JOPA: Accessing Ontologies in an Object-oriented Way

Martin Ledvinka and Petr Křemen
Czech Technical University in Prague, Technická 2, Praha, Czech Republic

Keywords: Ontology, Persistence, Application Access, Complexity.

Abstract: Accessing OWL ontologies programmatically by complex IT systems brings many problems stemming from ontology evolution, their open-world nature and expressiveness. This paper presents Java OWL Persistence API (JOPA), 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, advanced caching, transactional approach, 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. Last but not least, we compare our object-oriented solution to low level Sesame API in terms of efficiency.

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 (Křemen and Kouba, 2012).

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 (Ledvinka and Křemen, 2014), 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 ((Křemen and Kouba, 2012) and (Křemen, 2012)), 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. Lastly, we compare our solution to low level Sesame API in terms of efficiency.

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[1] for storage access. The paper is concluded in Section 5.

2 RELATED WORK

Some object-oriented solutions try to approximate ontological OWL reasoning (Motik et al., 2009)

[1]Application Programming Interface

by means of procedural code, like (Meditskos and Bassiliades, 2008), or (Poggi, 2009). 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 (Horridge and Bechhofer, 2011), Sesame (Broekstra et al., 2002) or Jena (Carroll et al., 2004). 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 Section 4.2.2.

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 OOM[2]. Representatives of this paradigm are e.g. Empire (Grove, 2010) or AliBaba[3]. 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. On the other hand, these solutions bury most of the ontological expressiveness. Comparing to these systems, our solution allows to monitor ontology changes during its evolution, access both asserted and inferred knowledge and provide individual types as well as extra properties through additional Java annotations.

A thorough discussion of these architectures can be found in (Ledvinka and Křemen, 2014) and in (Křemen, 2012). JOPA, introduced in Section 3 aims at taking the best of both types, as shown in Figure 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 Section 3).

[2]Object-ontological Mapping

[3]http://rdf4j.org/, accessed 02-12-2014

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 (JCP, 2009).

In this section, we introduce the layered architecture of JOPA.

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 (JCP, 2009), 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.

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 (JCP, 2009) to a large extent. It contains CRUD[4] operations: *find*, *persist*, *merge* and *remove*, but it enhances them with versions supporting context descriptors (Ledvinka and Křemen, 2014). It

[4]Create, Retrieve, Update, Delete

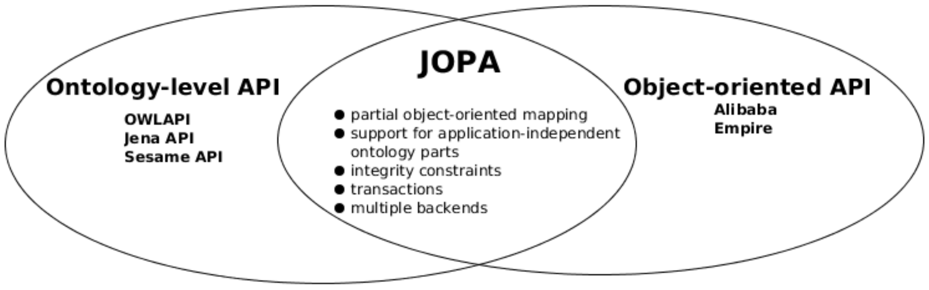

Figure 1: JOPA and related approaches.

also contains operations for transaction management and cache access.

Mapping. As was stated earlier, the mapping between ontological data and the application domain model is based on a contract expressed using integrity constraints. In practice, these integrity constraints are compiled into a set of Java annotations that guard their semantics on the object level. This allows easy validation of cardinality constraints, as well as domains and ranges of annotation/data or object properties. In this respect, JOPA is very similar to Empire(Grove, 2010). An example of a JOPA entity class can be seen in Listing 1. The `@Types` field contains information about ontology classes to which individual represented by an instance of this type belongs.

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

Listing 1: Example of a business entity class declaration with JOPA annotations representing the object-ontological mapping.

It is necessary to point out that the identity of a business object in JOPA is given not only by the individual's IRI[5], but also by the ontology class mapped by its type, i.e. in the underlying ontology the individual must be explicitly stated to be of the specified ontology class. It should also be noted that JOPA does not support blank nodes and anonymous individuals. JOPA does not support class subsumption either, due to the lack of clear relationship between ontological and object identity, as well as the lack of multiple inheritance in many object-oriented languages, like Java. To facilitate working with entity relationships, JOPA does support operation cascading.

Unmapped Properties. In addition to the fixed set of modelled properties, JOPA enables the application to access also the properties which are not part of the object model. The property values are currently restricted to their string representation and 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.

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

[5]Internationalized Resource Identifier

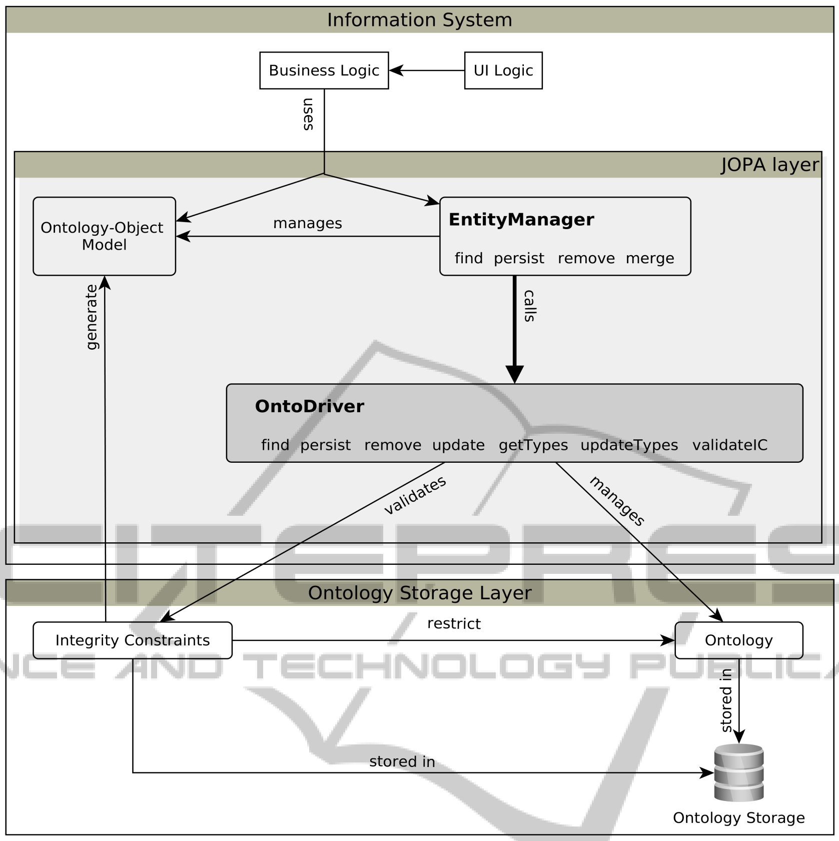

Figure 2: JOPA Architecture. A clear separation between JOPA and the underlying ontology store layer can be seen. Integrity constraints are used to represent the contract between the ontology and the application object model.

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.

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* (JCP, 2009), which is shared between all open persistence contexts and enables quick entity lookup. Another performance improving feature is the support for lazily loaded attributes[6].

3.2 OntoDriver

OntoDriver is a software layer designed to unify access to various ontology storages. It achieves this goal by presenting a single API to JOPA and enabling the implementation to use whatever framework is required by the underlying storage, e.g. Sesame API for Sesame storage or Jena for SDB. In this regard, OntoDriver is similar to a JDBC[7] driver known from

[6]Lazily loaded attribute values are retrieved from the data source only upon application request.

[7]Java Database Connectivity

the relational world. But in contrast to JDBC, where all operations are done using SQL[8] 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 (Harris and Seaborne, 2013) queries for information retrieval and SPARQL Update (Gearon et al., 2013) statements for data manipulation.

3.2.1 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 $\mathcal{SROIQ}(\mathcal{D})$[9]. In the next sections, consider an OWL 2 DL ontology $O = (\mathcal{T}, \mathcal{A})$, consisting of a TBox $\mathcal{T} = \{\tau_l\}$ and an ABox $\mathcal{A} = \{\alpha_l\}$, where α_l 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 class*, $P \in N_r$ 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 (Motik et al., 2009) and $\mathcal{SROIQ}(\mathcal{D})$ (Horrocks et al., 2006).

In addition to ontological (open-world) knowledge, a *set* $\mathcal{S}_C = \{\gamma_i\}$ *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 (Tao et al., 2010).

By *multi-context ontology* we denote a tuple $\mathcal{M} = (O_d, O_1, ..., O_n)$, where each O_l is an ontology identified by a unique IRI and is called *context*, 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 (Cyganiak et al., 2014). An *ontology store* is a software layer that provides access to $\mathcal{M}$.

An *axiom descriptor* δ_a is a tuple $(i, \{(r_1, b_1)...(r_k, b_k)\})$, where $i \in N_i$, $r_m \in N_r$, $b_m \in \{0,1\}$ and $m \in 1...k$. 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 $(i, \{(r_1, v_1)...(r_k, v_k)\})$, where $i \in N_i$, $r_m \in N_r$, $v_m \in N_i$ and $m \in 1...k$. 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,
- $persist(\mathcal{M}, \delta_v) = O_d \cup \{\alpha_1...\alpha_s\}$, where $\alpha_1...\alpha_s$ are property assertion axioms created from role-value pairs in δ_v,
  - Persist axioms representing entity attribute values,
  - Used by `EntityManager.persist()` in OOM,
- $remove(\mathcal{M}, \delta_a) = O_d \setminus \{\alpha'_1...\alpha'_t\}$, where $\alpha'_1...\alpha'_t$ are property assertion axioms for the roles specified in δ_a,
  - Remove axioms representing entity attribute values,
  - Used by `EntityManager.remove()` in OOM,
- $update(\mathcal{M}, \delta_v) = (O_d \setminus \{\alpha'_1...\alpha'_t\}) \cup \{\alpha_1...\alpha_s\}$, where $\alpha'_1...\alpha'_t$ are original property assertion axioms for the roles $r_1...r_k$ defined in δ_v and $\alpha_1...\alpha_s$ 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,

[8]Standard Query Language

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

- $updateTypes(\mathcal{M}, i, \{c_1...c_k\}) = (O_d \setminus \{\alpha'_1...\alpha'_t\}) \cup \{\alpha_1...\alpha_k\}$, 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,
- $validateIC(\mathcal{M}, \{\gamma_1...\gamma_k\}) : 2^{\mathcal{M}} \times 2^{N_i \times N_r \times N_i} \times \mathcal{S}_c \to \{0, 1\}$, where $\gamma_m \in \mathcal{S}_c$ and $m \in 1...k$,
  - Validate the specified integrity constraints, verifying *reasoning-time* integrity constraints which cannot be validated at runtime (Křemen and Kouba, 2012),
  - 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.

3.2.2 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 (Bishop et al., 2010)) and Virtuoso (Erling, 2012). 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 (Broekstra et al., 2002) 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. We expect the performance of asking for the property values one by one using the Sesame filtering API to be similar to using a single SPARQL query with union on triple patterns, because eventually both strategies lead to evaluation of multiple triple patterns with the subject and property bound and no joins. Nevertheless, 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. However, we plan to investigate the possibility of using SPARQL instead of the Sesame filtering API as well.

Another important point is how the Sesame OntoDriver deals with transactions. As was mentioned in Section 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 (Sirin et al., 2007), 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 look at the prototypical implementation of OntoDriver described in Section 3.2.2 and measure the operation complexity on some real data.

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 (Bishop et al., 2010), GraphDB is a Sesame SAIL[10] with rule-based reasoner using forward chaining,

Stardog,[11] 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.

4.1.1 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 (Comer, 1979). GraphDB uses a modified version of B-trees – a B+ tree (Hepp et al., 2007). 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[12].

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 (Ontotext, 2014) and Stardog (Stardog, 2014).

4.1.2 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.

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

[10]Storage And Inference Layer

[11]http://www.stardog.com, accessed 02-12-2014

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

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 (Comer, 1979). 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	$O(C_R) + O(log_b n)$	$O(log_b n)$	$O(log_b n)$

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 Section 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 (Bishop et al., 2010). 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 will benefit from the non-materializing approach of Stardog.

Of course, these are only theoretical results which consider asymptotic complexities and disregard possible hidden constants. We are currently working on a benchmark which would validate these expectations.

4.2 OntoDriver in Practice

In this section we briefly evaluate the benefits of OntoDriver from two important perspectives – performance and code metrics.

4.2.1 Performance of JOPA with OntoDriver

After examining theoretical complexity of the OntoDriver API, we can proceed to measuring performance of our OntoDriver prototype. 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 JOPA has to do. As an extra result, we will discuss the real performance w.r.t. our theoretical approximation of the asymptotic complexity of GraphDB. The test machine setup is following:

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

A class diagram of the benchmark schema is shown in Figure 3. The application model is rather small, but sufficient to exercise most of the features supported by JOPA. The application model and the datasets are based on the UOBM benchmark (Ma et al., 2006), the datasets were generated using a generator application (Zhou et al., 2013).

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,

Table 2: Benchmark results. The times are average from 100 runs of the benchmark. T_{load} represents the amount of time it took GraphDB to load the respective datasets.

Dataset	T_{load}/s	$T_{persist}/s$		T_{find}/s		T_{update}/s		T_{remove}/s	
		JOPA	Sesame	JOPA	Sesame	JOPA	Sesame	JOPA	Sesame
UOBM 1	38	4.158	2.209	13.738	13.353	32.28	9.571	36.456	2.740
UOBM 5	213	4.245	2.252	13.830	13.366	32.461	9.993	36.718	2.918
UOBM 10	424	4.255	2.260	13.840	13.293	32.625	10.077	36.433	3.024

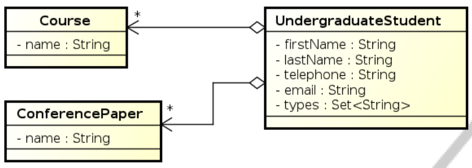

Figure 3: Benchmark application model. Although small in size, it exercises most of the concepts supported by JOPA, including inferred entity types and data and object properties with lazy loading.

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.

Benchmark Results Discussion. The benchmark results show that JOPA performs comparably when loading or 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 (JCP, 2009), which would be suitable for the *update* and *remove* scenarios. Still, there is a large margin for improvement in JOPA for these operations.

Table 2 also shows that dataset loading times grow linearly for GraphDB. Other times do not show any particular trends. The sample is probably to small to be compared to our theoretical findings.

In the future, we would like to try comparing different strategies of implementing OntoDriver. We are also currently working on a performance comparison of Stardog and GraphDB, using a set of SPARQL queries specifically designed to correspond to the operations defined in the OntoDriver API.

4.2.2 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 cohesive objects with attributes and possibly add behaviour to those objects, thus increasing readability and maintainability of the application. This cannot be achieved using plain Sesame API without writing a lot of wrapper code. This difference is very similar to what the developer gains when using JPA instead of pure JDBC. Due to the lack of space we cannot show a code example using the two approaches here, but we believe that readers familiar with Sesame API, OWL API or Jena know how tedious and clumsy code using these frameworks can be.

To provide at least a short insight into the difference of the amount of code that needs to be written for basic ontological data manipulation using JOPA and Sesame API, Table 3 compares the number of lines of code of classes used to perform the benchmark described in the previous section. The entity classes are not included in the comparison, because they are reusable objects not specific to the runners.

Table 3: Benchmark runner code length. *LOC* represents the number of lines of code in each of the runner classes. The benchmark project is available at https://krizik.felk.cvut.cz/kbss/JopaDemo/, the runner classes are `JopaRunner` and `SesameRunner`.

Framework	*LOC*
JOPA	123
Sesame API	296

We believe that even on such a small example we have demonstrated the benefits of using JOPA with OntoDriver against low level APIs like Sesame.

5 CONCLUSIONS

We have introduced JOPA as a solution for object-oriented access to ontologies. We have described the architecture of JOPA and its prototypical implementation, including the OntoDriver component with its API and implementation using Sesame API. OntoDriver represents a software layer which separates object-ontological mapping in JOPA from access to different storages, providing a general and concise API. Based on this API, we have examined complexity of its operations both theoretically with regards to two well-known storages and practically in a benchmark of our prototype.

In the future, we intend to work on implementations of OntoDriver for other storages and a more thorough benchmark of these storages in cooperation with the corresponding drivers.

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.

REFERENCES

Bishop, B., Kiryakov, A., Ognyanoff, D., Peikov, I., Tashev, Z., and Velkov, R. (2010). OWLIM: A family of scalable semantic repositories. *Semantic Web – Interoperability, Usability, Applicability*.

Broekstra, J., Kampman, A., and van Harmelen, F. (2002). Sesame: A Generic Architecture for Storing and Querying RDF and RDF Schema. In *Proceedings of the First International Semantic Web Conference on The Semantic Web*, pages 54–68.

Carroll, J. J., Dickinson, I., Dollin, C., Reynolds, D., Seaborne, A., and Wilkinson, K. (2004). Jena: implementing the semantic web recommendations. In *Proceedings of the 13th international World Wide Web conference (Alternate Track Papers & Posters)*, pages 74–83.

Comer, D. (1979). The Ubiquitous B-Tree. *Computing Surveys*, 11(2).

Cyganiak, R., Wood, D., and Lanthaler, M. (2014). RDF 1.1 Concepts and Abstract Syntax. Technical report, W3C.

Erling, O. (2012). Virtuoso, a Hybrid RDBMS/Graph Column Store. *IEEE Data Engineering Bulletin*, 35(1):3–8.

Gearon, P., Passant, A., and Polleres, A. (2013). SPARQL 1.1 Update. Technical report, W3C.

Grove, M. (2010). Empire: RDF & SPARQL Meet JPA. *semanticweb.com*.

Harris, S. and Seaborne, A. (2013). SPARQL 1.1 Query Language. Technical report, W3C.

Hepp, M., de Leenheer, P., de de Moor, A., and Sure, Y. (2007). *Ontology Management: Semantic Web, Semantic Web Services, and Business Applications*. Springer.

Horridge, M. and Bechhofer, S. (2011). The OWL API: A Java API for OWL ontologies. *Semantic Web – Interoperability, Usability, Applicability*.

Horrocks, I., Kutz, O., and Sattler, U. (2006). The even more irresistible SROIQ. In *Proc. of the 10th Int. Conf. on Principles of Knowledge Representation and Reasoning (KR 2006)*, pages 57–67.

JCP (2009). JSR 317: Java^{TM} Persistence API, Version 2.0.

Křemen, P. (2012). *Building Ontology-Based Information Systems*. PhD thesis, Czech Technical University, Prague.

Křemen, P. and Kouba, Z. (2012). Ontology-Driven Information System Design. *IEEE Transactions on Systems, Man, and Cybernetics: Part C*, 42(3):334–344.

Ledvinka, M. and Křemen, P. (2014). JOPA: Developing Ontology-Based Information Systems. In *Proceedings of the 13th Annual Conference Znalosti 2014*.

Ma, L., Yang, Y., Qiu, Z., Xie, G., Pan, Y., and Liu, S. (2006). Towards a Complete OWL Ontology Benchmark. In *ESWC'06 Proceedings of the 3rd European conference on The Semantic Web: research and applications*, pages 125–139.

Meditskos, G. and Bassiliades, N. (2008). A Rule-Based Object-Oriented OWL Reasoner. *IEEE Transactions on Knowledge and Data Engineering*, 20(3):397–410.

Motik, B., Parsia, B., and Patel-Schneider, P. F. (2009). OWL 2 Web Ontology Language Structural Specification and Functional-Style Syntax. W3C recommendation, W3C.

Ontotext (2014). GraphDB-SE–GraphDB6–Ontotext Wiki. http://owlim.ontotext.com/display/GraphDB6/GraphDB-SE+Indexing+Specifics.

Poggi, A. (2009). Developing Ontology Based Applications with O3L. *WSEAS Trans. on Computers*.

Sirin, E., Parsia, B., Grau, B. C., Kalyanpur, A., and Katz, Y. (2007). Pellet: A practical OWL-DL reasoner. *Web Semantics: Science, Services and Agents on the World Wide Web*, 5(2).

Stardog (2014). Stardog Docs. http://docs.stardog.com/.

Tao, J., Sirin, E., Bao, J., and McGuinness, D. L. (2010). Integrity Constraints in OWL. In Fox, M. and Poole, D., editors, *AAAI*. AAAI Press.

Zhou, Y., Grau, B. C., Horrocks, I., Wu, Z., and Banerjee, J. (2013). 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*.