which is shared between all open persistence contexts and enables quick entity lookup. Another performance improving feature is the support for lazily loaded attributes[4].

3.2 OntoDriver

OntoDriver is a software layer designed to decouple the object-ontological mapping done by JOPA from the actual access to the underlying storage. The goal for such decoupling is on the one hand the ability to switch between different storages without having to modify the application code. On the other hand, such layer enables vendor-specific optimizations of the storage access.

The concept of OntoDriver is similar to a JDBC[5] driver known from the relational world. But in contrast to JDBC, where all operations are done using SQL[6] statements, OntoDriver provides dedicated CRUD operations, which give the implementations more opportunity for optimizations, since they know beforehand what operation is executed.

However, the OntoDriver API does not eliminate the possibility of using SPARQL [12] queries for information retrieval and SPARQL Update [13] statements for data manipulation.

OntoDriver API. The key idea behind OntoDriver is a unified API providing access to ontology storages. To formally describe the API, let us first define basic ontological terminology:

Theoretical Background. We consider programmatic access to OWL 2 DL ontologies, corresponding in expressiveness to the description logic $\mathcal{SROIQ}(\mathcal{D})$[7]. In the next sections, consider an OWL 2 DL ontology $\mathcal{O} = (\mathcal{T}, \mathcal{A})$, consisting of a TBox $\mathcal{T} = \{\tau_I\}$ and an ABox $\mathcal{A} = \{\alpha_I\}$, where α_I is either of the form $C(i)$ (class assertion), or $P(i,j)$ (object property assertion), where $i, j \in N_i$ are OWL named individuals, $C \in N_c$ is a *named concept*, $P \in N_r$ is a *named object property*. Other axiom types belong to $\mathcal{T}$. W.l.o.g. we do not consider $C(i)$ and $P(i,j)$ for complex C and P here. We do not consider anonymous individuals either. See full definition of OWL 2 DL [4] and $\mathcal{SROIQ}(\mathcal{D})$ [14].

In addition to ontological (open-world) knowledge, a *set* $\mathcal{S}_\mathcal{C} = \{\gamma_i\}$ *of integrity constraints* is used to capture the contract between an ontology and an information system object model. Each integrity constraint γ_i has the form of an OWL axiom with closed-world semantic, as defined in [15].

4 Lazily loaded attribute values are retrieved from the data source only upon application request.

5 Java Database Connectivity.

6 Standard Query Language.

7 For the sake of compactness, we neglect datatypes and literals ($\mathcal{D}$) and use description logic notation.

By *multi-context ontology* we denote a tuple $\mathcal{M} = (\mathcal{O}_d, \mathcal{O}_1, \ldots, \mathcal{O}_n)$, where each $\mathcal{O}_I$ is an ontology identified by a unique IRI and is called *context*, $\mathcal{O}_d$ denotes the *default ontology (default context)* which is used when no other context is specified. This structure basically corresponds to an RDF dataset with named graphs [16]. An *ontology store* is a software layer that provides access to $\mathcal{M}$.

An *axiom descriptor* δ_a is a tuple $(i, \{(r_1, b_1) \ldots (r_k, b_k)\})$, where $i \in N_i$, $r_m \in N_r$, $b_m \in \{0, 1\}$ and $m \in 1 \ldots k$. The b_ms specify whether inferred values for the given role should be included as well. The axiom descriptor is used to specify for which information the OntoDriver is queried.

An *axiom value descriptor* δ_v is a tuple $(i, \{(r_1, v_1) \ldots (r_k, v_k)\})$, where $i \in N_i$, $r_m \in N_r$, $v_m \in N_i$ and $m \in 1 \ldots k$. The v_ms represent property assertion values for the given individual and property. The axiom value descriptor specifies information which shall be inserted into the storage.

Please note that for the sake of readability we have omitted context information from the formal definitions. In reality, a context can be specified for the whole descriptor and for each role.

OntoDriver API. The core operations of the OntoDriver API are as follows:

– $find(\mathcal{M}, \delta_a)$: $2^{\mathcal{M}} \times N_i \times N_r^k \times \{0, 1\}^k \rightarrow 2^{N_i \times N_r \times N_i}$, where δ_a is an axiom descriptor,
  - Given an individual, load values for the specified properties,
  - Used by `EntityManager.find()` in OOM,

– $persist(\mathcal{M}, \delta_v) = \mathcal{O}_d \cup \{\alpha_1 \ldots \alpha_s\}$, where $\alpha_1 \ldots \alpha_s$ are property assertion axioms created from role-value pairs in δ_v,
  - Persist axioms representing entity attribute values,
  - Used by `EntityManager.persist()` in OOM,

– $remove(\mathcal{M}, \delta_a) = \mathcal{O}_d \setminus \{\alpha'_1 \ldots \alpha'_t\}$, where $\alpha'_1 \ldots \alpha'_t$ are property assertion axioms for the roles specified in δ_a,
  - Remove axioms representing entity attribute values,
  - Used by `EntityManager.remove()` in OOM,

– $update(\mathcal{M}, \delta_v) = (\mathcal{O}_d \setminus \{\alpha'_1 \ldots \alpha'_t\}) \cup \{\alpha_1 \ldots \alpha_s\}$, where $\alpha'_1 \ldots \alpha'_t$ are original property assertion axioms for the roles $r_1 \ldots r_k$ defined in δ_v and $\alpha_1 \ldots \alpha_s$ are new property assertion axioms created for role-value pairs in δ_v,
  - Remove old and assert new values for entity attributes,
  - Used by `EntityManager.merge()` or on attribute change during transaction in OOM,

– $getTypes(\mathcal{M}, i, b)$: $2^{\mathcal{M}} \times N_i \times \{0, 1\} \rightarrow 2^{N_c}$, where the resulting axioms represent types of the specified individual i, b specifies whether inferred types should be included as well,
  - Get types of the specified named individual,
  - Used by `EntityManager.find()` in OOM,

– $updateTypes(\mathcal{M}, i, \{c_1 \ldots c_k\}) = (\mathcal{O}_d \setminus \{\alpha'_1 \ldots \alpha'_t\}) \cup \{\alpha_1 \ldots \alpha_k\}$, where $c_m \in N_c$, the α'_m are original class assertion axioms and the α_o are the new class assertion axioms for the given individual i,
  - Updates class assertion axioms for the given individual by removing obsolete types and adding new ones,

  - Used by `EntityManager.persist()`, `EntityManager.merge()` or on attribute change during transaction in OOM,
- *validateIC*$(\mathcal{M}, \{\gamma_1 \dots \gamma_k\}) : 2^{\mathcal{M}} \times 2^{N_i \times N_r \times N_i} \times \mathcal{S}_c \rightarrow \{0, 1\}$, where $\gamma_m \in \mathcal{S}_c$ and $m \in 1 \dots k$,
  - Validate the specified integrity constraints, verifying *reasoning-time* integrity constraints which cannot be validated at runtime [1],
  - Called on transaction commit in OOM.

The actual programming interface written in Java contains, besides methods representing the above operations, also methods for issuing statements (presumably SPARQL and SPARQL Update) and transaction managing methods. We omit these here for the sake of brevity.

Prototype of OntoDriver. To evaluate our design of OntoDriver, we have created a prototypical implementation. For this prototype, we have chosen to use Sesame API. One of the main reasons for such decision was that there exist Sesame API connectors for some of the most advanced ontology repositories including GraphDB (successor of OWLIM, see [17]) and Virtuoso [18]. The implementation can thus be used to access a variety of storages. More optimized implementations of OntoDriver which would exploit specific features of the underlying storages can be created, but the prototype was intended as a general proof of concept for the layered design of JOPA.

The Sesame OntoDriver uses neither SPARQL nor the SeRQL [8] language to perform data manipulation. We use the Sesame filtering API, which filters statements according to subject, predicate and object (i.e. it basically corresponds to triple pattern matching in a SPARQL query). On the one hand, this requires for example asking for each property of an individual separately (or asking for all of them by making the property unbound). On the other hand a SPARQL query that would correspond to the *find* operation (see above) would be a union of triple patterns. In addition, we have a more fine-grained control over the operation itself, because we are able to specify whether inferred statements should or should not be included in the query result. This is an important feature of JOPA and can generally not be done in standard SPARQL statements.

Another important point is how the Sesame OntoDriver deals with transactions. As was mentioned in Sect. 3.1, JOPA transfers the burden of making changes done in a transaction visible to the transaction itself to the OntoDriver. The prototype handles this task by creating local graphs of added and removed statements. When the store is queried for some knowledge, the added and removed transactional snapshots are used to enhance the results returned by the storage to reflect the transactional changes. These local graphs are of course unique to every transaction on the OntoDriver level. Currently, this approach is handicapped by the fact that such local graphs do not provide any reasoning support, so they represent only explicit assertions. A solution to this drawback would be for example using an in-memory reasoner, e.g. Pellet [19], for the local graphs.

We are also considering another possible solution for keeping the transactional changes. This solution would require temporary contexts created by the store, which would hold the transactional changes kept currently in the local graphs. This would enable us to transfer the reasoning task over to the underlying storage. This solution remains as an idea for the future development.

4 Operation Complexity Analysis

The OntoDriver API enables us to examine the complexity of operations it consists of. In this section we consider this complexity with regards to several selected ontology storages. A careful reader may have noticed that some of the operations in the API could share the same implementation, for instance $update(\mathcal{M}, \delta_v)$ can be implemented using $remove(\mathcal{M}, \delta_a)$ and $persist(\mathcal{M}, \delta_v)$. Thus, we concentrate the analysis on the following operations:

- $find(\mathcal{M}, \delta_a)$,
- $persist(\mathcal{M}, \delta_v)$,
- $remove(\mathcal{M}, \delta_a)$.

When done with theoretical complexity analysis, we will proceed to experimental evaluation of our theoretical assumptions.

4.1 Complexity Analysis

For the theoretical complexity analysis, we have selected two well known storages, each representing a different approach to reasoning – one performing total materialization on data insertion, the other reasoning at query time and doing no materialization (the difference being similar to forward and backward chaining strategies in rule systems):

GraphDB, formerly known as OWLIM [17], is a Sesame SAIL[8] with rule-based reasoner using forward chaining,
Stardog,[9] performs real-time model checking with no materialization.

Each of these strategies has its pros and cons. Total materialization is fast in querying, as there is no reasoning performed at query execution time. On the other hand, statement removal and insertion are slow. In addition it is necessary to specify reasoning expressiveness before any data is inserted. Total materialization can also cause significant inflation of the dataset size. Real-time reasoning keeps the dataset compact and it is fast on insertion, however performing reasoning at query time can be time consuming.

[8] Storage And Inference Layer.
[9] http://www.stardog.com, Accessed 02-12-2014.

A Note on Indexes. The most important part of every ontology storage is its index – it determines how quickly the data can be accessed. Ontology repositories follow the trend of data storages from other domains and use B-trees [20]. GraphDB uses a modified version of B-trees – a B+ tree [21]. There is not much information about the indexing strategies of Stardog, but we were able to determine that it also uses a B+ tree from a post in Stardog forum[10].

To efficiently access data which are statements consisting of three parts – *subject* (S), *predicate* (P) and *object* (O), the storages usually contain multiple indexes. Since there exist six combinations of the three statement parts, there could be up to six different indexes. With increasing number of indexes the space required to store the data and the indexes obviously grows. Another problem of multiple indexes is their updating when the data is modified. Given the fact that most storages also support contexts, the number of possible indexes grows even more.

Therefore, storages usually restrict themselves to only a few indexes, based on the structure of the most frequent queries. It is often the case that property is bound in such queries. Thus, storages mostly use PSO and POS indexes, with others optionally available. The PSO index searches statements first by *predicate*, then by *subject* and last by *object*. The POS index is similar, only switching object and subject. Although the indexes are designed for generic RDF statements, they are adequate in our setup, as the ontological axioms manipulated by OntoDriver have the form of atomic class assertions, or atomic property assertions, both being serialized as single RDF triples. The PSO and POS indexes are also the default ones used by GraphDB [22] and Stardog [23].

Analysing Complexity of Typical Operations. In the following paragraphs we will examine time complexity of each of the operations enumerated at the beginning of this section with regards to the selected storages, with a short comment on possible implementations of these operations in OntoDriver.

Table 1. Asymptotic time complexity of the selected operations for GraphDB and Stardog. b is branching factor of the index B+ tree, n is the size of the dataset. The complexity of processing B+ trees is described in [20]. C_R is the reasoning cost, which depends on the selected language expressiveness and m is the number of reasoning cycles performed in materialization of statements inserted into GraphDB.

Storage	T_{find}	$T_{persist}$	T_{remove}
GraphDB	$O(log_b n)$	$O(\sum_{i=0}^{m} C_{Ri} \times log_b n)$	$O(\sum_{i=0}^{m} C_{Ri} \times log_b n)$
Stardog	$O(C_R) + O(log_b n)$	$O(log_b n)$	$O(log_b n)$

$find(\mathcal{M}, \delta_a)$ Multiple strategies can be employed to realize the *find* operation, but in essence they all perform a search for property assertion axioms where the

[10] The post is available at http://tinyurl.com/ke4ozf7, accessed 25-01-2015.

individual and property are bound. Therefore, the PSO index will be triggered. However, while GraphDB will proceed directly to finding the corresponding data, Stardog must first perform reasoning and rewrite the query according to the schema semantics. The complexity can be seen in Table 1.

We would like to stress here that the *find* operation is theoretically very favourable in terms of possible performance, because it does not require any joins, as it is supposed to return a simple union of property values for a single individual. Therefore it is straightforwardly mappable to the PSO index.

$persist(\mathcal{M}, \delta_v)$. Persisting assertion values specified in δ_v requires insertion of the corresponding statements into the storage's indexes (in our case the PSO and POS indexes).

In addition, in GraphDB, a materialization of statements inferred from the inserted knowledge is performed. Thus, from a set of statements K_E, inserted into the database, a new set K_I^0 of statements is derived, K_I^0 being in turn inserted into the dataset, triggering more materialization, until a set K_I^m is inserted, from which no additional knowledge can be deduced. This, of course, makes the *persist* operation in GraphDB more complex than in Stardog. Again, the theoretical complexity is shown in Table 1.

$remove(\mathcal{M}, \delta_a)$. Doing *remove* in an ontology requires knowledge of what exactly should be removed. Thus, JOPA performs *epistemic remove*, i.e. only values of properties mapped in the object model are removed. Therefore if the dataset contains property values which are not mapped by the object model managed by JOPA, these values are retained. In case the entity contains a field gathering unmapped asserted properties (see Sect. 3.1), the unmapped values are contained in this attribute and, to be consistent with the epistemic remove, JOPA deletes all statements where the removed individual is the subject.

remove in Stardog is again relatively straightforward. Since JOPA does allow only removal of explicit statements, there is no reasoning required. The procedure thus consists of finding the relevant statements and removing them from the index.

The situation is more interesting in GraphDB, because with the removal of explicit statements, some inferred knowledge may become irrelevant. GraphDB resolves the operation with a combination of forward and backward chaining [17]. In short, all possible inferred data is found from the removed statements first (this is the forward chaining part). From the results, backward chaining is performed to determine whether the implicit knowledge is backed by explicit knowledge other than that being removed. If not, the inferred statements are removed as well. Asymptotically, the complexity of *remove* in GraphDB is the same as *persist*, see Table 1.

The asymptotic complexities suggest that GraphDB is more suitable for read-oriented applications, especially when the expressiveness of reasoning increases. In these cases the cost of inference in GraphDB is paid when the dataset is loaded and at the actual runtime the queries will presumably be much faster. On the other hand, applications performing a lot of data modifications can benefit from the non-materializing approach of Stardog.

4.2 Experimental Complexity Evaluation

The theoretical complexity analysis provides information about asymptotic behaviour of the operations in the selected storages. However, there are hidden constants not visible in the formulas which may play a significant role for working with real-world data volumes. These hidden constants are mostly connected with the internal implementation of the storages. Hence, we decided to verify our conclusions with measurement.

Existing Storage Benchmarks. There exist several benchmarks for ontology storages. The key differences between them is expressiveness of their ontology and the queries. The Berlin SPARQL Benchmark (BSBM) [24] is purely RDF-oriented and used for testing of SPARQL endpoints. The well known Lehigh University Benchmark (LUBM) [25] contains basic OWL constructs, however, its schema is still missing more expressive features like nominals or number restrictions. The University Ontology Benchmark (UOBM) [26] is built upon LUBM, but adds more expressiveness, including transitive and equivalent properties or cardinality restrictions.

However, the queries used in the aforementioned benchmarks are general-purpose queries and they are not suitable for the object-oriented access scenario, which we primarily want to explore. Indeed, the operations required by JOPA (or any other OOM-supporting framework) are, as the reader already knows from Sect. 3.2, relatively straightforward and are focused on working with properties of a single individual. Therefore, we decided to create our own benchmark.

Object-UOBM. The benchmark we created exploits the existing schema and dataset generator of the UOBM benchmark. But instead of the generic queries of UOBM, it contains a set of eight queries tailored to the application access scenario. The queries are written in SPARQL and SPARQL Update in order to be interoperable for different storages, but they capture the essence of operations defined in the OntoDriver API.

We used this benchmark to evaluate performance of GraphDB and Stardog storages in order to verify our theoretical results.

Object-UOBM Results. Complete results of the experiments we conducted and a detailed description of Object-UOBM can be found in [27], we give just a brief overview here. The first observation that can be made from the results is that the price of real-time reasoning in Stardog is very high. The performance of a query representing the *find* operation with reasoning in Stardog is less than one query per second, while without reasoning it is able to answer nearly one thousand queries per second (see Fig. 3). A more surprising result is that SPARQL Update queries representing the *persist* and *update* were performing better in GraphDB than Stardog, although GraphDB has to perform materialization on insertion. Only for a DELETE query (representing the *remove*

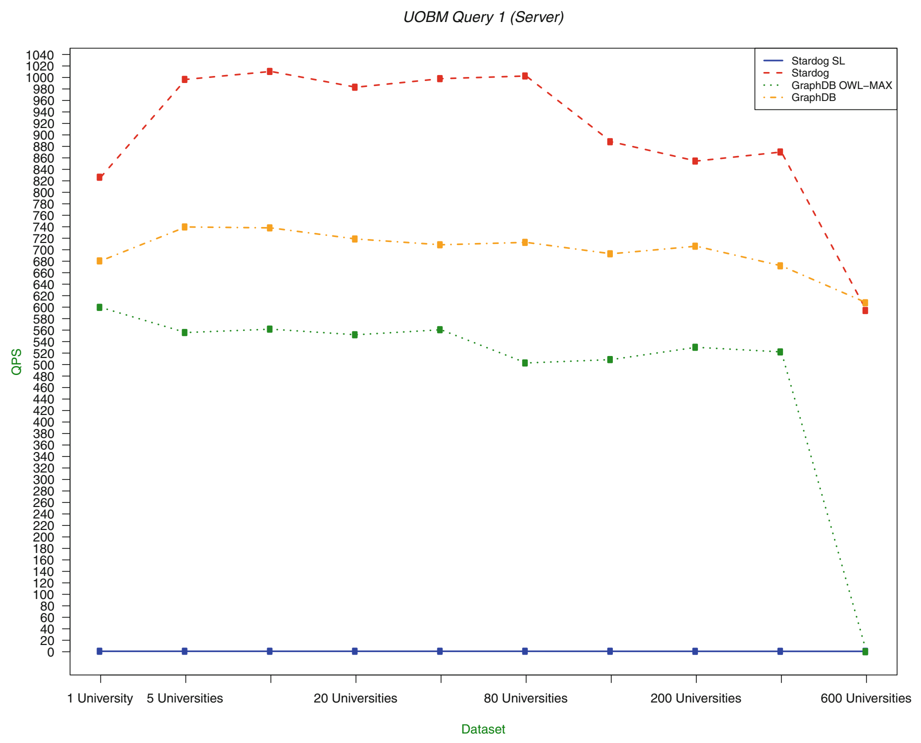


Fig. 3. Performance of Stardog and GraphDB for a SELECT query representing the *find* operation. Both storages were evaluated with reasoning (the Stardog SL and GraphDB OWL-Max lines, see [27] for explanation of the reasoning levels) and without it. GraphDB failed to answer the query for the largest dataset when using reasoning. One can cleary see the performance penalty of real-time reasoning in Stardog.

operation) the combination of forward and backward chaining made GraphDB slower than Stardog.

Another interesting result of the benchmark is that when loading an entity, it appears to be more efficient to filter the statements only by subject without specifying the properties to load. We had three variations of the same loading query, the first using unbound property and value, the second using a UNION of triple patterns specifying properties to load and the third doing the same, but using OPTIONAL. While the query using OPTIONAL performed decidedly worst, the query with unbound property performed the best, although it was loading unnecessary values of properties which were not mapped in the object model. Despite the fact that this situation could change in cases where the object model would map only a small portion of properties and values related to an individual, it seems to be more reasonable to use the unbound property strategy rather than specifying the properties to load explicitly.

Overall, it appears that GraphDB is more suitable for application access than Stardog. One has only to accept the fact that bulk loading is significantly faster in Stardog and that he has to specify repository expressiveness when creating it. The application-specific performance of GraphDB, which consists of querying and modifying existing data and inserting relatively small portions of new data, is better than that of Stardog.

5 Practical Evaluation of JOPA and OntoDriver

In this section we first evaluate the performance of OntoDriver when compared to a low-level approach represented by the Sesame API. Then we discuss our experience from developing a real-life IT system using JOPA.

5.1 Performance and Code Metrics Evaluation

In this section we briefly evaluate the performance of OntoDriver and compare it to approaches which directly use the native API of the underlying storage.

Performance of JOPA with OntoDriver. While we have already examined the theoretical complexity of operations present in the OntoDriver API and experimentally verified them on existing storages, we also need to validate that our implementation is efficient. The goal of this evaluation is to determine performance differences between ontology access using JOPA and OntoDriver and using Sesame API directly. Since the OntoDriver prototype internally uses Sesame API, we hardly expect JOPA to outperform pure Sesame API solution, instead we will concentrate on the possible performance penalties stemming from the additional logic that JOPA has to do. The test machine setup is as follows:

- Linux Mint 17 (64-bit)
- Java 8 update 31 (HotSpot), -Xms6g -Xmx6g
- Sesame API 2.7.14, GraphDB 6.0 RC6
- Intel i5 2.67 GHz
- 8 GB RAM

A class diagram of the benchmark schema is shown in Fig. 4. The application model is rather small, but sufficient to exercise a large part of the features supported by JOPA. The application model and the datasets are based on the UOBM benchmark [26], which we already used in our storage benchmark [27], and the datasets were generated using a generator application [28].

Results of the benchmark are shown in Table 2. The *find* operation was loading approximately 450 instances of `UndergraduateStudent`. Each of them was connected to three courses in average. Thus, the total number of loaded individuals with properties was more than 1800, representing over 5000 statements. The *persist* test inserted 500 new instances of `UndergraduateStudent`, connected to four existing courses and a new paper, into the ontology. The *update* evaluation updated the name and telephone of each of the previously persisted student, removed reference to one of his courses and added another one instead. Finally, the *remove* benchmark removed the 500 persisted undergraduate students.

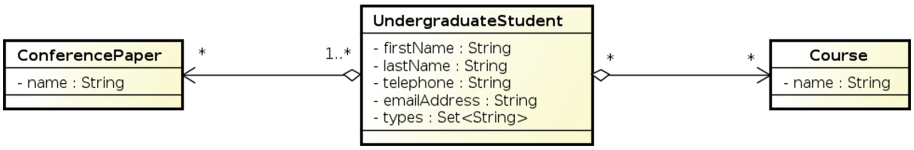


Fig. 4. Benchmark application model. Although small in size, it exercises most of the concepts supported by JOPA, including inferred entity types and data and object properties with lazy loading.

Table 2. Benchmark results. The times are average from 100 runs of the benchmark.

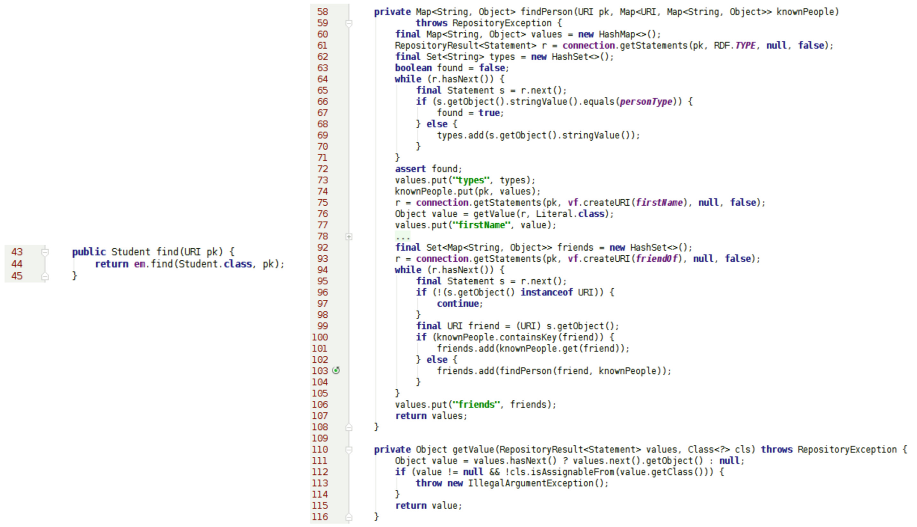
Dataset	$T_{persist}/s$		T_{find}/s		T_{update}/s		T_{remove}/s	
	JOPA	Sesame	JOPA	Sesame	JOPA	Sesame	JOPA	Sesame
UOBM 1	4.158	2.209	13.738	13.353	32.28	9.571	36.456	2.740
UOBM 5	4.245	2.252	13.830	13.366	32.461	9.993	36.718	2.918
UOBM 10	4.255	2.260	13.840	13.293	32.625	10.077	36.433	3.024

Benchmark Results Discussion. The benchmark results show that JOPA performs comparably when loading and persisting entities. It is important to point out that to mimic the behaviour of JOPA on entity loading, the Sesame API runner was verifying that the object property values were of the correct type. However, there is a significant performance gap between Sesame API and JOPA in *update* and *remove*. Major part in this gap is given by the fact that JOPA first has to load the entities before updating or removing them. For Sesame API, we simply removed (and inserted) the required statements without loading them first. Of course, the benchmark is skewed in this regard, because a real world application would most likely require the entity loading anyway. Also, JOPA currently does not support the `getReference` method [11], which would be suitable for the *update* and *remove* scenarios. Still, there is a large margin for improvement in JOPA for these operations.

In the future, we would like to try comparing different strategies of implementing OntoDriver.

JOPA and OntoDriver versus Sesame API. One of the most important advantages of using JOPA with OntoDriver is the ability to treat ontological individuals with their properties as coherent objects with attributes and possibly add behaviour to those objects, thus increasing readability and maintainability of the application. Such task cannot be accomplished using domain-independent APIs like Sesame API or OWL API without writing a large amount of boilerplate code. This difference is very similar to what the developer gains when using JPA instead of pure JDBC. Consider the example in Fig. 5. The difference in the amount of code written is clear, and the Sesame code does not even make any checks for correct types (for instance that a property value is another individual and not a literal) or integrity constraints.

JOPA and OntoDriver versus Domain-Specific Frameworks. The basic idea of JOPA and OntoDriver and domain-specific solutions like Empire or AliBaba is very similar – enable programmers to work with ontological data in object-oriented fashion. However, JOPA adds to this basic concept features which enable the user to exploit the nature of ontologies to more extent. JOPA supports working with unmapped properties, types and explicit distinction between inferred and asserted knowledge. In addition, the OntoDriver and its API enables JOPA to supports a wide range of ontological storages. AliBaba, on the other hand, is tied to storages supporting the Sesame API. Empire does have support for custom storage connectors, which are used by the framework via dependency injection. JOPA also offers better isolation of transactions and more advanced caching.



```java
public Student find(URI pk) {
    return em.find(Student.class, pk);
}
```

```java
private Map<String, Object> findPerson(URI pk, Map<URI, Map<String, Object>> knownPeople)
        throws RepositoryException {
    final Map<String, Object> values = new HashMap<>();
    RepositoryResult<Statement> r = connection.getStatements(pk, RDF.TYPE, null, false);
    final Set<String> types = new HashSet<>();
    boolean found = false;
    while (r.hasNext()) {
        final Statement s = r.next();
        if (s.getObject().stringValue().equals(personType)) {
            found = true;
        } else {
            types.add(s.getObject().stringValue());
        }
    }
    assert found;
    values.put("types", types);
    knownPeople.put(pk, values);
    r = connection.getStatements(pk, vf.createURI(firstName), null, false);
    Object value = getValue(r, Literal.class);
    values.put("firstName", value);
    ...
    final Set<Map<String, Object>> friends = new HashSet<>();
    r = connection.getStatements(pk, vf.createURI(friendOf), null, false);
    while (r.hasNext()) {
        final Statement s = r.next();
        if (!(s.getObject() instanceof URI)) {
            continue;
        }
        final URI friend = (URI) s.getObject();
        if (knownPeople.containsKey(friend)) {
            friends.add(knownPeople.get(friend));
        } else {
            friends.add(findPerson(friend, knownPeople));
        }
    }
    values.put("friends", friends);
    return values;
}

private Object getValue(RepositoryResult<Statement> values, Class<?> cls) throws RepositoryException {
    Object value = values.hasNext() ? values.next().getObject() : null;
    if (value != null && !cls.isAssignableFrom(value.getClass())) {
        throw new IllegalArgumentException();
    }
    return value;
}
```

Fig. 5. Find an entity. On the left hand side using JOPA. Entity definition is omitted, but it corresponds to the one shown in Listing 1.1. On the right hand side using Sesame API. The difference in the amount of code is clear.

6 Experience Using JOPA and OntoDriver

We have had an opportunity to use JOPA in a real-world application. This application is used to create reports about safety occurrences in aviation and is developed as part of the INBAS project [11]. We will now briefly summarize our experience.

[11] http://www.inbas.cz, Accessed on 07-08-2015.

Positives. One of the main positives of using JOPA stems from the frame-based approach and has already been discussed in great detail – it is the fact that the application works with coherent and logically defined objects. The objects can be, in addition to having behaviour of their own, easily passed over system boundaries, so for example we use the same domain objects when communicating with a JavaScript-based front end via REST web services, where they are serialized into JSON and deserialized back.

The support for operation cascading further reduces the amount of code by automatically carrying out the operation over relationships marked for cascading.

Another benefit of using JOPA and OntoDriver is the clear separation of storage access. Thus, we are able to use fast in-memory storage in tests and during development and switch to server based GraphDB solution when running in production. The only change that has to be made is modifying repository URL in a configuration file.

Deficiencies. The code imprint could be further reduced by integration with application frameworks like Spring. With that integration, transactions could be marked declaratively using annotations, persistence context injected automatically by the container and capabilities like Spring repositories used to minimize the data layer code that has to be written.

A feature that is missing in JOPA is the support for bidirectional relationships. Unless the relationship is explicitly defined using an inverse object property, there is currently no way of specifying a backwards reference to another object.

However, the greatest deficiency of JOPA and other domain-specific frameworks like Empire or AliBaba is the lack of support for ontology concept subsumption. Ontologies rely heavily on class hierarchies and it is very cumbersome to work around the lack of their support in persistence frameworks. For instance, in our application, we have a concept called *RunwayIncursion*, which describes an event when an object intrudes on a runway. This intruding object can be of the following types: *Aircraft*, *Vehicle* or *Person*. In ontology, this can modelled using the following hierarchy:

$$\begin{gathered}\mathcal{T} = \{Aircraft \sqsubseteq AerodromeAgent, \\ Vehicle \sqsubseteq AerodromeAgent, \\ Person \sqsubseteq AerodromeAgent, \\ RunwayIncursion \equiv \forall hasIntruder.AerodromeAgent\}\end{gathered}$$

It would be very convenient to be able to map such hierarchy to the domain model shown in Fig. 6. However, due to the lack of support for concept subsumption, this cannot be achieved in JOPA and we have to model such structure using an entity with fields of types *Aircraft*, *Vehicle* or *Person*, where only one of the fields can be non-null.

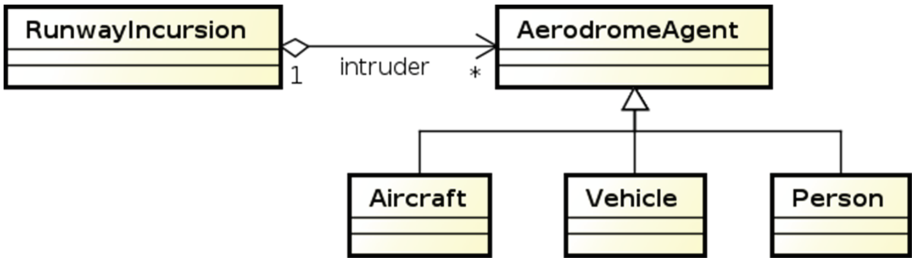


Fig. 6. Domain model, which would model an ontological concept hierarchy and range restriction.

6.1 Ontology Concept Subsumption

Let us now describe why it is difficult to represent concept subsumption in object-ontological mapping frameworks. Concept subsumption [29] is used to create hierarchies of classes[12] and can be used to model the *is a* relationship between the subsumed class and its subsumer (parent). Individuals of the subsumed class are also instances of the parent class, inheriting all of its properties. Unfortunately, such model cannot be straightforwardly transformed into the object-oriented paradigm. The *is a* relationship is realized through class *inheritance* in object-oriented languages. One problem with the Java language, in which most of the frameworks for working with ontologies are written, is that is supports only single-parent inheritance. However, ontological classes can be subsumed by multiple other classes.

A more crucial problem with ontological class hierarchies is the identity of the individuals. Consider the concept hierarchy described above and an individual *John* which belongs to the *Person* concept.

$$\mathcal{A} = \{Person(John)\}$$

Because *John* is an instance of *Person*, he is also an instance of *AerodromeAgent*. Now consider the object model in Fig. 6, which reflects the aforementioned concept hierarchy in object-oriented paradigm. If we load our individual *Adam* first as an *AerodromeAgent* and then as a *Person*, they will be two completely separate objects with the same IRI. However, it is still the same individual in the ontology.

This identity mismatch can lead to situations where for example the application changes the affiliation of the *AerodromeAgent* instance of *John* and at the same time changes the affiliation of the *Person* instance of *John*, but in the ontology, whichever modification comes first will be overwritten by the latter.

[12] *Class* is a term used in the OWL 2 language specification [4] and it corresponds to the term *concept*, used in description logics underlying the OWL language. We will use the terms interchangeably, unless a disambiguation between ontological classes and object-oriented paradigm classes is necessary.

As the astute reader may have noticed, the concept subsumption problem is not restricted to situations when the corresponding domain classes are related via inheritance. The same problem would occur if *AerodromeAgent* and *Person* were unrelated.

7 Conclusions

We have introduced JOPA as a solution for application access to ontologies, along with the OntoDriver, which separates the object-ontological mapping layer from the actual storage access, providing more opportunities for storage-specific optimizations and preventing vendor lock-in. We have examined the theoretical complexity of the operations defined in the OntoDriver API and tested two of the most advanced ontology storages for their suitability for application access.

We have also discussed our experience with JOPA as persistence provider in a real-world application, its benefits and deficiencies. We paid particular attention to the lack of support for class subsumption, which is a problem not specific to JOPA, but to all frameworks providing any form of object-ontological mapping.

In the future, we plan to thoroughly research the possibility of supporting some form of concept subsumption and domain class inheritance respectively in object-ontological mapping frameworks. We would also like to study possible optimizations of operations required by ontology-based applications.

Acknowledgements. This work was supported by the grant No. SGS13/204/OHK3/3T/13 Effective solving of engineering problems using semantic technologies of the Czech Technical University in Prague and No. TA04030465 Research and development of progressive methods for measuring aviation organizations safety performance of the Technology Agency of the Czech Republic.

References

1. Křemen, P., Kouba, Z.: Ontology-driven information system design. IEEE Trans. Syst. Man Cybern. Part C **42**, 334–344 (2012)
2. Ledvinka, M., Křemen, P.: JOPA: developing ontology-based information systems. In: Proceedings of the 13th Annual Conference Znalosti 2014 (2014)
3. Křemen, P.: Building ontology-based information systems. Ph.D. thesis, Czech Technical University, Prague (2012)
4. Motik, B., Parsia, B., Patel-Schneider, P.F.: OWL 2 web ontology language structural specification and functional-style syntax. In: W3C Recommendation, W3C (2009)
5. Meditskos, G., Bassiliades, N.: A rule-based object-oriented OWL reasoner. IEEE Trans. Knowl. Data Eng. **20**, 397–410 (2008)
6. Poggi, A.: Developing ontology based applications with O3L. WSEAS Trans. Comput. **8**(8) August 2009
7. Horridge, M., Bechhofer, S.: The OWL API: a Java API for OWL ontologies. In: Semantic Web - Interoperability, Usability, Applicability (2011)

8. Broekstra, J., Kampman, A., van Harmelen, F.: Sesame: a generic architecture for storing and querying RDF and RDF schema. In: Horrocks, I., Hendler, J. (eds.) ISWC 2002. LNCS, vol. 2342, pp. 54–68. Springer, Heidelberg (2002)
9. Carroll, J.J., Dickinson, I., Dollin, C., Reynolds, D., Seaborne, A., Wilkinson, K.: Jena: implementing the semantic web recommendations. In: Proceedings of the 13th International World Wide Web Conference (Alternate Track Papers & Posters), pp. 74–83 (2004)
10. Grove, M.: Empire: RDF & SPARQL Meet JPA. semanticweb.com (2010)
11. JCP: JSR 317: Java^{TM} Persistence API, Version 2.0 (2009)
12. Harris, S., Seaborne, A.: SPARQL 1.1 Query Language. Technical report, W3C (2013)
13. Gearon, P., Passant, A., Polleres, A.: SPARQL 1.1 Update. Technical report, W3C (2013)
14. Horrocks, I., Kutz, O., Sattler, U.: The even more irresistible SROIQ. In: Proceedings of the 10th International Conference on Principles of Knowledge Representation and Reasoning (KR 2006), pp. 57–67 (2006)
15. Tao, J., Sirin, E., Bao, J., McGuinness, D.L.: Integrity constraints in OWL. In: Fox, M., Poole, D. (eds.): AAAI. AAAI Press (2010)
16. Cyganiak, R., Wood, D., Lanthaler, M.: RDF 1.1 concepts and abstract syntax. Technical report, W3C (2014)
17. Bishop, B., Kiryakov, A., Ognyanoff, D., Peikov, I., Tashev, Z., Velkov, R.: OWLIM: a family of scalable semantic repositories. In: Semantic Web - Interoperability, Usability, Applicability (2010)
18. Erling, O.: Virtuoso, a hybrid RDBMS/graph column store. IEEE Data Eng. Bull. **35**, 3–8 (2012)
19. Sirin, E., Parsia, B., Grau, B.C., Kalyanpur, A., Katz, Y.: Pellet: a practical OWL-DL reasoner. Web Semant. Sci. Serv. Agents World Wide Web **5**, 51–53 (2007)
20. Comer, D.: The Ubiquitous B-Tree. Comput. Surv. **11**, 121–137 (1979)
21. Hepp, M., de Leenheer, P., de Moor, A., Sure, Y.: Ontology Management: Semantic Web, Semantic Web Services, and Business Applications. Springer, New York (2007)
22. Ontotext: GraphDB-SE-GraphDB6-Ontotext Wiki (2014) http://owlim.ontotext.com/display/GraphDB6/GraphDB-SE+Indexing+Specifics
23. Stardog: Stardog Docs (2014). http://docs.stardog.com/
24. Bizer, C., Schultz, A.: The Berlin SPARQL benchmark. Int. J. Seman. Web Inf. Syst. **5**(2), 1–24 (2009)
25. Guo, Y., Pan, Z., Heflin, J.: LUBM: A benchmark for OWL knowledge base systems. J. Web Semant. **3**, 158–182 (2005)
26. Qiu, Z., Liu, S., Pan, Y., Ma, L., Xie, G.T., Yang, Y.: Towards a complete OWL ontology benchmark. In: Sure, Y., Domingue, J. (eds.) ESWC 2006. LNCS, vol. 4011, pp. 125–139. Springer, Heidelberg (2006)
27. Ledvinka, M., Křemen, P.: Object-UOBM: an ontological benchmark for object-oriented access. In: Klinov, P., Mouromtsev, D. (eds.) KESW 2015. CCIS, vol. 518, pp. 132–146. Springer, Heidelberg (2015)
28. Zhou, Y., Grau, B.C., Horrocks, I., Wu, Z., Banerjee, J.: Making the most of your triple store: query answering in OWL 2 using an RL reasoner. In: Proceedings of the 22nd International Conference on World Wide Web (2013)
29. Baader, F., Calvanese, D., McGuinness, D.L., Nardi, D., Patel-Schneider, P.F. (eds.): The Description Logic Handbook: Theory, Implementation, and Applications. Cambridge University Press, New York (2003)