Semantic Web 0 (2018) 1–0
IOS Press

A comparison of object-triple mapping frameworks

Martin Ledvinka [a,*] and Petr Křemen [a]
[a] *Department of Cybernetics, Faculty of Electrical Engineering, Czech Technical University in Prague, Technická 2, 166 27 Prague 6 - Dejvice, Czech Republic*
E-mails: martin.ledvinka@fel.cvut.cz, petr.kremen@fel.cvut.cz

Abstract. Domain-independent information systems like ontology editors provide only limited usability for non-experts when domain-specific linked data need to be created. On the contrary, domain-specific applications require adequate architecture for data authoring and validation, typically using the object-oriented paradigm. So far, several frameworks mapping the RDF model (representing linked data) to the object model have been introduced in the literature. In this paper, we develop a novel framework for comparison of object-triple mapping solutions in terms of features and performance. For feature comparison, we designed a set of qualitative criteria reflecting object-oriented application developer's needs. For the performance comparison, we introduce a benchmark based on a real-world information system that we implemented using one of the compared OTM solutions – JOPA. We present a detailed evaluation of a selected set of object-triple mapping libraries and show that they significantly differ both in terms of features and time and memory performance.

Keywords: Object-triple Mapping, Object-ontological Mapping, Object-oriented Programming, RDF, Benchmark

1. Introduction

The idea of the *Semantic Web* [1] might not have seen such an explosive success as the original *Web*, but with technology leaders like Facebook[1], Google[2], IBM [2] or Microsoft[3], as well as emerging national linked data [3] platforms[4], adopting its principles, it seems it has finally taken root. Semantically meaningful data, which allow to *infer* implicit knowledge, described, stored and queried using standardized languages and interconnected via global identifiers can provide a huge benefit for application users on many levels.

1.1. Motivating scenario

Let's consider a fictive national civil aviation authority (CAA), which is a public governmental body, and is obliged to publish its data as open data. The CAA decides to develop a safety management system for the oversight of national aviation organizations, involving safety occurrence reporting, safety issues definition, and aviation organization performance dashboard. The CAA can then concentrate on suspicious patterns identified by the system (e.g., from frequently occurring events) during its inspections at the aviation organizations or during safety guidelines preparation. To establish data and knowledge sharing between the CAA and the aviation organizations, CAA decides to share the safety data as 5-star Linked Data [4]. To achieve this, the system is designed on top of a proper ontological conceptualization of the domain. This allows capturing safety occurrence reports with deep insight into the actual safety situation, allowing to model causal dependencies between safety occurrences, describe event participants, use rich typologies of occurrences, aircraft types, etc. Furthermore, integration of

*Corresponding author. E-mail: martin.ledvinka@fel.cvut.cz.
[1] http://ogp.me, accessed 2018-03-01
[2] https://tinyurl.com/g-knowledge-graph, accessed 2018-03-01
[3] https://tinyurl.com/ms-concept-graph, accessed 2018-03-01.
[4] e.g. the Czech linked open data cloud https://linked.opendata.cz, the Austrian one https://www.data.gv.at/linked-data (both accessed 2018-04-04).

other available Linked Data sources, e.g. a register of aviation companies (airlines, airports etc.) and aircraft helps to reveal problematic aviation organizations. A system with related functionality aimed at the spacecraft accident investigation domain was built at NASA by Carvalho et al. [5, 6].

To be usable by non-experts, the system is domain specific, not a generic ontology editor/browser. Design and development of such an application require efficient access to the underlying data. Using common semantic web libraries like Jena [7] or RDF4J [8] for the development of the application ends up with a large amount of boilerplate code, as these libraries work on the level of triples/statements/axioms and do not provide frame-based knowledge structures. The solution is to allow developers to use the well-known object-oriented paradigm and provide an *object-triple mapping* (OTM) that defines the contract between the application and the underlying knowledge structure. For this purpose, many frameworks appeared trying to offer such a contract. However, these frameworks differ a lot in their capabilities as well as efficiency, neither of which is systematically documented. In this survey paper, we provide a framework for comparing various OTM solutions in terms of features and performance. In addition, we use this framework and try to systematize the existing OTM approaches with the aim of helping developers choose the most suitable one for their use case.

1.2. Contribution

The fundamental question of this paper is: "Which object-triple mapping solution is suitable for creating and managing object-oriented applications backed by an RDF triple store?". The actual contribution of this paper can be split into three particular goals:

1. Select a set of *qualitative criteria* which can be used to compare object-triple mapping libraries for their use in object-oriented applications.
2. Design and develop a benchmark for performance comparison of object-triple mapping libraries. This benchmark should be easy-to-use and require as little effort as possible to accommodate a new library into the comparison.
3. Compare selected object-triple mapping frameworks in terms of their features and performance.

The remainder of this paper is structured as follows. Section 2 presents the necessary background on the RDF language, OTM systems, and Java performance benchmarks. Section 3 reviews existing approaches to comparing OTM libraries and benchmarking in the Semantic Web world in general. Section 4 presents the framework designed in this work. Section 5 introduces the OTM libraries chosen for the comparison and discusses the reasons for their selection. Then, Sections 6 and 7 actually compare the selected libraries using the framework designed in Section 4. The paper is concluded in Section 8. Appendices A.1 and A.2 contain full reports of time performance, resp. scalability comparison.

2. Background

The fundamental standard for the Semantic Web is the Resource Description Framework (RDF) [9]. It is a data modeling language built upon the notion of *statements* about *resources*. These statements consist of three parts, the *subject* of the description, the *predicate* describing the subject and the *object*, i.e., the predicate value. Such statements – *triples* – represent elements of a labeled directed graph, an *RDF graph*, where the nodes are resources (IRIs or blank nodes) or literal values (in the roles of subjects or objects) and the edges are properties (in the role of predicates) connecting them. RDF can be serialized in many formats, including RDF/XML, Turtle or N-triples. Taking an RDF graph and a set of *named graphs* (RDF graphs identified by IRIs), we get an *RDF dataset*.

Listing 1: Turtle serialization of an RDFS schema.

```
@prefix rdf:
 <http://www.w3.org/1999/02/22-rdf-syntax-ns#> .
@prefix rdfs:
 <http://www.w3.org/2000/01/rdf-schema#> .
@prefix xsd: <http://www.w3.org/2001/XMLSchema#> .
@prefix doc: <http://onto.fel.cvut.cz/ontologies/
     documentation/>.
@prefix as:
 <http://onto.fel.cvut.cz/ontologies/a-s/> .
@prefix ufo:
 <http://onto.fel.cvut.cz/ontologies/ufo/> .

doc:documents
   a rdf:Property ;
   rdfs:range as:occurrence ;
   rdfs:domain doc:occurrence_report .

doc:has_file_number
   a rdf:Property ;
   rdfs:range xsd:long .

as:occurrence
   a rdfs:Class ;
   rdfs:subClassOf ufo:event .

doc:occurrence_report a rdfs:Class .
ufo:event a rdfs:Class .
```

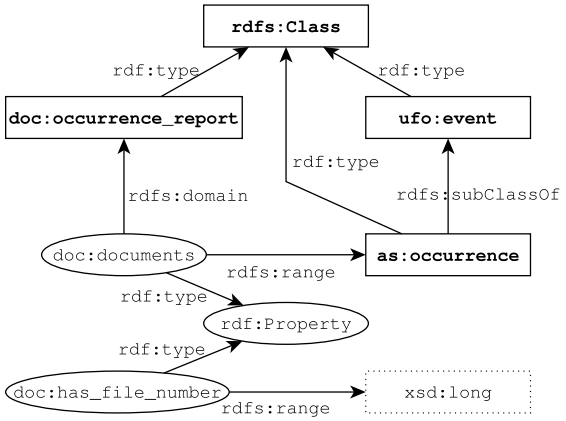


Fig. 1. RDF Graph representing a simple RDFS ontology.

The expressive power of RDF is relatively small. To be able to create an *ontology* (a formal, shared conceptualization) of a domain, more expressive languages like RDF Schema (RDFS) [10], OWL [11] and OWL 2 [12] were introduced. They extend RDF with constructs like classes, domains and ranges of properties, and class or property restrictions. A simplistic example of an RDFS ontological schema can be seen in Figure 1, its serialization in Turtle [13] is then displayed in Listing 1. An important feature of the Semantic Web is also that RDF is used to represent both the schema and the actual data.

The SPARQL Query Language (SPARQL) [14] and SPARQL Update [15] are standardized languages used to query and manipulate RDF data. A SPARQL query example is shown in Listing 2; it selects all classes and, if present, also their immediate super types.

Listing 2: A SPARQL query example.

```
PREFIX rdf:
  <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
PREFIX rdfs:
  <http://www.w3.org/2000/01/rdf-schema#>

SELECT DISTINCT ?cls ?supcls WHERE {
   ?cls a rdfs:Class .
   OPTIONAL {
      ?cls rdfs:subClassOf ?supcls .
   }
}
```

The primary focus of this paper (as well as most of the discussed OTM solutions) is on RDFS. However, OWL (2) is referenced in several places to provide further context.

2.1. Accessing semantic data

Ontological data are typically stored in dedicated semantic databases or *triple stores*, such as RDF4J [8], GraphDB [16] or Virtuoso [17]. To be able to make use of these data, an application needs means of accessing the underlying triple store. Unfortunately, there is no common standard for accessing a semantic database (like ODBC [18] for relational database access). This lack of standardization gave rise to multiple approaches and libraries that can be basically split into two categories (as discussed in [19]):

Domain-independent libraries like Jena [7], OWL API [20] or RDF4J (former Sesame API) [8]. Such libraries are suitable for generic tools like ontology editors or vocabulary explorers that do not make any assumptions about the ontology behind the data the application works with.

Domain-specific libraries, like AliBaba [21], Empire [22] or JOPA [23] are examples of libraries which employ some form of *object-triple* (OTM) (or *object-ontological* (OOM)) *mapping* to map semantic data to domain-specific objects, binding the application with some assumption about the form of data.

Domain-specific libraries are more suitable for creating object-oriented applications. Such applications express the domain model in the form of object classes, and their properties (attributes or relationships). OTM, in this case, allows to automatically map the notions from the object-oriented realm to the semantic data model and vice versa. Domain-independent libraries perform no such thing. They treat the underlying data at the same level as the RDF language, i.e. at the level of statements, without any assumption about their meaning. Therefore, a domain-specific application attempting to use such libraries would either need to treat its data in the same manner, which makes it extremely difficult to understand, develop and maintain; or the developer would end up writing his own version of an object-triple mapping library. Either way, it is more favorable to leverage dedicated object-triple mapping solutions. In addition, some OTM frameworks allow accessing various data sources in a unified way, support transactional processing and compile-time checking of correct model usage. Table 1 shows a simplified mapping considered by the majority of the OTM frameworks between RDF(S) terms and object model artifacts. We shall use these notions throughout this paper.

Table 1
Simplified mapping between RDF(S) terms and object model artifacts

Ontology	Object
RDFS class	Entity class
RDF property	Attribute
RDFS class instance	Entity (object)

An analogous division is used in [24] where *indirect* and *direct* models correspond to the domain-independent and domain-specific approaches respectively. The authors argue that direct models are more suitable for domain-specific applications and propose a hybrid approach, where parts of the application model are domain independent and parts are domain specific.

The difference between domain-independent and domain-specific libraries is similar to the difference between JDBC [25] and JPA [26] in the relational database world. JDBC is a row-based access technology, which requires a lot of boilerplate code. In addition, not all databases fully adhere to the SQL[5] standard, so using JDBC, which is based on SQL, may bring compatibility issues. JPA, on the other hand, provides *object-relational mapping* (ORM) and makes access to various databases transparent.

There exists a number of OTM frameworks. However, the user base of these libraries is still relatively small, which is reflected in the small amount of information available about them. Some libraries are tied to a specific underlying storage, others' APIs do not reflect the specifics of access to semantic data. Ultimately, one would like to be able to determine which of the libraries is the most suitable for his/her object-oriented application.

2.2. Performance benchmarking in Java

As was already stated, the developed comparison framework consists of a set of qualitative criteria and a performance benchmark. The benchmark is written in Java, which is hardly a surprise since Java is the most prominent language in the Semantic Web community. Therefore, approaches to performance testing in Java were also reviewed.

Java-based performance testing has certain specifics, which are given by the nature of the Java platform, where each program is compiled into an intermediate language – the *bytecode* – and interpreted at runtime by the *Java virtual machine* (JVM). JVM is able to optimize frequently executed parts of the running program. These optimizations occur during execution of the application. JVM also automatically manages application memory via a mechanism called *garbage collection* which automatically frees memory that is no longer required by the application because objects occupying the memory are no longer referenced. Of course, if the application forgets to discard objects it does not use anymore – it has a so-called *memory leak* – it may eventually run out of free memory. Although garbage collection is nowadays extremely efficient, it still requires CPU time and may influence the application performance. Java benchmarks thus usually take into account application throughput and its dependence on the amount of memory available (*heap size*). The best known benchmarks in this area are SPEC [27] and DaCapo [28]. DaCapo measures application throughput w.r.t. heap size. In addition, it samples heap occupation and composition and new object allocation rate. The authors of [29] count garbage collection events and the time JVM spends executing them.

Generally, when benchmarking Java applications, it is necessary to execute a *warm-up* period, which allows JVM to perform code optimizations because otherwise, the results could register a significant speedup after the initial unoptimized phase. Georges et al. [30] discuss further practices for executing Java application benchmarks and interpreting their results, introducing a statistically rigorous methodology for Java benchmarking. Their methodology was also used in the experiments in this work.

3. Related work

This section overviews related existing approaches. First, we present existing object-triple mapping library comparisons (both feature and performance-wise) and then take a look at established storage benchmarks and discuss how our benchmark framework complements them.

There are very few comparisons of OTM libraries. Most benchmarks and comparisons concentrate on the underlying storage and its performance. While this is certainly a very important aspect, one needs to be able to make a qualified decision when choosing from a set of object-triple mapping frameworks. Holanda et al. [31] provide a performance comparison between

[5] Structured Query Language, a data query, definition, manipulation and control language for relational databases.

their framework JOINT-DE and AliBaba. However, their benchmark uses a minimalistic model (create operation works with a single entity with one string attribute) and no other library is taken into account. Cristofaro [32] provides a short feature comparison of various OTM libraries supporting the Virtuoso storage server [17]. A more recent and elaborate feature-based comparison can be found in [33] which compares a set of selected libraries from the point of development activity, documentation, ease of use and querying capabilities. In [34], authors of the well-known LUBM benchmark [35] do not present any actual comparison, but they introduce a set of requirements a Semantic Web knowledge base system benchmark should fulfill. Although they again target mainly semantic database benchmarks, the criteria defined in their article apply to a large extent to this work as well and their satisfaction will be shown in Section 4.

The number of benchmarks comparing storage systems is, on the other hand, relatively large[6]. The best known benchmarks in this area are the Berlin SPARQL Benchmark [36], LUBM [35] and UOBM [37]. These benchmarks evaluate the performance of the storage engines and, in case of the more expressive LUBM and UOBM, also allow to compare the engines in terms of their inference capabilities. Our work in [38] represents a step towards the application-oriented usage because it leverages the UOBM model and data and uses a set of queries specifically designed to simulate the behavior of OTM libraries. All these benchmarks use the SPARQL endpoint provided by the evaluated system, which makes them easy to apply to various storage technologies. The benchmark developed here concentrates more on the overhead OTM libraries represent on top of the storage execution time. That is why, in Section 5, only one underlying storage was used to eliminate its influence on the results.

4. Design of the comparison framework

This section introduces the comparison framework which consists of two parts:

1. A set of criteria used to evaluate the features supported by the compared libraries.
2. A performance benchmark.

[6]See for example the list at https://www.w3.org/wiki/RdfStoreBenchmarking, accessed 2018-01-02.

4.1. Feature criteria

We have designed a set of criteria, which can be used to evaluate an OTM library's suitability in a particular use case. The criteria can be split into three main categories:

General (GC) General criteria are based on the principles known from application development and relevant object-relational mapping features.

Ontology-specific (OC) These criteria take into account specific features of the Semantic Web language stack based on RDF and its design. They are not specific to OTM libraries and could be applied to libraries used to access semantic data in general.

Mapping (MC) Mapping criteria concern important techniques used for the object-triple mapping. They are motivated by the differences of ontologies on the one side and the object-model on the other side [39].

The selection is based on existing knowledge from other domains (e.g. ORM), literature and our own experience with developing ontology-based applications. We intended to incorporate also other ontology-based information systems, but we were not able to find publicly available source code of any ontology-based application which uses an OTM library. For each criterion, we describe its genesis, as well as the condition under which we consider the criterion to be satisfied.

GC1 – Transactions Transactional processing, i.e. splitting work into individual, indivisible operations, is one of the fundamental paradigms of computer science [40]. It originated in databases, but, due to its universally applicable principles, it is used throughout information systems with various levels of granularity. Nevertheless, support for transactions is one of the main features an OTM library has to provide, be it using an internal transaction engine or relying on the underlying storage.

Transactions are characterized by the *ACID* acronym, which stands for *Atomicity* of the transaction, eventual *Consistency* of data, *Isolation* of concurrent transactions and *Durability* of the transaction results. Given the complex nature of transaction isolation, where several isolation levels exist[7], and the lack of its treatment in all evaluated tools, our goal when examining trans-

[7]E.g. https://tinyurl.com/transaction-isolation, accessed 2018-03-01.

actional support in the OTM libraries is to evaluate their implementation of ACD.

Fully satisfied if: The library supports ACD user-controlled transactions.

GC2 – Storage access variability Storage access variability refers to the library's ability to connect to semantic storages of various vendors. While application access to relational databases is standardized in two ways – the *Open Database Connectivity* (ODBC) [18] is a platform-independent database access API; the *Java Database Connectivity* (JDBC) [25] is a Java-based database access API (a JDBC driver can use an ODBC driver internally) – the semantic world lacks a corresponding standard. For instance, although RDF4J-based access is supported by most of the industry's biggest repository developers, it is not possible to use it to connect to a Jena SDB store, which is backed by a relational database. To allow access to different storages, the OTM framework has to account for this either by having modules for access to various storages or by defining an API whose implementations would provide access to the repositories. Another possibility, although probably not very efficient, could be accessing a storage using a SPARQL [41] or *Linked Data Platform* [42] endpoint.

Fully satisfied if: The library supports connecting to a triple store by means of at least two different APIs (e.g. Jena, RDF4J) or via a generic SPARQL endpoint API.

GC3 – Query result mapping Besides reading individual instances with known identifiers, it is often necessary to read a set of instances corresponding to some search criteria (for example, read all reports with the given severity assessment which document events that occurred during the last month). Naturally, mapping of SPARQL query results to objects is expected, as SPARQL is the standard query language for the Semantic Web, but other languages like SeRQL (a Sesame query language) can be supported. A similar feature is supported by JPA [26], where SQL results can be mapped to entities or dedicated Java objects.

Fully satisfied if: SPARQL query results can be declaratively mapped to a business-level object model (entities).

GC4 – Object-level query language Related to *GC3* is the question of query languages supported by the OTM library. While SPARQL is adequate in most situations, it can be cumbersome to work with, especially given the fact that one has to deal with IRIs and prefixes. When querying the data from an object-oriented application, it is convenient to be able to leverage the object model in the queries. This is again supported by JPA [26], where one can use the *Java Persistence Query Language* (JQPL) – a query language with SQL-like syntax but exploiting classes, attributes, and relationships instead of tables and columns. Furthermore, it also defines the *Criteria API*, which allows one to construct queries dynamically at runtime using builder objects. Criteria API is especially suitable when a large number of optional filtering criteria can be used in the query and juggling with string concatenation would be difficult and error-prone. In addition, Criteria API provides the benefit of compile-time syntax checking, because it uses regular Java classes and methods.

Prototypical solutions already exist in the Semantic Web community. For instance, Hillairet et al. [43] use the Hibernate Query Language, an implementation of the standard Object Query Language [44] (OQL)[8] used by one of the most popular ORM vendors, to translate queries to/from SPARQL. Stadler and Lehmann [45], on the other hand, present an engine rewriting Criteria API queries to SPARQL.

Fully satisfied if: The library provides an object model-based query language which can be used to access the data (e.g., like JPQL).

GC5 – Detached objects As pointed out in [31], applications need to be able to *detach* persistent objects from their connection to the storage in order to work with them, for instance when the entity is passed up through web application's layers and transferred over the network in response to a client request or stored in an application-level cache. The opposite (an object which is always *managed*) can lead to an excessive amount of retained connections to the storage, blocking the resources of the machine. The difference between detached and managed entities is indeed in that managed entities are tracked by the persistence library, which watches for changes in these objects.

Fully satisfied if: The persistence framework allows to attach/detach objects to/from the repository connection.

GC6 – Code/ontology generator Setting up a database schema or an object model may seem like a one-time work, but as the application evolves, so does often the schema. Keeping both the ontology schema and the

[8] JPQL is another implementation of a subset of OQL.

object model in sync automatically can spare the developer a decent amount of time and bugs (e.g., a typo in an IRI). For this reason, JPA allows one to generate database schema from an object model [26]. In addition, many JPA implementations support also the converse – generating an object model from database tables. There also exist libraries like Owl2Java [46] which allow generating an object model from an ontology. To this end, three approaches to object model vs ontology schema transformation are discussed in [47] – manual mapping of an object model, its automatic generation from an ontological schema and a hybrid approach, which the authors deem most useful based on its correctness/efficiency ratio. In the hybrid scenario, a basic model is generated automatically and the developer then fine tunes it manually. Conversely, one could start with an object model, generate an ontology schema from it and continue extending the schema further.

Fully satisfied if: The library provides a generator to synchronize the object model with the ontology.

OC1 – Explicit inference treatment Whether a statement is asserted or inferred is not distinguishable from the formal point of view [48]. However, application developers need to use the information whether an attribute value is asserted/inferred to properly design application logic. The main reason is that inferred statements can only be modified through asserted statements. For instance, it is perfectly fine to remove a statement about an instance's type. However, if the type is inferred (e.g. because of a property range declaration), one cannot simply go and delete it, it is necessary to modify the statements upon which the inference is based. An OTM library has to deal with such situations consistently, e.g. by preventing modifications of inferred values. This issue would be even more important for OWL-based libraries because OWL is more expressive than RDFS and it allows richer inference.

Fully satisfied if: The library distinguishes handling of inferred and asserted statements.

OC2 – Named graphs Named graphs [49] are an important feature of RDF, as they allow to split RDF data into meaningful and identifiable parts inside an RDF dataset. Although there exist different semantics for treatment of named graphs, triple stores mostly adopt the one where the dataset's default graph represents a union or merge of its named graphs [50]. This strategy makes named graphs suitable for the logical structuring of data. Consider a company repository containing information about projects, business contracts and employees. It is sensible to let the employees occupy a different named graph than projects, possibly together with different degrees of access control.

Fully satisfied if: The OTM framework provides a way to access different graphs (default graph as well as named graphs) of an RDF dataset.

OC3 – Automatic provenance management RDF allows to record *provenance* information about the data using the RDF *reification* vocabulary [10]. This approach, although not without flaws [51], provides an interesting alternative to *auditing* in JPA, which is not standardized and is done by various libraries in an ad hoc manner. Quasthoff and Meinel [52] introduce a prototype which is able to generate provenance data in RDF processing, more specifically, it connects newly created statements to triples upon which they are based. We can formulate an example of benefits of automatic provenance tracking in terms of the benchmark model introduced later in Section 4.2.1 – instead of manually assigning the last editor or author of a report, the OTM could set it based on user session information available in the application.

Fully satisfied if: The library provides a configurable way to automatically generate provenance data about operations performed by the library, including the operation originator and time frame.

MC1 – Inheritance mapping Class hierarchies represent a major component of every domain conceptualization. Although RDFS does not support advanced class declaration expressions like disjointness, intersection or union, which are part of OWL [11], it allows building class hierarchies using the RDFS `subClassOf` property. When mapping class hierarchies to an object model, the developer may encounter one technical issue: some programming languages, including Java, do not support multiple inheritance on the class level, whereas RDFS does. This is often resolved using *interfaces*, which do not have this restriction.

Mapping ontological inheritance to an object model actually brings subtle conceptual issues. For example, consider an ontology containing the following statements:

```
@prefix rdf:
 <http://www.w3.org/1999/02/22-rdf-syntax-ns#> .
@prefix rdfs:
 <http://www.w3.org/2000/01/rdf-schema#> .
@prefix ex:
 <http://onto.fel.cvut.cz/ontologies/example/> .

ex:A rdf:type rdfs:Class.
ex:B rdf:type rdfs:Class; rdfs:subClassOf ex:A.
```

```
ex:C rdf:type rdfs:Class; rdfs:subClassOf ex:A.
ex:a rdf:type ex:B, ex:C.
```

An application that would attempt to load an entity a (mapped to an RDFS class instance `ex:a`) as an instance of an entity class A[9] (mapped to an RDFS class `ex:A`) might experience the following three problematic scenarios:

1. If inference is not enabled in the storage, `ex:a` is not an instance of `ex:A` so a cannot be loaded as an instance of A.
2. If inference is enabled and the application attempts to remove the loaded entity a of entity class A, it will in effect attempt to remove an inferred class assertion `ex:a rdf:type ex:A`.
3. The OTM library might attempt to look for an explicit class assertion in the same RDFS class inheritance hierarchy. However, that would lead to an ambiguous result because both `ex:a rdf:type ex:B` and `ex:a rdf:type ex:C` are explicit class assertions, but they are at the same level in the hierarchy.

Thus, OTM frameworks need to support inheritance mapping due to its ubiquity, yet attention has to be paid to its treatment.

Fully satisfied if: The library supports both single-class and multiple-class inheritance mapping.

MC2 – Unmapped data access An application's object model may represent only a part of the domain ontology. Similarly, one may start building an application using a simplified object model and gradually extend its coverage of the underlying ontology. Yet, there exist situations in which access to the portion of the ontology not mapped by the object model may be required. Consider an aviation safety occurrence reporting application (will be discussed in the next section) based on an aviation safety occurrence reporting ontology which extends a generic safety reporting ontology [53]. The generic safety ontology can be used for other high-risk industries such as power engineering, railroad transportation, etc. There may exist an application which allows browsing safety occurrence reports from different industries using an object model based on the generic ontology. Yet, the application could provide access to the industry-specific property values without them being mapped by the object model.

Another example of the benefits of accessing unmapped ontological data is ontology-based classification. In the aviation safety reporting tool, events can be classified using RDFS classes from a predefined vocabulary [54]. However, the application does not need to map these classes in order to use them if the OTM library allows access to these unmapped types.

Fully satisfied if: The library allows to access data which are not mapped by the OTM into the object model.

MC3 – RDF collections and containers By default, all RDF properties are plural. In addition, RDF does not impose any kind of ordering restriction on their values, giving them effectively the semantics of a mathematical *set*. RDF defines the notions of *containers* and *collections*, which allow to represent groups of values with different kinds of semantics. For instance, an RDF container `rdf:Seq` represents an ordered unbounded sequence of elements [10]. On the other hand, an RDF collection `rdf:List` is a sequence of elements with a known length [10]. Lists (or sequences of data in general) are ubiquitous in programming, so it is important to support mapping of such RDF structures.

Fully satisfied if: The library allows to manage both RDF collections and containers.

4.2. Performance benchmark

To provide a full picture of the OTM libraries, we complement the (qualitative) feature comparison with a (quantitative) performance comparison. To make the comparison more usable and repeatable, it was decided to design a benchmark framework which could be used to compare as large variety of OTM libraries as possible.

As [34] and [55] point out, a benchmark should aim at simulating real-world conditions as much as possible, especially avoiding microbenchmarks, which are easy to get wrong. Luckily, we had an experience of developing ontology-based safety information solutions for the aviation domain in the INBAS[10] and B-INBAS[11] projects. A part of these solutions is a safety occurrence reporting tool (RT)[12]. RT is a web applica-

[9] We shall use the sans serif font to express object language constructs throughout this work.

[10] https://www.inbas.cz, accessed 2018-01-03.

[11] https://www.inbas.cz/web/binbas, accessed 2018-01-06.

[12] Source code is available at https://github.com/kbss-cvut/reporting-tool, accessed 2018-01-06.

tion written in Java, backed by ontologies and ontological data and its functionality and model were used as a base for the benchmark framework. One might ask why we did not reuse the object model and data generator from one of the established benchmarks. There are several reasons for this decision. As far as the generator is concerned, the main goal was to restrict unwanted randomization. We wanted the benchmark to generate precise numbers of instances so that all the libraries would have the same conditions. Also, the generator had to create object instances, whereas, for example, the LUBM generator outputs RDF. The model is based on a real-world application, it is small enough to enable adaptation to new benchmarked libraries, yet it exercises most of the common mapping features (described in Section 4.2.1).

4.2.1. Benchmark model

We took an excerpt of the RT model and modified it by removing some of the attributes which use features already present in other parts of the model. It is built upon a conceptualization of the aviation domain called the *Aviation Safety Ontology* [56] which is based on the *Unified Foundational Ontology* (UFO) [57]. RDFS serialization of the model ontology is visualized in Figure 2.

The goal of the benchmark is to exercise both common mapping features known from JPA and other features specific to OTM revealed in our experience with building ontology-based applications. Therefore, the model contains both singular and plural relationships, properties with literal values, inheritance, and references to resources not mapped by the object model (the `ufo:has_event_type` property references an external vocabulary, a `doc:logical_record` references an unmapped `doc:question-doc:answer` tree). The last point illustrates how the object model can be connected to a larger domain model, which exceeds the application's area of operation. The corresponding object model is illustrated in Figure 3.

We conclude this part with a remark regarding instance identifiers. The identifier attribute was not included in declarations of the entity classes in the model because various libraries use various types for the identifier, e.g. String, URI or a vendor-specific class. Moreover, since some of the libraries do not make a mapped object's identifier (the RDF resource) accessible at all, a *key* property was added, which explicitly specifies a unique identifier of the instance. The notion of a unique key identifying an instance was formalized in the OWL 2 specification [12] in the form of `owl:hasKey` axioms. In contrast to them, our key has no formal semantics and its uniqueness is purely conventional. It is used to find matching objects when verifying operation results.

4.2.2. Benchmark operations

The set of operations executed by the benchmark is supposed to represent a common domain-specific application scenario. Therefore, the scenarios include a basic set of create, retrieve, update and delete (CRUD) operations, plus a batch persist and a find all query. The CRUD operations represent a typical form of operation of a business application, where data are persisted, updated, queried and occasionally deleted. The find all query is another example, where the user requests a list of all instances of some type, e.g., for a table of all reports in the system. The batch persist, on the other hand, may represent a larger amount of data being imported by an application at once.

Each operation consists of three phases; it has optional *set up* and *tear down* phases, which are not measured but are used to prepare test data and verify results respectively. Between these optional phases is the actual *execution* phase, duration of which is measured. Test data are generated before each operation.

OP1 – Create *OP1* represents a typical operation performed by a domain-specific business application. It creates an object graph centered around some entity (an OccurrenceReport instance in this case) and then persists it. This simulates data being received for instance via REST services, connecting them to the logged-in user and persisting them into the repository. To make the results more representative, this transaction is repeated multiple times for separately created objects – a longer runtime reduces the influence of things like just-in-time compilation, garbage collection and possibly imprecise small-scale time measurement in JVM. The operation specification is as follows:

1. *Set up* Persist Person instances to allow newly created reports to connect to them (simulating interconnection of existing and newly added data).
2. *Execution* Assign a random author and last editor to the generated reports and persist all of them in separate transactions.
3. *Tear down* Verify the persisted data.

OP2 – Batch create As was already stated, the batch create operation represents for example data being imported or processed by an application in one transaction. Thus, the mode of operation is almost the same

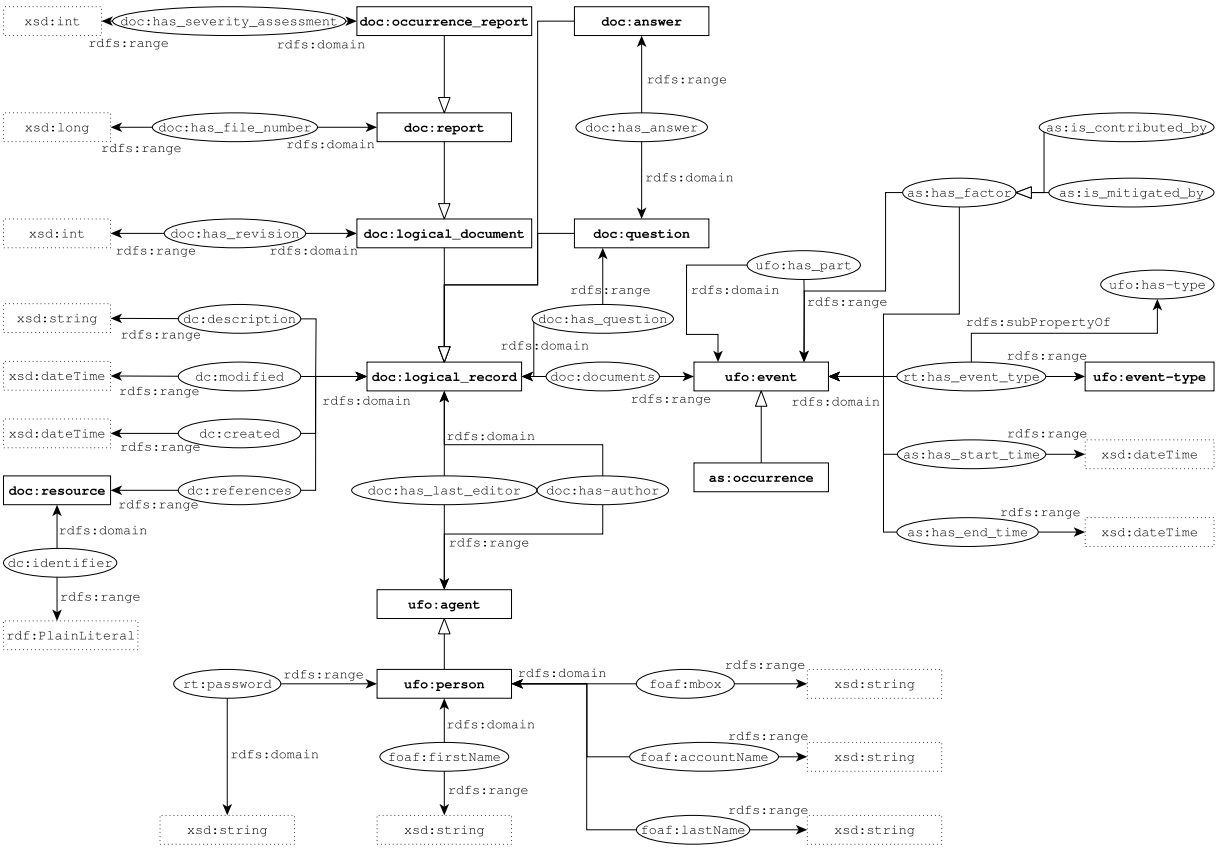


Fig. 2. Benchmark ontology visualization. Rectangles with solid line represent RDFS classes, rectangles with dotted line denote literal datatypes and ellipses are RDF properties used as RDF triple subjects/object. Each edge represents an RDF triple with a source/target node label representing the triple subject/object and the edge label representing the triple predicate. Unlabeled edges ended with a hollow arrow denote triples with the `rdfs:subClassOf` predicate. Prefixes `ufo`, `as`, `doc`,`xsd`,`rdf` and `rdfs` have been introduced in Listing 1. Prefix `dc` denotes the Dublin Core namespace `http://purl.org/dc/terms/`, prefix `foaf` denotes the FOAF namespace `http://xmlns.com/foaf/0.1/` and `rt` denotes an application specific namespace `http://onto.fel.cvut.cz/ontologies/reporting-tool`.

as *OP1*, only now all the entities are persisted in one large transaction.

1. *Set up* Persist Person instances to allow newly created reports to connect to them (simulating interconnection of existing and newly added data).
2. *Execution* Assign a random author and last editor to the generated reports and persist all of them in one large transaction.
3. *Tear down* Verify the persisted data.

OP3 – Retrieve *OP3* stands for the application requesting a specific entity together with its object graph (an instance and its property values in RDF terms). Again, to increase the clarity of the measurement, multiple objects are read one by one. This operation also verifies that all the required data were loaded by the persistence library.

1. *Set up* Persist test data using the same process described in *set up* and *execution* of *OP2*.
2. *Execution* Iterate through all existing reports, read each report using the existing report's identifier. Verify that the loaded report corresponds to the existing one.

OP4 – Retrieve all A "find all instances of a type" query is a typical operation for many applications. Its implementations can vary. Some libraries support mapping SPARQL query results to objects, so a SPARQL SELECT query is used, others contain such a method directly in their API. The goal is ultimately the same – retrieve a relatively large amount of entities together with their references at once.

1. *Set up* Persist test data using the same process described in *set up* of *OP3*.

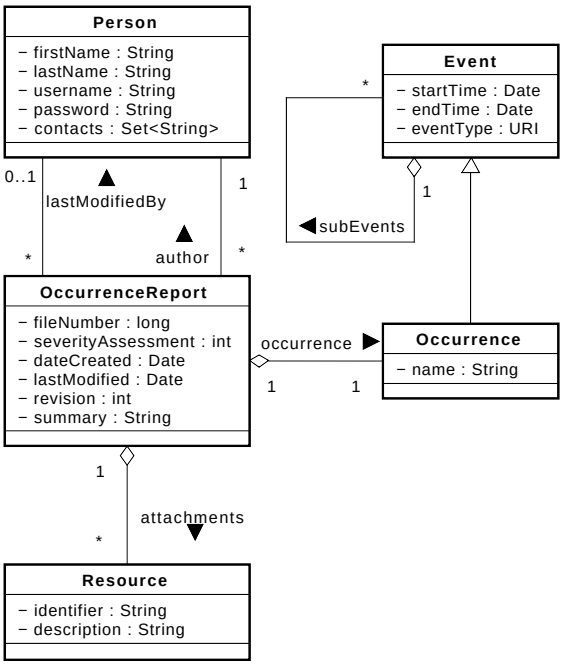


Fig. 3. UML class diagram of the benchmark model.

2. *Execution* Read all reports and verify them. If the library API provides a dedicated method for this task, use it. Otherwise, use a SPARQL SELECT query.

OP5 – Update Update merges a modified entity into the repository. Several of its attributes and attributes of objects it references are updated, with both literal values and references to other objects being changed. Also, a reference to a new object that needs to be persisted is added. This operation is done in a single transaction. Several objects are updated in this way.

1. *Set up* Persist test data using the same process described in *set up* of *OP3*.
2. *Execution* For one half of the existing reports, perform the following updates (each object in a separate transaction):
   - set a different last editor (singular reference attribute) and last modified date (singular datetime attribute),
   - change occurrence name (singular string attribute),
   - change severity assessment (singular integer attribute),
   - increment revision number by one (singular integer attribute),
   - add a newly generated attachment (plural reference attribute),

   and merge the updated entity into the repository.
3. *Tear down* Verify the updates.

OP6 – Delete While modern triple stores do not particularly concentrate on removal operations since storage capacity is relatively cheap and many systems tend to create new versions instead of modifying existing data, business applications sometimes need to delete data, e.g. for privacy or security reasons. *OP6* simulates precisely this situation, where an object is deleted, including its references. Once again, multiple objects are deleted in separate transactions to improve result robustness.

1. *Set up* Persist test data using the same process described in *set up* of *OP3*.
2. *Execution* Delete one half of the existing reports, including their dependencies, i.e. everything from their object graph except the author and last editor.
3. *Tear down* Verify that all relevant data were removed.

A benchmark runner is used to execute the operations according to a configuration. The runner collects a number of statistics:

- The fastest and slowest round execution time,
- Mean round execution time,
- Q1, Q2 (median) and Q3 of round execution times,
- Total execution time,
- Standard deviation of round execution times.

In addition, it allows a file to be configured into which raw execution times for rounds are output so that they can be further analyzed.

The whole benchmark framework is available online[13]. Table 2 shows conformance of the comparison framework developed here to the Semantic Web knowledge base system benchmark requirements defined in [34] and discussed in Section 3.

5. Libraries selected for comparison

In this section, we introduce the frameworks selected for the comparison. Several OTM libraries were

[13] https://kbss.felk.cvut.cz/gitblit/summary/otm-benchmark.git, accessed 2018-05-22.

Table 2

Evaluation of benchmark criteria defined in [34] on our benchmark. SUT stands for system under test

Req#	Description	Evaluation
G1	Scalability	The data generator allows generation of datasets of arbitrary size.
G2	Broad architectural scope	The comparison framework is platform and storage independent. The benchmark requires Java.
G3	Easy to add new SUT	The benchmark API requires implementation of approximately ten short classes to add a new SUT.
G4	Meaningful metrics	The benchmark collects commonly used performance metrics and statistics (mean, standard deviation etc.).
G5	Controlled measurements	Data generation is repeatable, warm-up period is included and configurable.
S1	Real-world workload	The benchmark measures performance of typical domain-specific object-oriented application operations.
S2	Metrics supporting various (possibly conflicting) requirements	The benchmark measures execution time and memory consumption. Plus it provides feature comparison criteria.
S3	Metrics capture result conformance to expectations	Not fully applicable (original intention was for inference completeness/soundness). The benchmark verifies correct results of the operations against predefined data.

selected based on their popularity, level of maturity and development activity – libraries with latest changes older than five years were omitted. The feature comparison is platform agnostic, so a diverse set of libraries was chosen:

- ActiveRDF
- AliBaba
- AutoRDF
- Empire
- JAOB
- JOPA
- KOMMA
- RDFBeans
- RDFReactor
- The Semantic Framework
- Spira
- SuRF

However, since the benchmark part of the comparison framework is based on Java, only Java OTM libraries were evaluated so that platform specifics (e.g. code being compiled to native code like in C++, to bytecode like in Java/C# or directly interpreted like in Ruby) do not influence the results. The selection is further narrowed by requiring the libraries to support triple store access – using plain RDF files for real-world applications is not suitable in terms of performance and data management. Finally, to improve the comparability, we decided to concentrate only on libraries which support access to a single triple store. This eliminates the influence of the storage technology and its implementation on the performance benchmark, where e.g. a Jena relational database-based storage could have different performance characteristics than an RDF4J storage. We picked RDF4J[14] [8] because the amount of mature tools for it is the greatest, including industry-grade storage implementations like GraphDB [16]. As a result, the following libraries were evaluated in the benchmark:

- AliBaba
- Empire
- JOPA
- KOMMA
- RDFBeans

RDFReactor was a candidate for the performance benchmark as well, but its RDF4J/Sesame storage connector works only with an embedded in-memory store and cannot connect to a remote repository. Thus, it was excluded from the benchmark selection.

The mapping in the selected Java libraries is realized via annotations which specify the mapping of Java entity classes to RDFS classes and attributes to RDF properties. Listing 3 shows an example entity class declaration in Empire.

[14] Since RDF4J is an evolution of the Sesame API, we will use the terms interchangeably throughout this paper. We will stress the difference when necessary.

Listing 3: Example of an Empire Java entity class with annotations specifying mapping to the RDF(S) data model. *Getters* and *setters* are methods used to get and set the values of attributes respectively.

```
@Entity
@Namespaces({"as",
"http://onto.fel.cvut.cz/ontologies/a−s/",
"ufo","http://onto.fel.cvut.cz/ontologies/ufo/",
"rt", "http://onto.fel.cvut.cz/ontologies/reporting−tool/"})
@RdfsClass("ufo:event")
public class Event implements SupportsRdfId {

    @RdfProperty("as:has_start_time")
    private Date startTime;

    @RdfProperty("as:has_end_time")
    private Date endTime;

    @OneToMany(fetch = FetchType.EAGER, cascade = CascadeType.ALL)
    @RdfProperty("ufo:has_part")
    private Set<Event> subEvents;

    @RdfProperty("rt:has_event_type")
    private URI eventType;

    // Getters and setters follow
}
```

The following paragraphs introduce all the evaluated OTM frameworks.

5.1. ActiveRDF

ActiveRDF [58] is an object-oriented API for Semantic Web data written in Ruby. It is built around the premise that dynamically-typed languages like Ruby are more suitable for semantic data representation because, like RDF, they do not require objects to conform to a specific class prescription. ActiveRDF represents resources by object proxies, attaching to them methods representing properties. Calling these methods translates to read/write queries to the storage. In addition, it supports an object-oriented query syntax based on PathLog [59] and is able to actually generate convenience filter methods automatically.

In fact, ActiveRDF is not a typical OTM framework like AliBaba or Empire, where RDFS classes are mapped to entity classes. Instead, it treats RDF resources as objects to which it dynamically attaches methods corresponding to their properties. One caveat of the highly dynamic nature of ActiveRDF is pointed out in [60] – it does not offer type correctness and typographical error verification present in libraries based on statically-typed languages such as Java.

5.2. AliBaba

AliBaba [21] is a Java framework for developing complex RDF-based applications. It is an evolution of the Elmo [61] library. AliBaba is an extension to the original Sesame API, so it supports only storages accessible via a Sesame/RDF4J connector. Its API is centered around an ObjectConnection, which allows to persist and retrieve objects mapped by AliBaba's OTM. It uses dynamically generated proxy objects to track updates. AliBaba also allows to use SPARQL queries to support more complex strategies of mapping values to Java attributes. On the other hand, the entities managed by AliBaba are not able to store IRIs of the resources they represent, which is a severe limitation requiring the developer to maintain a map of objects and identifiers externally.

In addition to OTM, AliBaba contains an HTTP server which can automatically make resources accessible as REST services, providing querying, inserting and deleting capabilities.

5.3. AutoRDF

AutoRDF [62] is a framework facilitating handling of RDF data written in C++. The fact that it is written in C++ allows it to run for instance on embedded devices – actually, one of its intended usages is in ID document verification. It is built on top of the Redland RDF library[15]. Similar to ActiveRDF, AutoRDF does not really store any data in the runtime objects. Instead, they act as proxies and all operations are directed to the underlying RDF. Besides RDFS, AutoRDF supports also several OWL features like `owl:oneOf` (mapped to enumerations), cardinality restrictions and classes being restrictions of other classes. It also supports anonymous resources (identified by RDF blank nodes) and automatic documentation generation based on selected RDFS vocabulary terms (e.g. `rdfs:label`, `rdfs:comment` or `rdfs:isDefinedBy`).

5.4. Empire

Empire [22] is an RDF-based implementation of the JPA standard [26]. Therefore, its API should be familiar to developers used to working with relational databases in Java. However, since parts of JPA deal with the physical model of the relational database under consideration (e.g. the @Table and @Column annotations), Empire is not fully JPA-compliant. On the other hand, it does support the basic EntityManager API, query API, and entity lifecycle. Unfortunately,

[15] http://librdf.org/, accessed 2018-03-10.

Empire's documentation is limited and it is often unclear which other parts of JPA are implemented.

It adds a set of RDF-specific annotations, which are used to express the mapping of Java classes and attributes to RDF classes and properties (see Listing 3). It does not support anonymous resources, so all instances it works with must have an IRI.

5.5. JAOB

The Java Architecture for OWL Binding (JAOB) [63], as its name suggests, is primarily intended for OWL ontologies. However, with minor adjustments, RDF(S) ontologies can be manipulated by it. First, the annotation used for declaring class mapping is called OWLClass, but an RDFS class IRI can be used as its value. Then, in contrast to RDF(S), OWL defines three kinds of properties - object properties, datatype properties and annotation properties [11]. These properties have equivalent annotations in JAOB. To use them with RDF(S), it is necessary to use OWLObjectProperty for attributes whose value is expected to be a reference to another object (RDF resource), OWLDataProperty can be used for every other attribute, i.e. with a literal value.

JAOB is built on top of OWL API, so it supports only ontologies stored in files. It basically works as *(un)marshaller*, so it is able to load or save collections of mapped objects. However, there is, for example, no direct way to update an entity in JAOB.

JAOB also contains a generator, which allows generating Java classes based on an ontology schema and vice versa.

5.6. JOPA

The Java OWL Persistence API (JOPA) [19, 23] is a persistence library primarily designed for accessing OWL ontologies. However, it can be used to work with RDF data as well. Its API is designed so that it resembles the JPA as much as possible, but it does take semantic data specifics like named graphs or inference into account. JOPA tries to bridge the gap between the object-oriented application world, which uses the *closed-world assumption* (missing knowledge is assumed false), and the ontological world, which is based on the *open-world assumption* (missing knowledge can be true in some worlds and false in others), by splitting the model into a static and dynamic part. The static part is mapped to the object model and its validity is guarded by *integrity-constraints* with closed-world semantics [64]. The dynamic part is not directly mapped (although it is accessible to a limited extent) and may evolve freely without affecting the object model.

Using JOPA with RDF(S) ontologies requires minor adjustments because JOPA is OWL-based. These adjustments are similar to those described for JAOB above.

5.7. KOMMA

The Knowledge Management and Modeling Architecture (KOMMA) [65] is a Java framework for building applications based on semantic data. A part of this framework is an object-triple mapping module, but the library itself has much richer functionality, including support for building graphical ontology editors based on the Eclipse Modeling Framework[16]. OTM in KOMMA is based on Java interfaces representing RDFS classes, for which it generates dynamic implementations at runtime. This allows, for instance, to accommodate support for multiple inheritance into the model. In addition, KOMMA also supports transaction management, caching and RDF named graphs.

5.8. RDFBeans

RDFBeans [66] is another OTM library built on top of RDF4J. It allows two forms of the object model: 1) the object model may consist of Java classes representing the RDFS classes; 2) the object model may consist of interface declarations forming the mapping, RDFBeans will generate appropriate implementations at runtime using the *dynamic proxy* mechanism. RDFBeans supports, besides the basic features like transactions and inheritance, also mapping of Java collections to RDF containers or operation cascading, e.g. when an updated entity is merged into the storage, objects referenced by this entity are merged as well. Instances of entity classes without an explicitly declared identifier attribute are mapped to RDF blank nodes.

5.9. RDFReactor

RDFReactor [67] is yet another Java OTM library. It supports code generation and uses proxy objects to access RDF data directly, instead of storing attribute values in entities. In addition to the mapped properties, RDFReactor entity classes also contain an API for

[16] http://www.eclipse.org/modeling/emf/, accessed 2018-01-02.

type information manipulation – methods getAll, add and remove allow to retrieve and update types of an instance at runtime. Besides RDFS, RDFReactor also supports selected OWL features like cardinality constraints, inverse properties.

All entity classes mapped by RDFReactor extend the ReactorBase class, which acts as a generic representation of an RDF resource, allowing untyped access to all properties the resource has. Subclasses can then explicitly map these properties to provide type-safe access.

5.10. The Semantic Framework

The Semantic Framework (SF) [68] is a Java library for object-oriented access to ontologies. Similar to JAOB or JOPA, SF is primarily built for OWL, but it can, again, be used with RDFS as well. It internally uses OWL API to process RDF files and its core is an extension of the Jenabean library [69]. The combination of Jena (internally used by Jenabean) and OWL API is in our opinion rather odd given the fact that Jena is able to process RDF files. The mapping is realized using Java annotations in entity classes.

5.11. Spira

Spira [70] is a framework for viewing RDF data as model objects written in Ruby. It allows to access RDF data both as domain objects (entities) and as RDF statements. In particular, one may view and edit statements representing an entity directly from its object representation. On the other hand, it is also possible to create RDF statements directly, without any object representation of the resource. Spira allows to 'view' resources as instances of various classes without requiring them to explicitly have the corresponding RDF type statement. It also allows to map one Ruby class to multiple ontological types, to work both with named and anonymous resources or to use localized property values, i.e. a single-valued property can have a value in multiple languages.

Spira uses RDF.rb[17] to access the underlying RDF data. It can work with multiple repositories at once, all of them accessed in a thread-safe manner.

5.12. SuRF

SuRF [71] is a Python object RDF mapper. SuRF is built on top of the RDFLib[18] library and allows access to various triple stores, including Sesame [8], Virtuoso [17] and a generic SPARQL endpoint. Attributes of SuRF entities are by default loaded lazily and references to other entities are automatically accompanied by convenience inverse versions, i.e. a knows attribute of an entity is complemented by an is_knows_of attribute on the target instances.

SuRF contains convenience API which can be used to retrieve instances of the mapped classes and filter them by their attribute values. One subtle feature of SuRF is that it allows to retrieve resources regardless of whether they are actually present in the triple store. To check for their existence, a dedicated is_present method has to be called.

5.13. Libraries omitted from selection

There are several OTM libraries which were omitted from the selection. Elmo [61], Jastor [72], Jenabean [69], Owl2Java [46] or Sommer [73] are obsolete, with their last versions published more than five years ago.

We also wanted to include a purely SPARQL endpoint-based solution, which would allow us to compare the performance of a storage agnostic library to libraries exploiting vendor-specific optimizations, but RAN-Map [74] is not published online (neither sources nor binaries) and TRIOO [75] is buggy and we were not able to adapt it to the benchmark application without significant changes to its code base (for example, it does not insert resource type statements on persist).

Libraries like JOINT-DE [31] (and the original JOINT), Sapphire [60] and the Semantic Object Framework [76] were excluded because their source code is not publicly available and the articles describing them do not provide enough details to evaluate the comparison criteria.

6. Feature comparison

Comparison of features of the selected libraries according to the criteria defined in Section 4 is summarized in Table 3. In the table, ✓ signifies that the crite-

[17] https://github.com/ruby-rdf/rdf, accessed 2018-03-20.

[18] https://github.com/RDFLib/rdflib, accessed 2018-03-20.

rion is fully satisfied according to the condition defined in Section 4. ○ signifies partial satisfaction and the text should be consulted for further details. We now expand on the results.

6.1. GC1 – Transactions

AutoRDF, JAOB and the Semantic Framework do not support transactions likely due to their lack of support for triple stores in general (more details in Section 6.2). While ActiveRDF, RDFReactor and Spira do support access to regular triple stores and for example, RDFReactor internally makes use of the store's transactional mechanism, they do not allow the programmer to control the transactions externally. Empire's API hints towards support for transactional processing but its implementation is rather strange. As will be discussed in Section 7, operations which insert/update/remove data are automatically committed without any way of preventing this behavior. So, for example, it is not possible to bind updates to multiple entities into one transaction.

AliBaba, KOMMA, and RDFBeans support transactions by relying on the underlying storage's transaction management, i.e. starting an OTM-level transaction begins a database-level transaction and its end causes the database-level transaction to end as well. SuRF, on the other hand, tracks the changes to objects by marking their updated attributes dirty and allows the programmer to commit the changes manually. JOPA handles transactions by creating clones of the manipulated objects and tracking their changes locally, pushing them into the storage on commit. In addition, the changes are stored in transaction-local models and are used to enhance results of subsequent operations in the same transaction, e.g. when an entity is removed, it will not be in the results of a find all query executed later in the same transaction.

6.2. GC2 – Storage access variability

AutoRDF, JAOB and the Semantic Framework support only access to RDF data stored in text files, albeit with various serializations. AliBaba and RDFBeans are tightly bound to the Sesame API, so they can access exclusively Sesame/RDF4J storages.

While KOMMA does not have any other implementation than RDF4J storage access, its internal APIs are designed so that new storage connector implementations can be added.

ActiveRDF supports access to various storages, including Jena, Sesame or a generic SPARQL endpoint. Empire contains storage access modules which can implement the required access API. For instance, it provides a connector to the Stardog database[19]. JOPA defines a storage access layer called the *OntoDriver*. This allows switching the underlying storage easily. Currently, JOPA supports Jena, OWL API and Sesame/RDF4J storages. RDFReactor uses a similar mechanism – an underlying storage access layer called *RDF2Go*, which at the time of writing supports Jena and RDF4J access. Spira, thanks to its reliance on RDF.rb, can access various storages, including Sesame, AllegroGraph[20] or MongoDB[21]. Similarly, SuRF can use the RDFLib to connect to AllegroGraph, Sesame or a SPARQL endpoint.

6.3. GC3 – Query result mapping

In contrast to JAOB, which does not provide any query API at all, AutoRDF, RDFBeans, the Semantic Framework and Spira do not provide a query API either but contain at least a find all method, which allows to retrieve all instances of the specified type. In case of RDFBeans the situation is peculiar because its RDFBeanManager wraps an instance of RDF4J RepositoryConnection which supports SPARQL query execution, yet RDFBeanManager does not expose this functionality. SuRF allows to execute arbitrary SPARQL queries, but cannot map their results to entities.

While ActiveRDF does not support mapping SPARQL query results to entities, its advanced query API obviates this issue by supporting almost complete SPARQL grammar [58].

Finally, AliBaba, Empire, JOPA and KOMMA support mapping SPARQL query results to entities by allowing to specify target entity class when retrieving query results.

6.4. GC4 – Object-level query language

ActiveRDF contains a variation of a criteria query API containing methods representing query operations like selection, filtering and projection. Additionally, it automatically generates methods for filtering objects by their properties, e.g. it will provide a method Per-

[19] https://www.stardog.com/, accessed 2018-03-20.
[20] https://franz.com/agraph/allegrograph/, accessed 2018-03-20.
[21] https://www.mongodb.com/, accessed 2018-03-20.

Table 3

Selected OTM libraries compared using criteria defined in Section 4. × means no support, ○ represents partial support (consult the main text for further details), ✓ is full support of the feature and N/A signifies that the feature cannot be evaluated in the particular case

Criterion	ActiveRDF	AliBaba	AutoRDF	Empire	JAOB	JOPA
GC1 (Transactions)	×	✓	×	○	×	✓
GC2 (Storage-agnostic)	✓	×	×	✓	×	✓
GC3 (Query result mapping)	○	✓	×	✓	×	✓
GC4 (Object query language)	✓	×	×	×	×	×
GC5 (Detached objects)	×	×	×	✓	✓	✓
GC6 (Model Generator)	N/A	✓	✓	×	○	○
OC1 (Explicit inference)	×	×	×	×	×	✓
OC2 (Named graphs)	✓	×	×	✓	×	✓
OC3 (Provenance management)	×	×	×	×	×	×
MC1 (Inheritance)	N/A	✓	✓	○	×	○
MC2 (Unmapped data access)	✓	×	×	×	×	✓
MC3 (RDF containers/collections)	×	✓	×	×	×	○

Criterion	KOMMA	RDFBeans	RDFReactor	SF	Spira	SuRF
GC1 (Transactions)	✓	✓	×	×	×	✓
GC2 (Storage-agnostic)	○	×	✓	×	✓	✓
GC3 (Query result mapping)	✓	×	×	×	×	×
GC4 (Object query language)	×	×	×	×	×	✓
GC5 (Detached objects)	×	✓	×	✓	×	×
GC6 (Model Generator)	×	×	✓	×	×	N/A
OC1 (Explicit inference)	×	×	×	×	×	×
OC2 (Named graphs)	✓	×	✓	×	×	×
OC3 (Provenance management)	×	×	×	×	×	×
MC1 (Inheritance)	○	○	○	○	○	N/A
MC2 (Unmapped data access)	×	×	✓	×	✓	✓
MC3 (RDF containers/collections)	×	✓	×	○	×	×

son.find_by_name for a class Person with an attribute name. A similar API exists in SuRF.

None of the other libraries support any object-level query language, so their users have to resort to SPARQL queries, if available.

6.5. GC5 – Detached objects

ActiveRDF, AliBaba, AutoRDF, RDFReactor, Spira, and SuRF do not support detached objects because their entities act as proxies which load attribute values from the repository when they are accessed for the first time or on each access, depending on the internal implementation. Conversely, setting attribute values causes the changes to be written into the storage immediately (except for SuRF which tracks changes locally). Therefore, they have to hold onto a connection to the storage in order to provide basic data access functions. Similarly, although it would appear that KOMMA does support detached objects because it is possible to close a KOMMA IEntityManager and still access attributes of an object read by the closed manager, the contrary is true. As pointed out in [31], the generated proxy objects retain a connection to the underlying storage. Closing an IEntityManager only closes a delegate object.

Empire, JAOB, JOPA, RDFBeans and the Semantic Framework, on the other hand, allow working with the

entities completely independently of the persistence context from which they were retrieved because they store the data in the actual objects.

6.6. GC6 – Code/ontology generator

Empire, KOMMA, RDFBeans, the Semantic Framework and Spira do not contain any generator capable of creating entity classes from ontology schema or vice versa. ActiveRDF and SuRF do not contain code generators either, but they do not actually use any static object model and all entity classes are generated on demand at runtime.

AliBaba, AutoRDF, JAOB, and JOPA contain generators able to build an object model from the ontology. The AliBaba generator supports both RDFS and OWL, but it is less configurable. AutoRDF supports several OWL constructs, but its core is RDFS-based. On the other hand, the generators in JAOB and JOPA are OWL-based and expect the ontology to contain declarations of OWL classes and object, data and annotation properties.

JAOB is the only library with a generator capable of creating an ontology schema based on an object model. This generator is able to create classes and their hierarchies plus OWL data and object property declarations with domains and ranges and information about whether the property is functional or not [11].

6.7. OC1 – Explicit inference treatment

JOPA is the only library to explicitly treat inferred statements. It takes the safest but most restrictive approach – it makes attributes containing inferred information read-only. This prevents the user from executing operations with undefined results, like removing an inferred attribute value but it also disallows any additive changes to the attribute value.

6.8. OC2 – Named graphs

Alibaba, AutoRDF, JAOB, RDFBeans, the Semantic Framework, Spira, and SuRF are not aware of RDF named graphs and work only with the default graph.

Empire supports named graphs to a limited extent. Named graph access is specified via the @NamedGraph annotation, which has two modes of operation. The first mode leads to each instance of the annotated class being stored in its own named graph. The second mode then allows to specify an IRI of a named graph into which all instances of the annotated class are saved. JOPA and ActiveRDF, on the other hand, allow one to specify a named graph per instance and for individual attributes. RDFReactor uses model instances which represent individual named graphs of the repository. Each entity is then associated with a single model.

6.9. OC3 – Automatic provenance management

Unfortunately, none of the evaluated libraries supports automatic provenance management. The main reason is arguably the fact that there is no standardized API which would allow the application to pass user context information (session) to the OTM framework in an automated way. Without this context, the OTM provider is not able to record real-time provenance data like username or user profile IRI.

6.10. MC1 – Inheritance mapping

Inheritance mapping is probably the most complex feature in the comparison framework, with many subtle issues. Despite this, several evaluated libraries take a straightforward approach which can often lead to unexpected results.

Consider Empire which does support multiple inheritance in that it is able to work with classes which inherit mapped getters/setters from multiple interfaces. However, it is unable to persist more than one type for an entity. So if each interface declares a mapped type, only one of them gets persisted. KOMMA and RDFBeans suffer from the same issue, i.e. they correctly interpret inherited attributes, but always persist only a single type. For example, let us have interfaces A and B, which are mapped to RDFS classes `ex:A` and `ex:B` respectively. Then, we declare an entity class C, which implements both A and B. Saving an object c of type C into the repository would result in either `ex:c rdf:type ex:A` or `ex:c rdf:type ex:B` being inserted, depending on the order of declarations in the implements clause of C, where `c` is mapped to `ex:c`. The Semantic Framework would also support multiple inheritance thanks to its use of Java classes and interfaces. However, the implementation extracts only attributes declared in a particular class, without regard for any fields inherited from its superclass. This in effect means that any values of fields declared in a superclass are ignored when persisting an instance.

JAOB does not support inheritance mapping. SuRF allows to specify multiple superclasses when loading a resource but they do not represent mapped ontological

classes, they are regular Python classes adding custom behavior to the instance. On the other hand, similarly to ActiveRDF, since loaded instances contain attributes corresponding to all properties found on the resource, the concept of inheritance mapping does not really apply in this case. Appropriate type assertion presence on a resource can be of importance w.r.t. the type hierarchy, but such an issue is more closely related to the way libraries handle explicit and inferred statements.

JOPA currently supports only class-based inheritance, so mapping classes with multiple supertypes is not possible. The same issue applies to RDFReactor. Similarly, Spira supports only single inheritance because Ruby does not support multiple class inheritance. On the other hand, Spira allows an entity to specify multiple types, so multiple ontological classes can be combined into a single mapped Ruby class, which can be used as another class's parent.

AliBaba and AutoRDF are thus the only libraries fully supporting class hierarchy mapping. Since AliBaba is a Java library, it relies on interfaces to support multiple inheritance. AutoRDF is able to exploit the built-in support for multiple inheritance of C++.

6.11. MC2 – Unmapped data access

Providing access to unmapped properties is difficult in statically typed languages like Java or C++. AliBaba, AutoRDF, Empire, JAOB, KOMMA, RDFBeans and the Semantic Framework do not support such a feature.

JOPA and RDFReactor, although being Java frameworks, do attempt to supply this access. In JOPA, there are two relevant attribute annotations – a @Types attribute allows to read and modify an instance's ontological types and @Properties denotes a map of property-value pairs, representing the properties not mapped by the rest of the entity attributes. RDFReactor solves the issue by making all entity classes implement the ReactorBase interface (actually, the classes usually extend the ReactorBaseImpl abstract class) whose API contains generic methods for setting and retrieving arbitrary properties attached to the instance. Internally, even the mapped attributes make use of this API.

ActiveRDF and SuRF, due to their lack of a static model, naturally allow accessing all the properties of a resource. In Spira, all model objects can be manipulated also at the RDF statement level, where unmapped properties are accessible, thanks to the integration with RDF.rb.

6.12. MC3 – RDF collections and containers

Support for RDF collections and containers is scarce among the evaluated libraries. The only libraries which fully implement this feature are AliBaba and RDFBeans. AliBaba supports RDF lists and RDF containers. Both are mapped to instances of java.util.List. If the list root has the type `rdf:List`, it is treated correspondingly, otherwise, it is assumed to be a RDF container. This root type management, however, has to be done on RDF statement level, no dedicated API is present for it in AliBaba. RDFBeans also allows to work with both RDF containers and lists. The type of the target RDF construct is specified via Java annotations.

The approaches of the Semantic Framework and JOPA may be regarded as partial support. The Semantic Framework automatically stores array and java.util.List attributes as RDF sequences, without any way to configure this behavior. JOPA supports the OWL design pattern of linked lists and linked lists with content nodes [77], but it does not support any of the RDF containers. The type of the list is specified via an annotation.

7. Performance comparison

All the benchmark operations revolve around an object graph whose template is shown in Figure 4. As can be seen, there is one central OccurrenceReport object connected to an Occurrence, which is the root of a three level-deep balanced binary tree of events. Each report has also a set of three Resources attached and is connected to a randomly selected pre-existing author and last editor (instances of Person).

7.1. Experiment setup

The experiments were conducted on a machine with the following setup:

- OS: Linux Mint 17.3 64-bit
- CPU: Intel Core i5-750 2.67 GHz (4 cores)
- RAM: 16 GB
- Disk: Samsung SSD 850 EVO 250 GB
- Java: Java HotSpot JDK[22] 8u161, 64-bit

[22] Java Development Kit

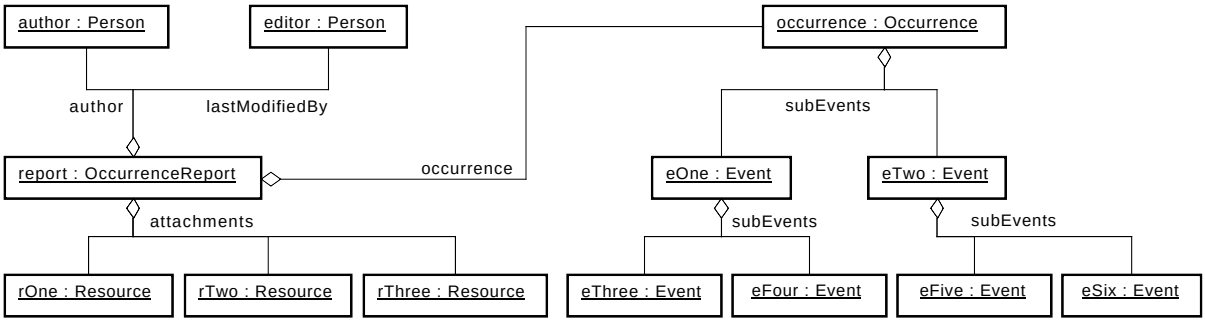


Fig. 4. UML object diagram of the object model used by the benchmark.

As a storage, we decided to use *GraphDB 8.4.1 Free* running on the same machine as the benchmark to eliminate possible network lag. GraphDB [16] is an industry-grade semantic graph database supporting the RDF4J API. The free version is limited to two queries in parallel, but this is of no concern, since the benchmark is not multithreaded. To ensure that none of the evaluated libraries had an unfair advantage by being specifically optimized for GraphDB, an additional experiment on a native RDF4J storage was conducted. Its results showed similar relative differences between the OTM frameworks as the measurements on GraphDB. The latest available versions of the libraries were used. Since some of them were not present in the Maven Central Repository[23] (an online repository of Java libraries), they had to be built from source code locally. The versions were (as of 20^{th} March 2018) as follows:

- *AliBaba 2.1*, built locally, sources downloaded from https://bitbucket.org/openrdf/alibaba/,
- *Empire 1.0*, built locally, sources downloaded from https://github.com/mhgrove/empire,
- *JOPA 0.9.15*, retrieved from Maven Central,
- *KOMMA 1.3.3-SNAPSHOT*, built locally, sources retrieved from https://github.com/komma/komma,
- *RDFBeans 2.2*, retrieved from Maven Central.

7.1.1. Performance

The scalability of the performance benchmark allowed custom specification of the size of data being used in the comparison. A scaling factor of one was used, which resulted in the following specification for individual operations:

OP1 – Create 300 Persons to choose the authors and last editors from, 300 OccurrenceReport instances together with their associated object graphs.

OP2 – Batch Create The same setup as *OP1*.

OP3 – Retrieve 300 OccurrenceReports together with their object graphs and 300 Persons pre-persisted (each report is assigned a random author and last editor from the Person instances).

OP4 – Retrieve All The same setup as *OP3*.

OP5 – Update The same setup as *OP3*. One half of the reports (every odd one, to be precise) are updated.

OP6 – Delete The same setup as *OP3*. One half of the reports (again, every odd one) are deleted, together with their dependencies.

Each operation was executed in 120 rounds, where the first twenty rounds represented a warm-up period which was not measured and was intended for the Java virtual machine to stabilize and perform optimizations. The following 100 rounds were measured. This was executed in three separate runs, i.e. three executions of the JVM, to account for different ways the code could be optimized, as discussed in [30]. The whole benchmark, i.e. all runs of all operations for all libraries, was executed automatically by a shell script. The repository was cleared after each round and GraphDB was restarted between the operations.

The benchmark was run for several values of heap size. The results should then reveal how sensitive the frameworks are to restricted heap size and how they scale as the heap size changes. The heap size was fixed (using the `-Xms` and `-Xmx` JVM configuration parameters) to:

- 32 MB
- 64 MB
- 128 MB
- 256 MB
- 512 MB

[23] http://search.maven.org/, accessed 2018-01-08.

– 1 GB

The maximum number of statements generated by the benchmark for *OP3* (the other operations are either the same or very similar) was 23 700 for Empire, JOPA, KOMMA and RDFBeans and 23 700 or 41 026 for AliBaba. The pair of numbers for AliBaba is due to the rather strange cascading strategy employed by AliBaba. When persisting an object, AliBaba automatically cascades the persist recursively to all references of the persisted entity. Such a strategy generates *blank nodes* for the referenced instances, leading to the 23 700 statements in total. If, on the other hand, one wanted to be able to denote the referenced instances (e.g. the occurrence or person) using a regular IRI, he would have to persist them separately. This, however, leads to a large duplication of data, because AliBaba will perform the cascading anyway, resulting in the 41 026 statements.

7.1.2. Memory

Measuring memory utilization turned out to be a bit tricky. Given the performance results, Java heap occupation could not have been measured throughout the execution of the performance benchmark, because it had a different duration for each library. Most of the literature concentrates on obtaining and analyzing heap dumps in the search for memory leaks (e.g., [55]) which was not the goal of this comparison.

In the end, a different strategy was used – a separate runner was developed which executed a series of CRUD operations similar to the ones used in the performance benchmark for a specific period of time. More precisely, the runner executed *OP1*, *OP4*, *OP5* and *OP6* in a sequence over and over until a specified timeout. This way, all the libraries were used for the same amount of time. To collect results, the Java virtual machine was configured to output detailed information about garbage collection events (using the `-XX:+PrintGCDetails` flag) and a fixed heap size was set, so that the application had a limited amount of memory available.

The particular setup for the memory benchmark was as follows:

– Runtime: 4 h,
– Heap size: 40 MB,
– Default garbage collector settings for the JVM used in the benchmark.

After the execution, *GCViewer*[24] was used to analyze the results and collect the following values:

– The number of young generation garbage collection events (YGC),
– The number of full garbage collection events (FGC),
– Accumulated garbage collection pause time (GCT),
– Application throughput.

To put these terms in context: since most objects in applications are short-lived, YGC is able to dispose of them quickly [55]. FGC events occur when the Java virtual machine needs to reclaim memory occupied by older objects and it is more demanding in terms of CPU time. Generally, an application benefits from as few FGC's as possible. Parts of both young generation and full garbage collection can happen in parallel to the application execution (if the CPU allows it), but sometimes the application has to be stopped so that the *garbage collector* can mark objects for disposal and remove them. It is these so-called *stop-the-world* events which are represented by GCT and which influence the application throughput. In particular, the throughput indicates how much of the total runtime the application spent actually performing its tasks, the rest being spent in garbage collection. More detailed explanation of garbage collection principles and strategies employed by the JVM can be found for example in [55, 78]. All in all, these values, together with the limited amount of available memory and the fixed execution time, offer a reasonable picture of how an application using these libraries is utilizing the available memory. A small heap size was used in order to put the garbage collector under more stress so that its overhead would be more noticeable. However, a 32 MB heap was not used since most evaluated libraries had issues using it (see Section 7.2.2).

7.2. Results

Let us now delve into the results of the performed experiments and attempt to interpret them. First, performance results are detailed, followed by a discussion of scalability of the evaluated libraries. Finally, memory utilization experiments are examined. Complete results can be found in Appendix A and also – together with the underlying data – online[25].

[24] https://github.com/chewiebug/GCViewer, accessed 2018-01-08.

[25] At https://kbss.felk.cvut.cz/web/kbss/otm-benchmark, accessed 2018-05-22.

7.2.1. Performance

In this part, relative performance differences between the evaluated libraries are discussed. As a representative example, results for benchmark with heap size set to 1 GB are visualized in a plot in Figure 5. Table 4 then complements the plot visualization with information on execution time mean, standard deviation and the 95 % confidence interval. Performance on other heap sizes will be discussed in Section 7.2.2 and complete results can be found in Appendix A.1.

All the libraries performed relatively consistently, with low deviation and the confidence interval size within 2 % of the execution time mean, as suggested by [30]. Only Empire exhibited larger standard deviation for the retrieval operations (*OP3* and *OP4*). The following paragraphs elaborate on individual operation results.

*Create (*OP1*) and batch create (*OP2*)* The character of *OP1* and *OP2* suggests that the libraries should perform better in *OP2*, because they have to execute only a single transaction compared to a linear number of transactions in *OP1*. The only library that defied this expectation and performed virtually the same in both *OP1* and *OP2* was Empire. This is due to the fact that Empire actually commits the transaction after each data modifying operation of its EntityManager, more precisely, after merge, persist and remove. Not only is this highly inefficient in the batch scenario, but it can also lead to incorrect behavior because it effectively prohibits a rollback of transactional changes or grouping multiple operations into one transaction. The rather poor performance of Empire is partially due to the fact that it iterates over all attributes of the object being persisted using Java *reflection* instead of relying on a *metamodel* (model of entity classes and their attributes) [26] built during application persistence initialization. Reflection is a powerful mechanism which allows a program to examine and modify itself and the environment in which it operates [79], but it represents a performance overhead. While middleware libraries in Java typically use reflection, it is important to be aware of its performance impacts and to avoid them when possible. The other factor in Empire's create performance is its strategy for checking that an instance does not already exist in the repository. It uses a SPARQL SELECT query with an unbound subject, predicate and object, filtering results by the subject being equal to the persisted individual's identifier. First, our experience with RDF4J indicates that executing SPARQL queries is in most cases slower than using the statement filtering API. Second, a simpler ASK query could have been used.

In both *OP1* and *OP2*, KOMMA performed the worst by a large margin. There are multiple factors participating in this result. One is that KOMMA generates classes and their implementation at runtime (more precisely, when they are first used), adding specific behavior to the getters/setters. The most significant factor, though, is that it attempts to remove old values from the repository whenever a setter is called on an entity. This strategy is sensible in the update scenario when one sets a new value of an attribute of a managed object, and it is actually used by AliBaba as well[26]. However, to persist an entity in KOMMA, it is necessary to call the createNamed method which returns an empty managed instance of the requested type and then use setters to initialize its attributes. Therefore, even when persisting a completely new object, KOMMA issues statement removal calls to the underlying storage, which has a detrimental effect on its performance.

RDFBeans performed slightly better than Empire. Looking at the implementation of its add method, two factors are of interest. One is that the method is *synchronized*. This incurs a slight performance overhead because Java has to manage the synchronization mechanism even for the single-threaded benchmark. And it is somewhat unexpected, given that the documentation of RDF4J specifically declares the RepositoryConnection, around which RDFBeans BeanManager is built, not to be thread-safe and recommends multi-threaded applications to obtain separate connections for each thread[27]. Second, RDFBeans uses the RDF4J API hasStatement method with a bound subject to check for a resource's existence in the repository. In contrast, JOPA uses the same method, but specifies the whole triple, checking whether the subject is an instance of the specified RDFS class. This is not only arguably faster, but it also allows to persist the same entity as an instance of different RDFS classes.

It can be seen that AliBaba outperforms the other libraries in both *OP1* and *OP2*. Especially the batch scenario (*OP2*), where it was able to perform a round in under one second, an order of magnitude faster than in *OP1*, not only illustrates the overhead of transactions, but also the overhead of the other OOM libraries.

[26] It turns out that portions of AliBaba's code base are reused in KOMMA.

[27] See https://tinyurl.com/repository-connection, accessed 2018-02-13.

Table 4

Results of the performance benchmark with all libraries running on a 1 GB heap. ($\bar{T}$) denotes mean execution time, (σ) standard deviation and CI_{95} 95 % confidence interval. O stands for the operation executed and L for the evaluated library. For each operation, the fastest library is underlined

O	*OP1 – Create*				
L	**AliBaba**	**Empire**	**JOPA**	**KOMMA**	**RDFBeans**
$\bar{T}$ (ms)	9 155.46	27 628.78	13 925.49	71 823.58	25 599.89
σ (ms)	174.77	124.39	148.02	646.03	137.00
CI_{95} (ms)	(9 135.68; 9 175.23)	(27 614.70; 27 642.86)	(13 908.74; 13 942.24)	(71 750.47; 71 896.68)	(25 584.38; 25 615.39)

O	*OP2 – Batch create*				
L	**AliBaba**	**Empire**	**JOPA**	**KOMMA**	**RDFBeans**
$\bar{T}$ (ms)	738.16	27 624.64	6 926.20	64 241.03	17 325.64
σ (ms)	65.98	176.91	184.14	835.19	121.09
CI_{95} (ms)	(730.70; 745.63)	(27 604.62; 27 644.66)	(6 905.36; 6 947.04)	(64 146.52; 64 335.54)	(17 311.97; 17 339.37)

O	*OP3 – Retrieve*				
L	**AliBaba**	**Empire**	**JOPA**	**KOMMA**	**RDFBeans**
$\bar{T}$ (ms)	19 838.94	36 632.24	8 748.13	48 581.99	59 486.47
σ (ms)	307.56	2 808.46	99.82	536.58	280.45
CI_{95} (ms)	(19 804.13; 19 873.74)	(36 314.44; 36 950.04)	(8 736.83; 8 759.42)	(48 521.27; 48 642.71)	(59 454.74; 59 518.21)

O	*OP4 – Retrieve all*				
L	**AliBaba**	**Empire**	**JOPA**	**KOMMA**	**RDFBeans**
$\bar{T}$ (ms)	21 811.98	35 852.54	8 855.95	49 211.94	57 521.18
σ (ms)	729.50	2 782.01	99.38	507.20	391.03
CI_{95} (ms)	(21 729.43; 21 894.53)	(35 537.73; 36 167.35)	(8 844.70; 8 867.19)	(49 154.54; 49 269.33)	(57 476.93; 57 565.43)

O	*OP5 – Update*				
L	**AliBaba**	**Empire**	**JOPA**	**KOMMA**	**RDFBeans**
$\bar{T}$ (ms)	11 127.57	19 154.26	11 896.90	52 817.48	12 802.06
σ (ms)	176.45	86.91	291.88	453.97	205.95
CI_{95} (ms)	(11 107.60; 11 147.53)	(19 144.43; 19 164.10)	(11 863.87; 11 929.93)	(52 766.11; 52 868.85)	(12 778.76; 12 825.37)

O	*OP6 – Delete*				
L	**AliBaba**	**Empire**	**JOPA**	**KOMMA**	**RDFBeans**
$\bar{T}$ (ms)	13 479.67	33 567.58	17 700.54	14 542.59	15 912.55
σ (ms)	216.33	739.66	144.05	281.12	106.94
CI_{95} (ms)	(13 455.19; 13 504.15)	(33 483.88; 33 651.28)	(17 684.23; 17 716.84)	(14 510.78; 14 574.40)	(15 900.45; 15 924.65)

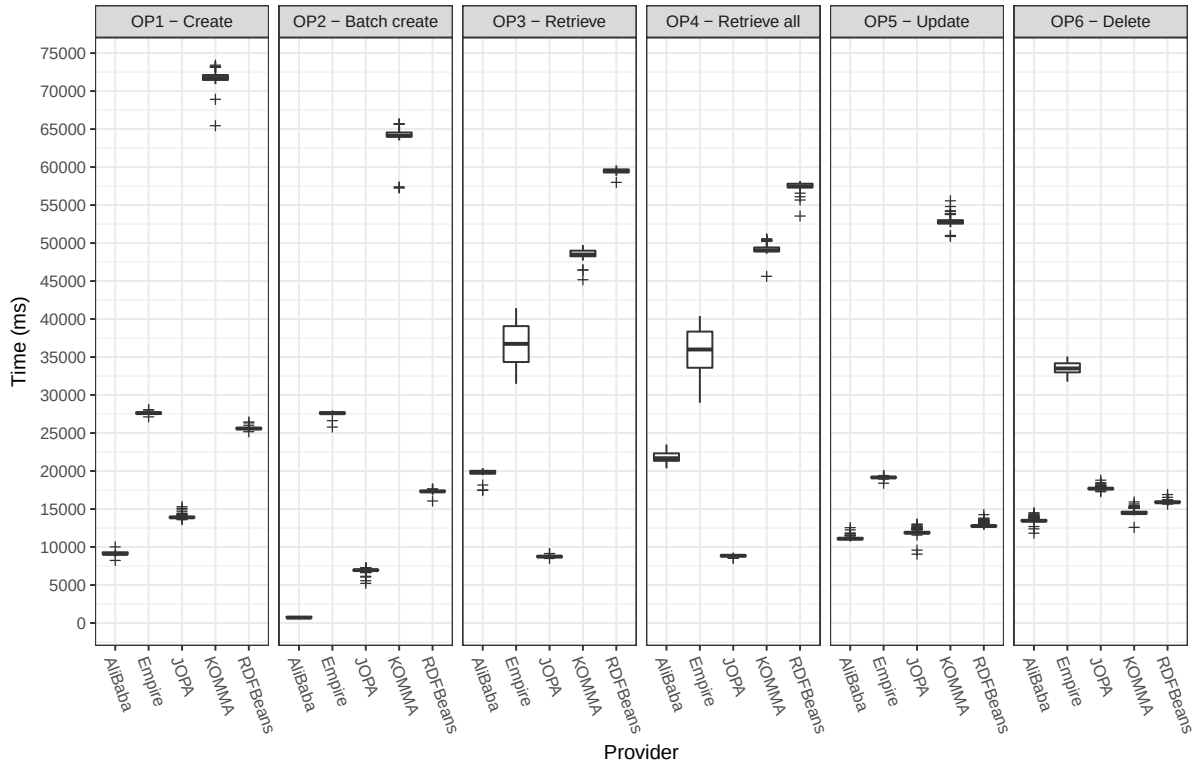


Fig. 5. Performance of the individual libraries on a 1 GB heap. The plots are grouped by the respective operations. Lower is better.

*Retrieve (*OP3*) and retrieve all (*OP4*)* While AliBaba was the fastest library in all other operations, it was outperformed by JOPA in the retrieval scenarios. Looking through the source code, all the libraries use different strategies for loading entity attributes.

AliBaba uses a single SPARQL SELECT query to get the subject and all the properties mapped by the entity class. To avoid failed joins, all the property patterns are wrapped in OPTIONAL. However, OPTIONAL increases the query complexity to PSPACE [80].

Empire adopts a different strategy. A SPARQL CONSTRUCT query is issued, retrieving a graph containing all statements concerning the specified subject. Then, it processes the attributes mapped by the target entity class and filters values for the corresponding properties from the constructed graph.

JOPA does not use SPARQL to retrieve entities for two reasons. One is our experience showing that filtering using the RDF4J API is more efficient. The other, and more important, is that a SPARQL query does not allow to specify whether inferred statements should be included or not and the result does not indicate the origin of the resulting statements (see *CR2* in Section 4). Instead, JOPA uses a single call to get all statements concerning the specified subject and then processes these statements, populating the resulting object with appropriate values. While this strategy may turn out inefficient in cases where the object model represents only a tiny portion of the graph around the loaded subject, it does sufficiently well in most cases. Indeed, the results show that it outperforms all the other strategies by a considerable margin.

KOMMA returns initially an empty object. When a getter is called, KOMMA loads the attribute value lazily using a SPARQL SELECT query. However, KOMMA has an internal cache of values, so it may happen that the attribute value is already cached and no query has to be performed. We argue that without it, KOMMA would be slower than RDFBeans, because using SPARQL queries for loading individual attributes is slower than using one larger query or the Sesame filtering API.

RDFBeans returns a fully initialized object, but it loads its attributes one by one using the RDF4J filtering API. Moreover, similarly to persist, it first checks for the subject's existence in the repository, which involves a rather costly statement filter with a bound subject. Thus, it was the slowest for both *OP3* and *OP4*.

*Update (*OP5*)* *OP5* showed comparable performance by AliBaba, JOPA, and RDFBeans, AliBaba outperforming the other libraries by less than one and two seconds respectively. KOMMA was again significantly slower. The update is also interesting in terms of the internal implementation in the particular libraries.

AliBaba, because it does not support detached objects, requires the user to first load the entity to be updated and then update it manually, essentially corresponding to the behavior Empire, JOPA and KOMMA implement internally. However, this can become extremely cumbersome – for instance, consider an object coming into the application in a PUT request to its REST API, signifying a data update. Then the application would either have to determine the changes by scanning the incoming object and comparing it to an existing object in the database, or use the incoming object as the new state and replace the existing data completely by iterating over all its attributes and merging their values into the repository. The approach of JOPA (see below) is much more flexible because it allows, for example, to specify whether the merge operation should be cascaded to objects referenced by the entity being merged.

Empire internally loads all the data concerning the subject which is being merged, removes them and then inserts the new, updated, data. While efficient, it is hardly the correct behavior. Consider a situation when the application is using a legacy RDF triple store and the object model covers only a part of the schema of the data. Invoking merge in Empire will not only delete the mapped data, but also all the other statements which have the same subject as the updated instance. The automatic commit of changes in Empire, which, given the setup of *OP5*, did not influence the performance but which can be harmful from the transactional behavior point of view nonetheless, should be also stressed.

JOPA and KOMMA internally load the instance into which updates will be merged (the same behavior is expected in JPA [26]). They then compare the merged instance with the loaded one and propagate the changes into the repository. The performance difference between these libraries essentially corresponds to their differences in instance loading. JOPA is not more efficient than AliBaba because of the rather complex machinery happening behind the scenes which is intended to provide behavior corresponding to JPA as much as possible. For instance, JPA specifies that when merging an entity with references to other objects, these references should not be merged and, actually, if they contain changes, these changes should be overwritten with the data loaded from the repository [26]. None of the other libraries behaves in this way.

RDFBeans works similarly to Empire in that it simply removes all statements concerning the updated subject and inserts new statements corresponding to the updated object. This, on the one hand, gives RDFBeans a significant performance boost, on the other hand, it can lead to incorrect behavior. As mentioned when discussing Empire, it can lead to the loss of data not mapped by the object model.

*Delete (*OP6*)* Except for Empire, all the libraries exhibited comparable performance for *OP6*.

AliBaba does not have a dedicated remove method, so entity removal in its case is a little more complicated. It consists of loading the object to be removed, setting all its attributes to null and then removing the class designation, i.e. the assertion of the instance's type.

The performance of Empire suffers from the fact that it uses the same strategy for loading statements as in the case of *OP3*, i.e. a SPARQL CONSTRUCT query. This query retrieves all statements with the specified subjects. Empire then removes these statements from the storage. Similar to update, this has the potentially harmful effect of removing statements which are not a part of the object model.

To avoid this issue, JOPA employs the *epistemic remove* strategy. Epistemic remove deletes only statements asserting properties mapped by the object model. Although less efficient than removing all statements concerning an individual, it is a safer option in terms of data consistency.

Unfortunately, AliBaba, Empire and JOPA do not deal with the situation where the removed resource is an object of other statements, i.e. it could be referenced by other instances (whether mapped by the object model or not). Relational databases resolve this using *referential integrity* which does not allow removal of a certain record as long as there are references to it in the form of *foreign keys*. In the open world of ontologies, a reference to an individual completely suffices to prove its existence, however, for an object model, this may not be (and often is not) the case. Therefore, OOM libraries should face this issue. In this regard, KOMMA and RDFBeans come with a solution.

Both KOMMA and RDFBeans perform entity removal by removing all statements whose subject *or* object is the resource being removed. This sufficiently

deals with the issue of referential integrity. On the other hand, this solution can lead to unintended data removal. For example, one may remove an instance without knowing by how many other resources (mapped or not) it is referenced. Another user can then load an object which formerly referenced the removed one and find that the connection has been severed for reasons unknown. We believe that this issue deserves a more thorough and systematic approach and it is one of the research issues considered for further development of JOPA.

7.2.2. Scalability

There are multiple ways in which the scalability of an application or a framework can be tested. In the case of this research, it was tested how well the libraries scale with regards to the memory available to the application.

Most of the libraries had issues with running on the smallest, 32 MB heap. The only framework able to perform the whole benchmark on a 32 MB heap was RDFBeans. AliBaba and JOPA were able to execute *OP1*, but failed on the other operations. Empire finished the *OP6* benchmark, but its standard deviation was so large that the results are inconclusive. KOMMA was not able to do any of the operations. In addition, Empire ran out of memory also for *OP4* on a 64 MB heap. The fact that it did not happen for *OP3*, which is similar to *OP4*, is likely due to the individual instances loaded by *OP3* being discarded right after the verification phase, whereas for *OP4* all the instances are loaded at once and then verified.

Contrary to the expectations, with increasing heap size, the benchmark runtime did not decrease. In fact, in some cases, it tended to increase slightly. This increase could be explained by the garbage collector having to manage a larger amount of memory. Overall, once the heap size passed the threshold after which the benchmark did not run out of memory, there was no definitive trend in terms of application performance with respect to the heap size. This can be seen in an example plot for *OP1* in Figure 6 (all plots regarding scalability w.r.t. heap size can be found in Appendix A.2).

To summarize, the benchmark developed as part of the OTM comparison framework appears not suitable for measuring scalability of the OTM libraries. In hindsight, this seems logical. Concerning scalability w.r.t. heap size, once the benchmark data fit into the available memory, increasing the heap size brings no performance benefit. Especially since garbage collection

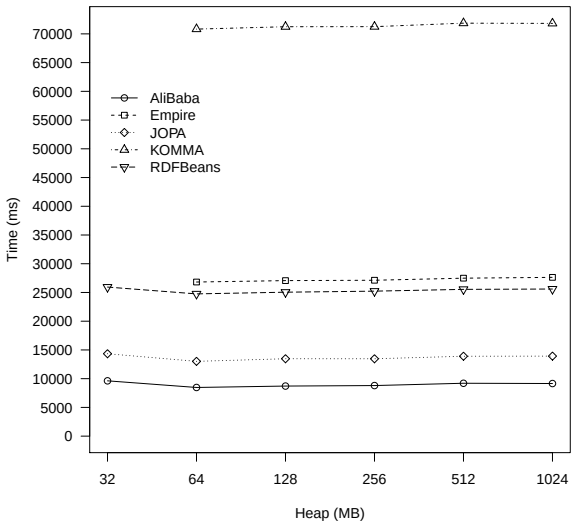


Fig. 6. Benchmark performance of *OP1* w.r.t. heap size. Lower is better.

is triggered after the setup of each round. A more useful scalability comparison could involve a variable number of concurrent clients. However, this is not supported by the presented benchmark. On the other hand, a concurrent access-based benchmark would have to be carefully designed in terms of data correctness and objectivity and a different triple store would have to be used.

7.2.3. Memory

Memory utilization is summarized in Table 5. It shows that both KOMMA and RDFBeans are relatively memory efficient, creating large numbers of short-lived objects which can be disposed of by the less-impactful young generation garbage collection. It is interesting that although being efficient, KOMMA was running out of memory when executing the performance benchmark on a 32 MB heap. AliBaba and JOPA required significantly more full garbage collections, but the application throughput remained over 96 %. This, of course, has to be considered also in the context of performance, where both AliBaba and JOPA clearly outperform KOMMA and RDFBeans. This indicates that AliBaba and JOPA use longer-living objects as a means to cache relevant data or to fetch the data more efficiently in terms of execution time.

Finally, the throughput of the benchmark application running Empire was approximately 85 %, significantly less than the other libraries. However, when we

tried the same experiment, but only with a 1 h runtime, its throughput was comparable to the other libraries. We suspect that there may be a memory leak in Empire. For the shorter runtime, the leak had not enough time to fully manifest. It would also explain Empire failing to execute *OP4* on a 64 MB heap.

8. Conclusions

We have introduced a novel framework for comparison of object-triple mapping libraries. It consists of a set of qualitative criteria describing features important for developers of domain-specific object-oriented applications and a performance benchmark written in Java. The framework was used to compare a diverse set of OTM libraries. The results indicate that significant differences exist between the evaluated OTM providers in terms of features, performance and their treatment of semantic data – for instance, some expect the object model to completely cover the schema of the data, others also support access to a subset of the schema.

Several conclusions may be derived from the comparison results. AliBaba demonstrated the best overall performance of the five evaluated libraries. However, it can only access Sesame-based storages and employs an unintuitive cascading strategy. Empire exhibited worse than average time performance and a potential memory leak in query mapping. On the other hand, its API directly implements portions of the JPA standard [26] which can make it suitable for existing projects migrating from relational to semantic databases. JOPA provides rich features as well as acceptable performance, memory utilization, and sound data access patterns for the general case. KOMMA, in contrast, performed the worst in all the operations but *OP6* and its API made it more difficult to use in the benchmark than the other libraries. RDFBeans is the most suitable library for environments with extremely limited memory. In comparison to the other four libraries evaluated in the performance benchmark, its time performance is, apart from retrieval operations, average.

The performance benchmark was designed so that adding new libraries into the comparison is relatively straightforward. However, attention must be paid to providing equal conditions to all compared libraries. For this reason, the evaluation provided in the paper concentrated on a single triple store implementation. The performance benchmark is published online and so are the results of the comparison presented in this work. We believe that these results might help potential users of OTM libraries in deciding which one to choose. In addition, the benchmark can be considered an example application for each of the libraries, complete with an object model declaration and the persistence layer setup and usage. Given the scarce documentation of most of the evaluated libraries, this can be a welcome benefit.

In the future, the benchmark should be extended to allow evaluating scalability w.r.t. multiple concurrent users, providing an even more realistic test case for the OTM frameworks.

Acknowledgements

This work was partially supported by grants No. GA 16-09713S Efficient Exploration of Linked Data Cloud of the Grant Agency of the Czech Republic and No.SGS16/229/OHK3/3T/13 Supporting ontological data quality in information systems of the Czech Technical University in Prague.

Table 5

Memory utilization summary. *YGC* (*FGC*) is young generation (full) garbage collection event count, *GCT* is the total time spent in garbage collection and *Throughput* is the application throughput

Measure	**AliBaba**	**Empire**	**JOPA**	**KOMMA**	**RDFBeans**
YGC	14 311	14 169	25 981	51 815	29 658
FGC	11 685	13 320	9 713	60	51
GCT (s)	496.98	1 326.58	448.82	73.75	40.4
Throughput (%)	96.37	84.56	96.88	99.49	99.72

Appendix A. Complete benchmark results

A.1. Performance

This section contains plots depicting performance of the evaluated libraries in the benchmark. Figure 7 shows results for a 32 MB heap, Figure 8 for a 64 MB heap, Figure 9 for a 128 MB heap, Figure 10 for a 256 MB heap, Figure 11 for a 512 MB heap and Figure 12 for a 1 GB heap. The tables contain the mean execution time, standard deviation and confidence intervals for each operation and each library. Each table represents benchmark results for a particular heap size. Table 6 contains results for a 32 MB heap, Table 7 for a 64 MB heap, Table 8 for a 128 MB heap, Table 9 for a 256 MB heap, Table 10 for a 512 MB heap, and Table 11 for a 1 GB heap.

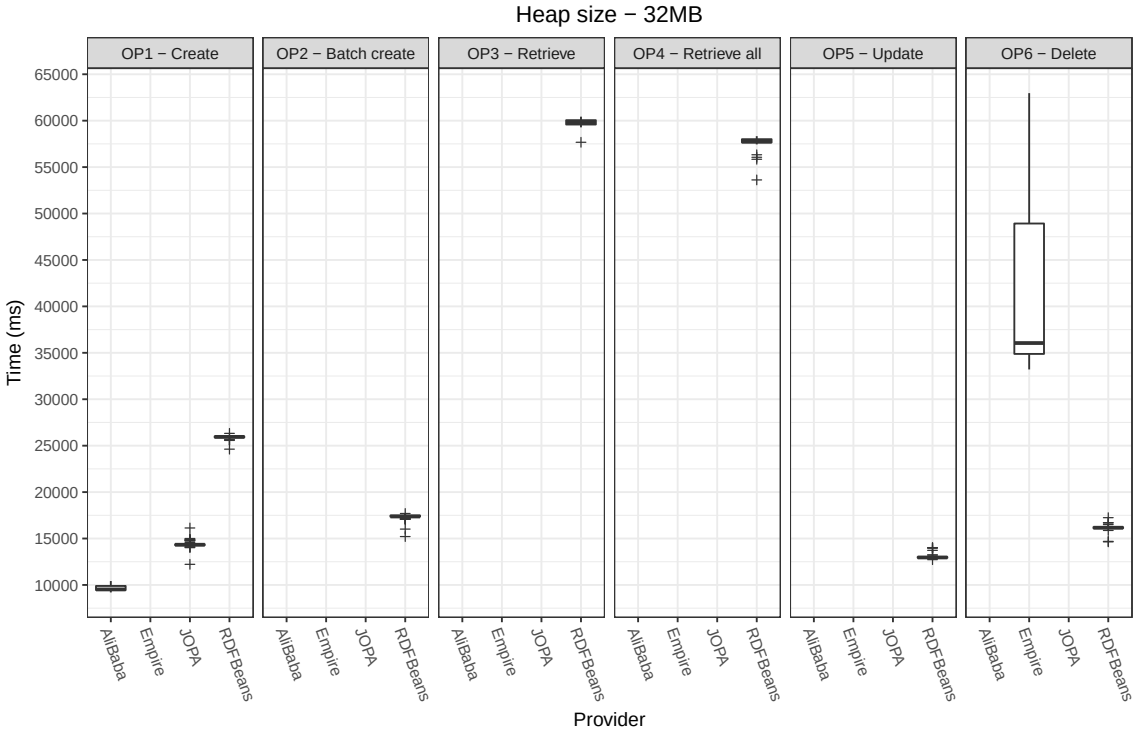


Fig. 7. Performance of the individual libraries on a 32 MB heap. The plots are grouped by the respective operations. Lower is better.

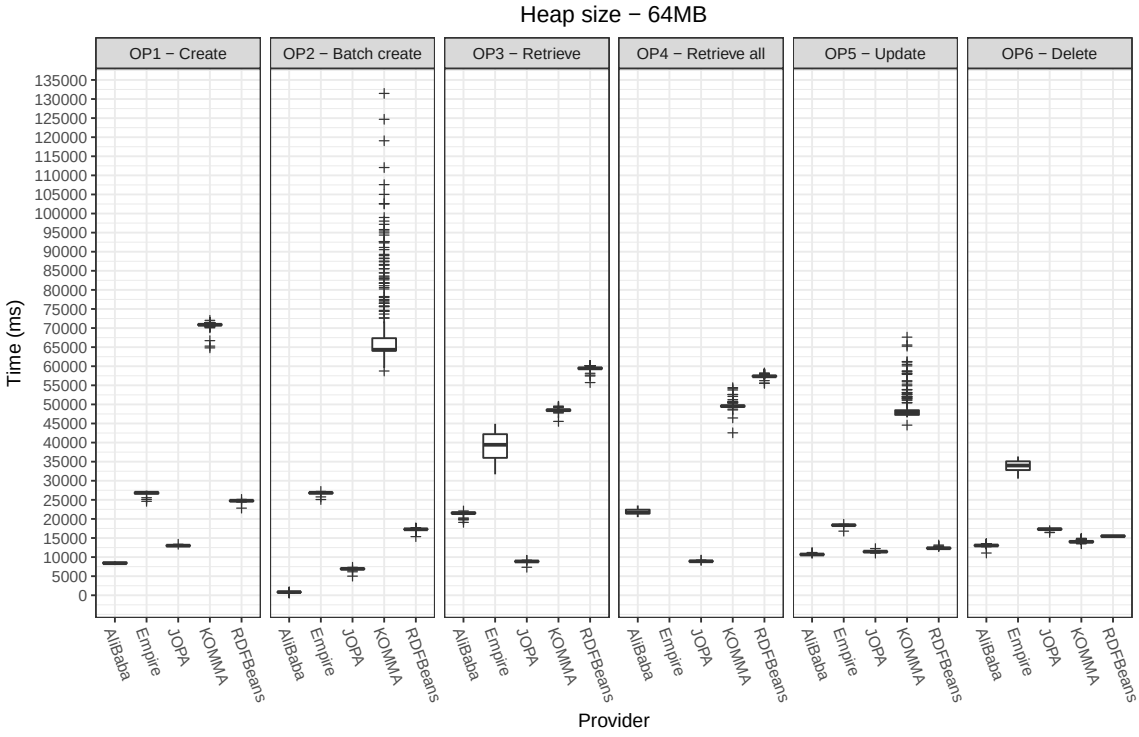


Fig. 8. Performance of the individual libraries on a 64 MB heap. The plots are grouped by the respective operations. Lower is better.

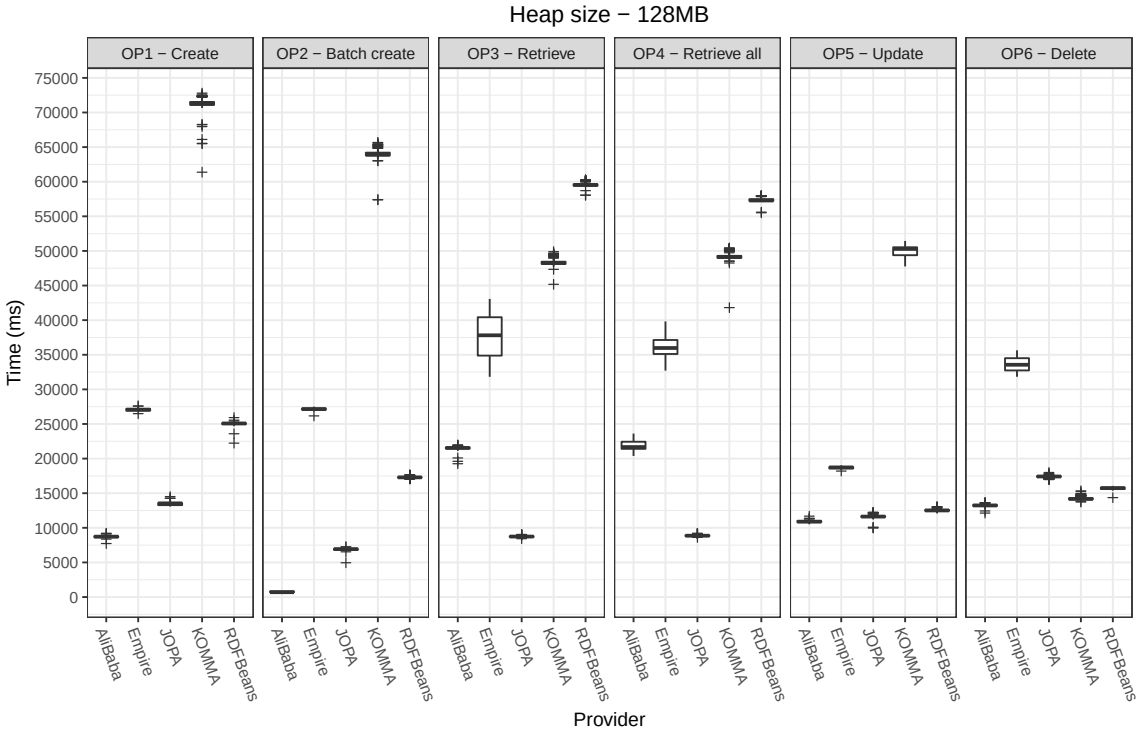


Fig. 9. Performance of the individual libraries on a 128 MB heap. The plots are grouped by the respective operations. Lower is better.

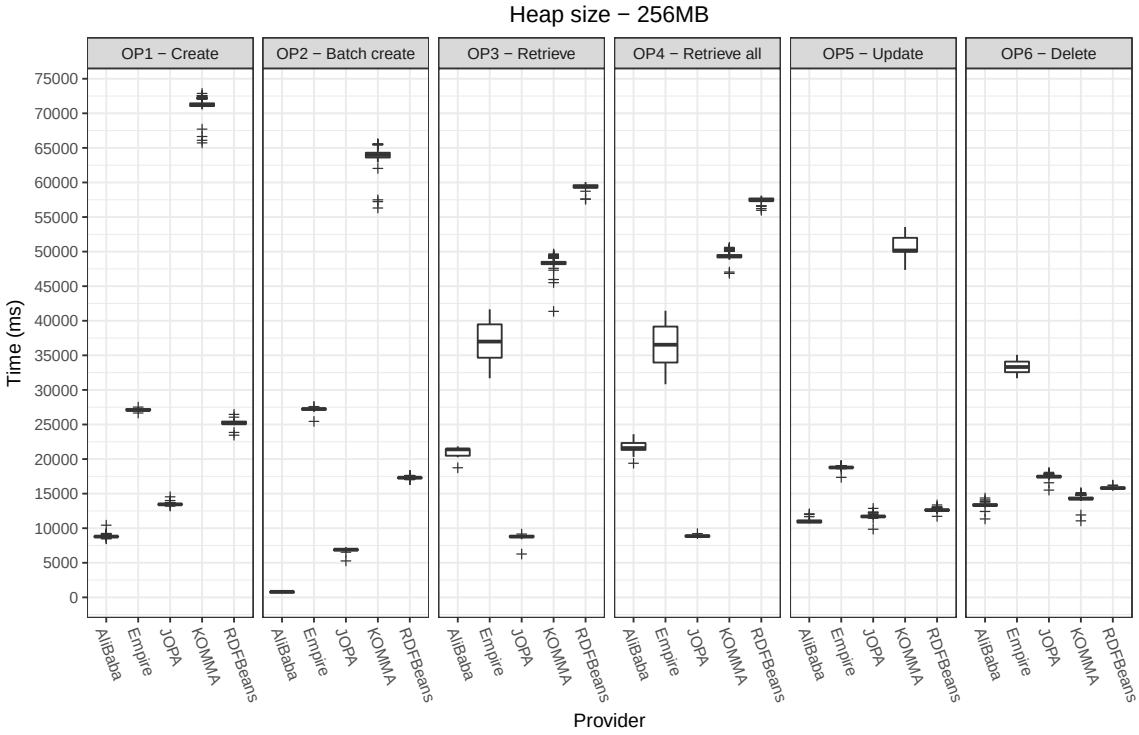


Fig. 10. Performance of the individual libraries on a 256 MB heap. The plots are grouped by the respective operations. Lower is better.

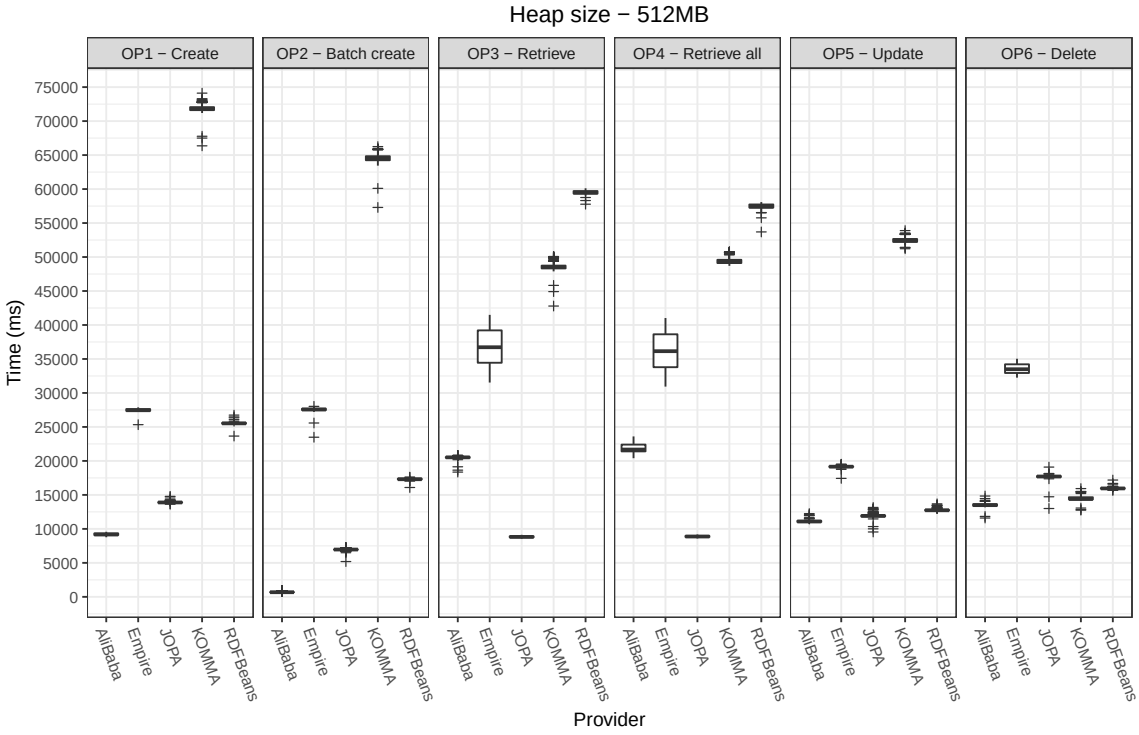


Fig. 11. Performance of the individual libraries on a 512 MB heap. The plots are grouped by the respective operations. Lower is better.

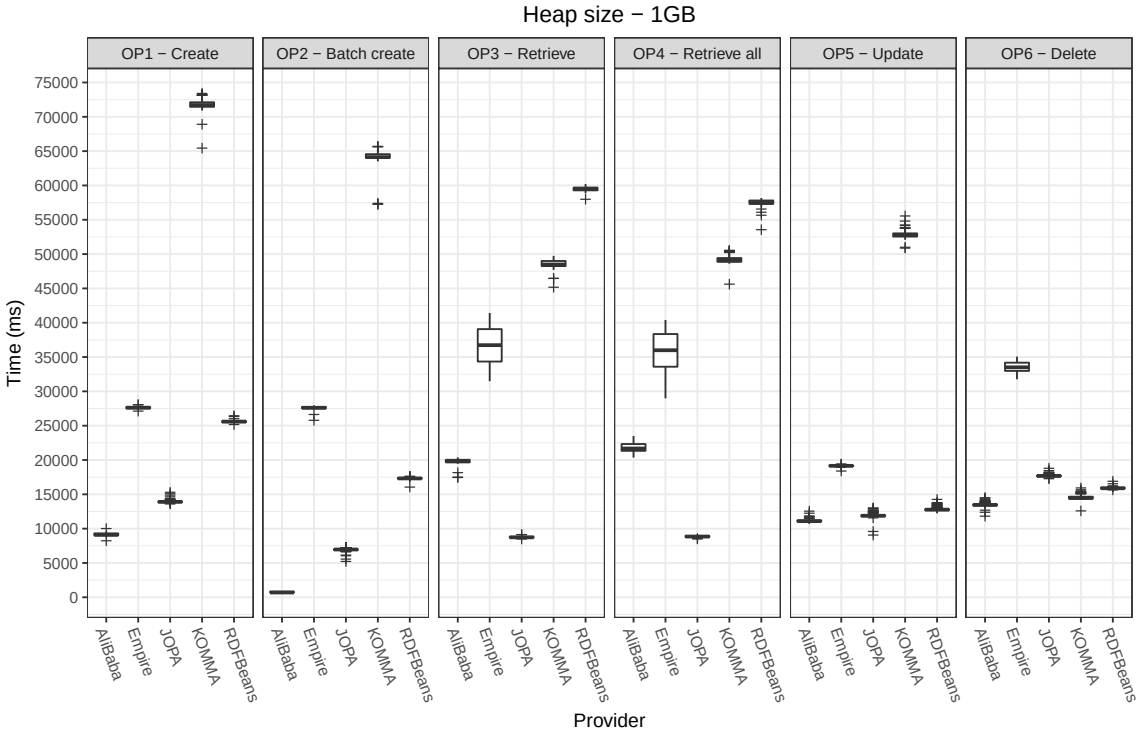


Fig. 12. Performance of the individual libraries on a 1 GB heap. The plots are grouped by the respective operations. Lower is better.

Table 6

Results of the performance benchmark with all libraries running on a 32 MB heap. ($\bar{T}$) denotes mean execution time, (σ) standard deviation and CI_{95} 95 % confidence interval. O stands for the operation executed and L for the evaluated library. For each operation, the fastest library is underlined

O	*OP1 – Create*				
L	**AliBaba**	**Empire**	**JOPA**	**KOMMA**	**RDFBeans**
$\bar{T}$ (ms)	9 624.75	×	14 337.39	×	25 940.71
σ (ms)	231.66	×	197.67	×	123.18
CI_{95} (ms)	(9 598.54; 9 650.97)	×	(14 315.02; 14 359.75)	×	(25 926.77; 25 954.65)

O	*OP2 – Batch create*				
L	**AliBaba**	**Empire**	**JOPA**	**KOMMA**	**RDFBeans**
$\bar{T}$ (ms)	×	×	×	×	17 398.23
σ (ms)	×	×	×	×	181.18
CI_{95} (ms)	×	×	×	×	(17 377.73; 17 418.73)

O	*OP3 – Retrieve*				
L	**AliBaba**	**Empire**	**JOPA**	**KOMMA**	**RDFBeans**
$\bar{T}$ (ms)	×	×	×	×	59 810.57
σ (ms)	×	×	×	×	304.38
CI_{95} (ms)	×	×	×	×	(59 776.13; 59 845.01)

O	*OP4 – Retrieve all*				
L	**AliBaba**	**Empire**	**JOPA**	**KOMMA**	**RDFBeans**
$\bar{T}$ (ms)	×	×	×	×	57 778.39
σ (ms)	×	×	×	×	362.33
CI_{95} (ms)	×	×	×	×	(57 737.39; 57 819.39)

O	*OP5 – Update*				
L	**AliBaba**	**Empire**	**JOPA**	**KOMMA**	**RDFBeans**
$\bar{T}$ (ms)	×	×	×	×	12 982.18
σ (ms)	×	×	×	×	142.54
CI_{95} (ms)	×	×	×	×	(12 966.05; 12 998.31)

O	*OP6 – Delete*				
L	**AliBaba**	**Empire**	**JOPA**	**KOMMA**	**RDFBeans**
$\bar{T}$ (ms)	×	40 753.17	×	×	16 169.85
σ (ms)	×	8 865.69	×	×	168.68
CI_{95} (ms)	×	(39 749.95; 41 756.40)	×	×	(16 150.77; 16 188.94)

Table 7

Results of the performance benchmark with all libraries running on a 64 MB heap. ($\bar{T}$) denotes mean execution time, (σ) standard deviation and CI_{95} 95 % confidence interval. O stands for the operation executed and L for the evaluated library. For each operation, the fastest library is underlined

O	*OP1 – Create*				
L	**AliBaba**	**Empire**	**JOPA**	**KOMMA**	**RDFBeans**
$\bar{T}$ (ms)	8 469.15	26 816.89	13 013.89	70 829.33	24 752.90
σ (ms)	180.32	271.77	122.68	571.92	168.57
CI_{95} (ms)	(8 448.77; 8 489.56)	(26 786.14; 26 847.64)	(13 000.01; 13 027.77)	(70 764.62; 70 894.05)	(24 733.82; 24 771.97)

O	*OP2 – Batch create*				
L	**AliBaba**	**Empire**	**JOPA**	**KOMMA**	**RDFBeans**
$\bar{T}$ (ms)	816.98	26 817.00	6 931.65	69 294.11	17 288.31
σ (ms)	54.10	183.64	164.25	11 064.78	146.06
CI_{95} (ms)	(810.86; 823.10)	(26 796.22; 26837.78)	(6 913.06; 6 950.24)	(68 042.03; 70 546.18)	(17 271.78; 17304.84)

O	*OP3 – Retrieve*				
L	**AliBaba**	**Empire**	**JOPA**	**KOMMA**	**RDFBeans**
$\bar{T}$ (ms)	21 531.18	39 095.80	8 872.86	48 472.58	59 460.36
σ (ms)	286.97	3 620.09	137.54	306.42	353.73
CI_{95} (ms)	(21 498.71; 21 563.66)	(38 686.15; 39 505.44)	(8 857.29; 8 888.42)	(48 437.90; 48 507.25)	(59 420.33; 59 500.39)

O	*OP4 – Retrieve all*				
L	**AliBaba**	**Empire**	**JOPA**	**KOMMA**	**RDFBeans**
$\bar{T}$ (ms)	21 904.37	×	8 917.77	49 603.73	57 410.77
σ (ms)	725.06	×	98.81	752.99	260.54
CI_{95} (ms)	(21 822.33; 21 986.42)	×	(8 906.59; 8 928.95)	(49 518.52; 49 688.94)	(57 381.28; 57 440.25)

O	*OP5 – Update*				
L	**AliBaba**	**Empire**	**JOPA**	**KOMMA**	**RDFBeans**
$\bar{T}$ (ms)	10 694.26	18 356.25	11 426.66	49 272.53	12 339.22
σ (ms)	133.35	118.52	136.96	3 995.99	109.96
CI_{95} (ms)	(10 679.17; 10 709.35)	(18 342.84; 18 369.66)	(11 411.17; 11 442.16)	(48 820.35; 49 724.71)	(12 326.77; 12 351.66)

O	*OP6 – Delete*				
L	**AliBaba**	**Empire**	**JOPA**	**KOMMA**	**RDFBeans**
$\bar{T}$ (ms)	13 052.14	33 955.28	17 326.18	14 083.81	15 491.79
σ (ms)	169.87	1 298.76	133.97	216.09	117.26
CI_{95} (ms)	(13 032.92; 13 071.36)	(33 808.31; 34 102.24)	(17 311.02; 17 341.34)	(14 059.36; 14 108.26)	(15 478.52; 15 505.06)

Table 8

Results of the performance benchmark with all libraries running on a 128 MB heap. ($\bar{T}$) denotes mean execution time, (σ) standard deviation and CI_{95} 95 % confidence interval. O stands for the operation executed and L for the evaluated library. For each operation, the fastest library is underlined

O	*OP1 – Create*				
L	**AliBaba**	**Empire**	**JOPA**	**KOMMA**	**RDFBeans**
$\bar{T}$ (ms)	8 713.70	27 064.14	13 469.14	71 241.91	25 048.47
σ (ms)	127.39	189.82	199.92	935.53	225.96
CI_{95} (ms)	(8 699.28; 8 728.11)	(27 042.66; 27 085.62)	(13 446.52, 13 491.77)	(71 136.05; 71 347.77)	(25 022.90; 25 074.04)

O	*OP2 – Batch create*				
L	**AliBaba**	**Empire**	**JOPA**	**KOMMA**	**RDFBeans**
$\bar{T}$ (ms)	739.42	27 153.00	6 898.00	63 977.56	17 304.30
σ (ms)	72.39	159.27	160.00	665.76	85.99
CI_{95} (ms)	(731.23; 747.61)	(27 134.97; 27 171.02)	(6 879.90; 6 916.11)	(63 902.22; 64 052.89)	(17 294.57; 17 314.03)

O	*OP3 – Retrieve*				
L	**AliBaba**	**Empire**	**JOPA**	**KOMMA**	**RDFBeans**
$\bar{T}$ (ms)	21 527.06	37 651.89	8 741.72	48 314.09	59 569.55
σ (ms)	226.91	3 184.72	96.87	390.20	268.05
CI_{95} (ms)	(21 501.38; 21 552.73)	(37 291.51; 38 012.26)	(8 730.76; 8 752.68)	(48 269.94; 48 358.25)	(59 539.22; 59 599.89)

O	*OP4 – Retrieve all*				
L	**AliBaba**	**Empire**	**JOPA**	**KOMMA**	**RDFBeans**
$\bar{T}$ (ms)	21 875.11	36 230.50	8 871.37	49 178.87	57 370.60
σ (ms)	784.56	1 752.28	97.11	567.24	263.63
CI_{95} (ms)	(21 786.33; 21 963.89)	(36 032.21; 36 428.78)	(8 860.38; 8 882.36)	(49 114.68; 49 243.05)	(57 340.76; 57 400.43)

O	*OP5 – Update*				
L	**AliBaba**	**Empire**	**JOPA**	**KOMMA**	**RDFBeans**
$\bar{T}$ (ms)	10 906.91	18 693.81	11 621.43	50 080.73	12 524.04
σ (ms)	145.07	133.32	204.09	632.03	120.72
CI_{95} (ms)	(10 890.50; 10923.33)	(18 678.73; 18 708.90)	(11 598.34; 11 644.53)	(50 009.21; 50 152.25)	(12 510.38; 12 537.70)

O	*OP6 – Delete*				
L	**AliBaba**	**Empire**	**JOPA**	**KOMMA**	**RDFBeans**
$\bar{T}$ (ms)	13 231.23	33 625.23	17 412.55	14 225.75	15 717.66
σ (ms)	151.42	994.90	127.01	220.62	157.61
CI_{95} (ms)	(13 214.10; 13 248.36)	(33 512.65; 33 737.81)	(17 398.18; 17 426.93)	(14 200.78; 14 250.71)	(15 699.82; 15 735.49)

Table 9

Results of the performance benchmark with all libraries running on a 256 MB heap. ($\bar{T}$) denotes mean execution time, (σ) standard deviation and CI_{95} 95 % confidence interval. O stands for the operation executed and L for the evaluated library. For each operation, the fastest library is underlined

O	*OP1 – Create*				
L	**AliBaba**	**Empire**	**JOPA**	**KOMMA**	**RDFBeans**
$\bar{T}$ (ms)	8 801.82	27 126.05	13 461.65	71 255.72	25 229.68
σ (ms)	125.52	122.86	99.60	679.13	256.97
CI_{95} (ms)	(8 787.61; 8 816.02)	(27 112.15; 27 139.95)	(13 450.38; 13 472.92)	(71 178.87; 71 332.57)	(25 200.60; 25 258.75)

O	*OP2 – Batch create*				
L	**AliBaba**	**Empire**	**JOPA**	**KOMMA**	**RDFBeans**
$\bar{T}$ (ms)	782.93	27 229.57	6 882.30	63 937.70	17 311.32
σ (ms)	67.60	156.16	142.41	840.59	94.71
CI_{95} (ms)	(775.28; 790.58)	(27 211.89; 27 247.24)	(6 866.19; 6 898.42)	(63 842.58; 64 032.82)	(17 300.61; 17 322.04)

O	*OP3 – Retrieve*				
L	**AliBaba**	**Empire**	**JOPA**	**KOMMA**	**RDFBeans**
$\bar{T}$ (ms)	21 115.87	36 909.70	8 793.69	48 354.16	59 441.71
σ (ms)	494.35	2 868.89	187.92	571.01	294.53
CI_{95} (ms)	(21 059.93; 21 171.81)	(36 585.06; 37 234.34)	(8 772.40; 8 814.93)	(48 289.54; 48 418.77)	(59 408.38; 59 475.03)

O	*OP4 – Retrieve all*				
L	**AliBaba**	**Empire**	**JOPA**	**KOMMA**	**RDFBeans**
$\bar{T}$ (ms)	21 802.91	36 421.56	8 867.83	49 378.57	57 491.72
σ (ms)	770.91	3 013.25	123.43	388.12	271.83
CI_{95} (ms)	(21 715.68; 21 890.15)	(36 080.59; 36 762.54)	(8 853.86; 8 881.79)	(49 334.65; 49 422.49)	(57 460.96; 57 522.48)

O	*OP5 – Update*				
L	**AliBaba**	**Empire**	**JOPA**	**KOMMA**	**RDFBeans**
$\bar{T}$ (ms)	10 998.84	18 786.56	11 697.42	50 779.10	12 622.81
σ (ms)	195.20	110.43	171.51	1 118.16	126.15
CI_{95} (ms)	(10 976.75; 11 020.92)	(18 774.06; 18 799.05)	(11 678.02; 11 716.83)	(50 652.57; 50 905.63)	(12 608.53; 12 637.08)

O	*OP6 – Delete*				
L	**AliBaba**	**Empire**	**JOPA**	**KOMMA**	**RDFBeans**
$\bar{T}$ (ms)	13 360.98	33 359.14	17 467.42	14 332.69	15 806.75
σ (ms)	203.04	861.21	183.60	320.33	109.86
CI_{95} (ms)	(13 338.01; 13 383.96)	(33 261.69; 33 456.60)	(17 446.64; 17 488.19)	(14 296.44; 14 368.94)	(15 794.32; 15 819.19)

Table 10

Results of the performance benchmark with all libraries running on a 512 MB heap. ($\bar{T}$) denotes mean execution time, (σ) standard deviation and CI_{95} 95 % confidence interval. O stands for the operation executed and L for the evaluated library. For each operation, the fastest library is underlined

O	*OP1 – Create*				
L	**AliBaba**	**Empire**	**JOPA**	**KOMMA**	**RDFBeans**
$\bar{T}$ (ms)	9 201.87	27 488.23	13 904.15	71 863.39	25 540.07
σ (ms)	149.49	202.24	113.96	632.12	174.80
CI_{95} (ms)	(9 184.95; 9 218.79)	(27 465.34; 27 511.11)	(13.891.26; 13 917.05)	(71 791.86; 71 934.92)	(25 520.29; 25 559.85)

O	*OP2 – Batch create*				
L	**AliBaba**	**Empire**	**JOPA**	**KOMMA**	**RDFBeans**
$\bar{T}$ (ms)	711.93	27 568.86	6 948.82	64 554.25	17 323.95
σ (ms)	64.86	303.20	139.14	669.83	120.04
CI_{95} (ms)	(704.59; 719.27)	(27 534.55; 27 603.17)	(6 933.08; 6 964.57)	(64 478.45; 64 630.05)	(17 310.36; 17 337.53)

O	*OP3 – Retrieve*				
L	**AliBaba**	**Empire**	**JOPA**	**KOMMA**	**RDFBeans**
$\bar{T}$ (ms)	20 507.68	36 729.19	8 835.20	48 577.78	59 516.95
σ (ms)	208.02	2 815.72	103.19	622.38	272.68
CI_{95} (ms)	(20 484.14; 20 531.22)	(36 410.57; 37 047.81)	(8 823.52; 8 846.88)	(48 507.35; 48 648.21)	(59 486.09; 59 547.80)

O	*OP4 – Retrive all*				
L	**AliBaba**	**Empire**	**JOPA**	**KOMMA**	**RDFBeans**
$\bar{T}$ (ms)	21 905.64	36 096.34	8 885.67	49 432.99	57 470.96
σ (ms)	778.16	2 838.06	109.66	470.01	364.24
CI_{95} (ms)	(21 817.59; 21 993.70)	(35 775.19; 36 417.49)	(8 873.26; 8 898.08)	(49 379.80; 49 486.17)	(57 429.75; 57 512.18)

O	*OP5 – Update*				
L	**AliBaba**	**Empire**	**JOPA**	**KOMMA**	**RDFBeans**
$\bar{T}$ (ms)	11 121.83	19 154.90	11 926.78	52 461.46	12 765.91
σ (ms)	170.39	133.68	283.87	365.42	142.05
CI_{95} (ms)	(11 102.55; 11 141.11)	(19 139.77; 19 170.02)	(11 894.66; 11 958.90)	(52 420.11; 52 502.81)	(12 749.83; 12 781.98)

O	*OP6 – Delete*				
L	**AliBaba**	**Empire**	**JOPA**	**KOMMA**	**RDFBeans**
$\bar{T}$ (ms)	13 520.13	33 562.09	17 681.34	14 487.75	15 965.50
σ (ms)	243.72	768.82	350.42	309.69	122.67
CI_{95} (ms)	(13 492.55; 13 547.71)	(33 475.09; 33 649.09)	(17 641.69; 17 720.99)	(14 452.71; 14 522.80)	(15 951.61; 15 979.38)

Table 11

Results of the performance benchmark with all libraries running on a 1 GB heap. ($\bar{T}$) denotes mean execution time, (σ) standard deviation and CI_{95} 95 % confidence interval. O stands for the operation executed and L for the evaluated library. For each operation, the fastest library is underlined

O	*OP1 – Create*				
L	**AliBaba**	**Empire**	**JOPA**	**KOMMA**	**RDFBeans**
$\bar{T}$ (ms)	9 155.46	27 628.78	13 925.49	71 823.58	25 599.89
σ (ms)	174.77	124.39	148.02	646.03	137.00
CI_{95} (ms)	(9 135.68; 9 175.23)	(27 614.70; 27 642.86)	(13 908.74; 13 942.24)	(71 750.47; 71 896.68)	(25 584.38; 25 615.39)

O	*OP2 – Batch create*				
L	**AliBaba**	**Empire**	**JOPA**	**KOMMA**	**RDFBeans**
$\bar{T}$ (ms)	738.16	27 624.64	6 926.20	64 241.03	17 325.64
σ (ms)	65.98	176.91	184.14	835.19	121.09
CI_{95} (ms)	(730.70; 745.63)	(27 604.62; 27 644.66)	(6 905.36; 6 947.04)	(64 146.52; 64 335.54)	(17 311.97; 17 339.37)

O	*OP3 – Retrieve*				
L	**AliBaba**	**Empire**	**JOPA**	**KOMMA**	**RDFBeans**
$\bar{T}$ (ms)	19 838.94	36 632.24	8 748.13	48 581.99	59 486.47
σ (ms)	307.56	2 808.46	99.82	536.58	280.45
CI_{95} (ms)	(19 804.13; 19 873.74)	(36 314.44; 36 950.04)	(8 736.83; 8 759.42)	(48 521.27; 48 642.71)	(59 454.74; 59 518.21)

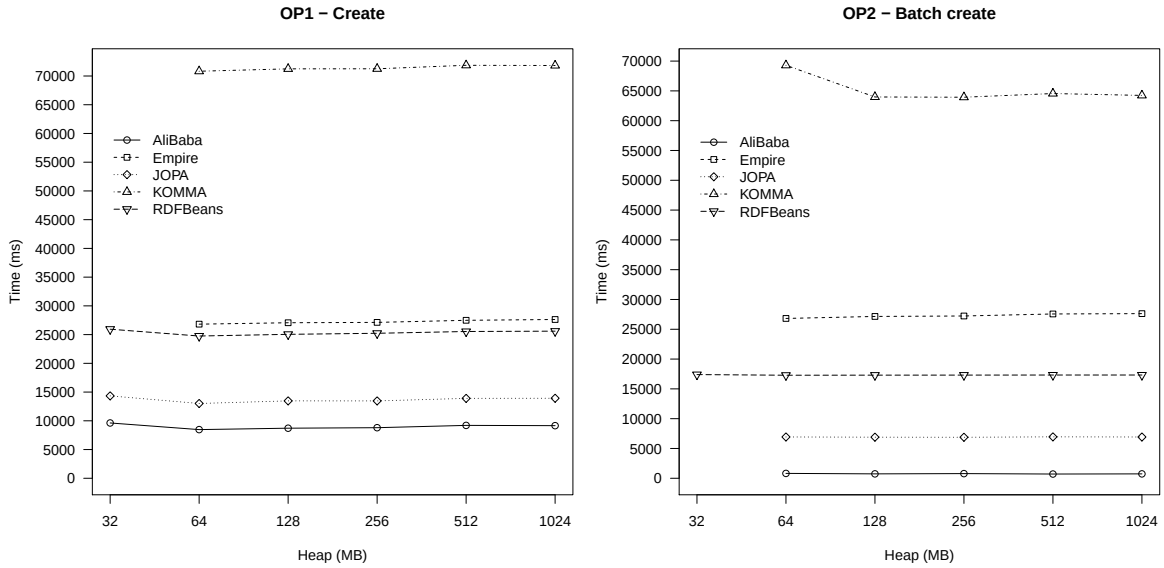
O	*OP4 – Retrive all*				
L	**AliBaba**	**Empire**	**JOPA**	**KOMMA**	**RDFBeans**
$\bar{T}$ (ms)	21 811.98	35 852.54	8 855.95	49 211.94	57 521.18
σ (ms)	729.50	2 782.01	99.38	507.20	391.03
CI_{95} (ms)	(21 729.43; 21 894.53)	(35 537.73; 36 167.35)	(8 844.70; 8 867.19)	(49 154.54; 49 269.33)	(57 476.93; 57 565.43)

O	*OP5 – Update*				
L	**AliBaba**	**Empire**	**JOPA**	**KOMMA**	**RDFBeans**
$\bar{T}$ (ms)	11 127.57	19 154.26	11 896.90	52 817.48	12 802.06
σ (ms)	176.45	86.91	291.88	453.97	205.95
CI_{95} (ms)	(11 107.60; 11 147.53)	(19 144.43; 19 164.10)	(11 863.87; 11 929.93)	(52 766.11; 52 868.85)	(12 778.76; 12 825.37)

O	*OP6 – Delete*				
L	**AliBaba**	**Empire**	**JOPA**	**KOMMA**	**RDFBeans**
$\bar{T}$ (ms)	13 479.67	33 567.58	17 700.54	14 542.59	15 912.55
σ (ms)	216.33	739.66	144.05	281.12	106.94
CI_{95} (ms)	(13 455.19; 13 504.15)	(33 483.88; 33 651.28)	(17 684.23; 17 716.84)	(14 510.78; 14 574.40)	(15 900.45; 15 924.65)

A.2. Scalability

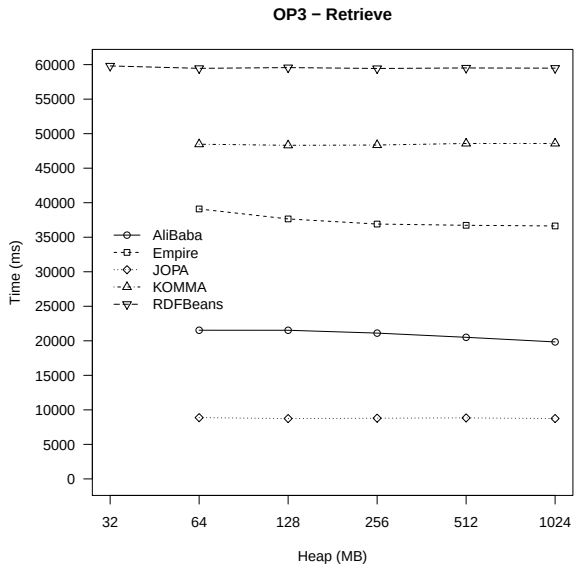
This section contains figures illustrating the scalability of the selected OTM libraries w.r.t. heap size for all operations. Figure 13 shows performance results depending on heap size for *OP1*, *OP2*, *OP3* and *OP4*. Figure 14 then for *OP5* and *OP6*.



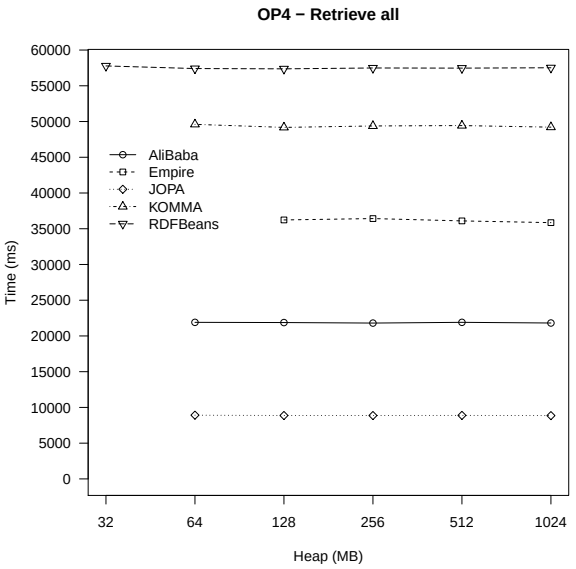
(a) Benchmark performance of *OP1* w.r.t. heap size.

(b) Benchmark performance of *OP2* w.r.t. heap size.

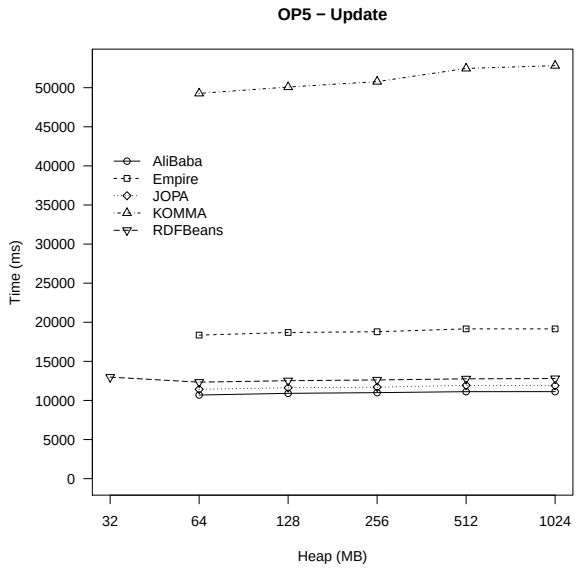
Fig. 13. Scalability w.r.t. heap size. Lower is better.



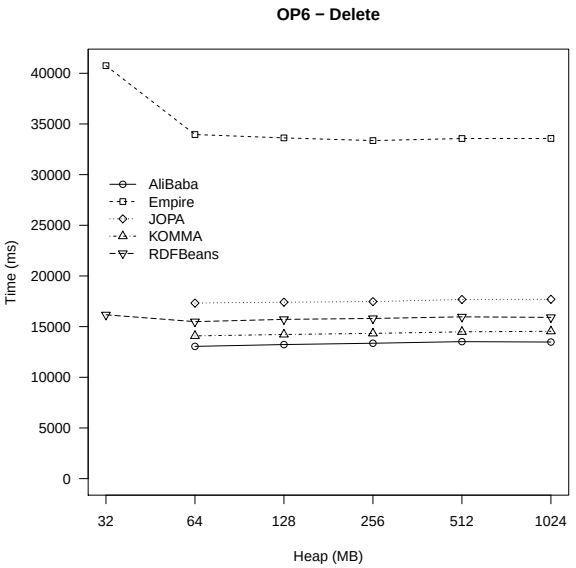
(a) Benchmark performance of *OP3* w.r.t. heap size.



(b) Benchmark performance of *OP4* w.r.t. heap size.



(c) Benchmark performance of *OP5* w.r.t. heap size.



(d) Benchmark performance of *OP6* w.r.t. heap size.

Fig. 14. Scalability w.r.t. heap size. Lower is better.

References

[1] T. Berners-Lee, J. Hendler and O. Lassila, The Semantic Web, *Scientific American* **284**(5) (2001), 28–37.

[2] A. Kalyanpur, B.K. Boguraev, S. Patwardhan, J.W. Murdock, A. Lally, C. Welty, J.M. Prager, B. Coppola, A. Fokoue-Nkoutche, L. Zhang, Y. Pan and Z.M. Qiu, Structured Data and Inference in DeepQA, *IBM J. Res. Dev.* **56**(3) (2012), 351–364, ISSN 0018-8646. doi:10.1147/JRD.2012.2188737. http://dx.doi.org/10.1147/JRD.2012.2188737.

[3] T. Heath and C. Bizer, *Linked Data: Evolving the Web into a Global Data Space*, 1st edn, Morgan & Claypool, 2011. ISBN 9781608454303. http://linkeddatabook.com/.

[4] D. Wood, M. Zaidman, L. Ruth and M. Hausenblas, *Linked Data: Structured Data on the Web*, Manning Publications Co., 2014.

[5] R.E. Carvalho, J. Williams, I. Sturken, R. Keller and T. Panontin, Investigation Organizer: the development and testing of a Web-based tool to support mishap investigations, in: *2005 IEEE Aerospace Conference*, 2005, pp. 89–98, ISSN 1095-323X. doi:10.1109/AERO.2005.1559302.

[6] R. Carvalho, S. Wolfe, D. Berrios and J. Williams, Ontology Development and Evolution in the Accident Investigation Domain, in: *2005 IEEE Aerospace Conference*, 2005, pp. 1–8, ISSN 1095-323X. doi:10.1109/AERO.2005.1559634.

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

[8] J. Broekstra, A. Kampman and F. van Harmelen, 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*, 2002, pp. 54–68.

[9] R. Cyganiak, D. Wood and M. Lanthaler, RDF 1.1 Concepts and Abstract Syntax, Technical Report, W3C, 2014.

[10] D. Brickley and R.V. Guha, RDF Schema 1.1, W3C Recommendation, W3C, 2014.

[11] S. Bechhofer, F. van Harmelen, J. Hendler, I. Horrocks, D.L. McGuinness, P.F. Patel-Schneider and L.A. Stein, OWL Web Ontology Language, W3C Recommendation, W3C, 2004.

[12] B. Motik, B. Parsia and P.F. Patel-Schneider, OWL 2 Web Ontology Language Structural Specification and Functional-Style Syntax, W3C Recommendation, W3C, 2009.

[13] D. Beckett, T. Berners-Lee, E. Prud'hommeaux and G. Carothers, RDF 1.1 Turtle, W3C Recommendation, W3C, 2014, https://www.w3.org/TR/turtle/, accessed 08-08-2017..

[14] S. Harris and A. Seaborne, SPARQL 1.1 Query Language, Technical Report, W3C, 2013.

[15] P. Gearon, A. Passant and A. Polleres, SPARQL 1.1 Update, Technical Report, W3C, 2013, Accessed 2018-01-08. https://www.w3.org/TR/sparql11-update.

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

[17] O. Erling and I. Mikhailov, RDF Support in the Virtuoso DBMS., in: *Conference on Social Semantic Web*, S. Auer, C. Bizer, C. Müller and A.V. Zhdanova, eds, LNI, Vol. 113, GI, 2007, pp. 59–68. ISBN 978-3-88579-207-9. http://dblp.uni-trier.de/db/conf/cssw/cssw2007.html#ErlingM07.

[18] Microsoft, Microsoft Open Database Connectivity (ODBC), Microsoft Corporation, 2016. https://cdn.simba.com/wp-content/uploads/2016/03/ODBC_specification.pdf.

[19] P. Křemen and Z. Kouba, Ontology-Driven Information System Design, *IEEE Transactions on Systems, Man, and Cybernetics: Part C* **42**(3) (2012), 334–344, ISSN 1094-6977.

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

[21] J. Leigh, AliBaba, 2007, Accessed 2018-01-02. https://bitbucket.org/openrdf/alibaba/.

[22] M. Grove, Empire: RDF & SPARQL Meet JPA, *semanticweb.com* (2010). http://semanticweb.com/empire-rdf-sparql-meet-jpa_b15617.

[23] M. Ledvinka and P. Křemen, JOPA: Accessing Ontologies in an Object-oriented Way, in: *Proceedings of the 17th International Conference on Enterprise Information Systems*, 2015.

[24] C. Puleston, B. Parsia, J. Cunningham and A. Rector, Integrating Object-Oriented and Ontological Representations: A Case Study in Java and OWL, in: *The Semantic Web - ISWC 2008*, A. Sheth, S. Staab, M. Dean, M. Paolucci, D. Maynard, T. Finin and K. Thirunarayan, eds, Springer Berlin Heidelberg, Berlin, Heidelberg, 2008, pp. 130–145. ISBN 978-3-540-88564-1.

[25] JCP, JDBC^{TM} 4.2 Specification, Oracle America, Inc., 2014.

[26] JCP, JSR 317: Java^{TM} Persistence API, Version 2.0, Sun Microsystems, 2009.

[27] Standard Performance Evaluation Corporation, SPECjbb®2015, 2015, Accessed 2018-01-04. https://www.spec.org/jbb2015/.

[28] S.M. Blackburn, R. Garner, C. Hoffmann, A.M. Khang, K.S. McKinley, R. Bentzur, A. Diwan, D. Feinberg, D. Frampton, S.Z. Guyer, M. Hirzel, A. Hosking, M. Jump, H. Lee, J.E.B. Moss, A. Phansalkar, D. Stefanović, T. VanDrunen, D. von Dincklage and B. Wiedermann, The DaCapo Benchmarks: Java Benchmarking Development and Analysis, in: *Proceedings of the 21st Annual ACM SIGPLAN Conference on Object-oriented Programming Systems, Languages, and Applications*, OOPSLA '06, ACM, New York, NY, USA, 2006, pp. 169–190. ISBN 1-59593-348-4. doi:10.1145/1167473.1167488. http://doi.acm.org/10.1145/1167473.1167488.

[29] G. Gousios, V. Karakoidas and D. Spinellis, Tuning Java's memory manager for high performance server applications, in: *Proceedings of the 5th International System Administration and Network Engineering Conference SANE 06*, A. Zavras, ed., Stichting SANE, 2006, pp. 69–83, NLUUG.

[30] A. Georges, D. Buytaert and L. Eeckhout, Statistically Rigorous Java Performance Evaluation, in: *Proceedings of the 22Nd Annual ACM SIGPLAN Conference on Object-oriented Programming Systems and Applications*, OOPSLA '07, ACM, New York, NY, USA, 2007, pp. 57–76. ISBN 978-1-59593-786-5. doi:10.1145/1297027.1297033. http://doi.acm.org/10.1145/1297027.1297033.

[31] O. Holanda, S. Isotani, I.I. Bittencourt, D. Dermeval and W. Alcantara, An Object Triple Mapping System Supporting Detached Objects, *Engineering Applications of Artificial Intelligence* **62**(C) (2017), 234–251, ISSN 0952-1976. doi:10.1016/j.engappai.2017.04.010. https://doi.org/10.1016/j.engappai.2017.04.010.

[32] P. Cristofaro, Virtuoso RDF Triple Store Analysis Benchmark & mapping tools RDF / OO, 2013, Accessed 2018-01-02. https://tinyurl.com/virt-rdf-map-tools.

[33] K. Schlegel, Selecting an RDF mapping library for cross-media enhancements, 2015, Accessed 2018-01-04. https://tinyurl.com/sel-rdf-map-lib.

[34] Y. Guo, A. Qasem, Z. Pan and J. Heflin, A Requirements Driven Framework for Benchmarking Semantic Web Knowledge Base Systems, *IEEE Transactions on Knowledge and Data Engineering* **19**(2) (2007), 297–309.

[35] Y. Guo, Z. Pan and J. Heflin, LUBM: A benchmark for OWL knowledge base systems, *Journal of Web Semantics* **3**(2–3) (2005), 158–182. http://dx.doi.org/10.1016/j.websem.2005.06.005;http://www.bibsonomy.org/bibtex/2924e60509d7e1b45c6f38eaef9a5c6bb/gromgull.

[36] C. Bizer and A. Schultz, The Berlin SPARQL benchmark, *International Journal On Semantic Web and Information Systems* **5**(2) (2009), 1–24.

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

[38] M. Ledvinka and P. Křemen, Object-UOBM: An Ontological Benchmark for Object-oriented Access, in: *Knowledge Engineering and the Semantic Web*, Springer, CCIS series, 2015.

[39] W.V. Siricharoen, Ontologies and Object models in Object Oriented Software Engineering, in: *Proceedings of the International MultiConference of Engineers and Computer Scientists 2006, IMECS '06, June 20-22, 2006, Hong Kong, China*, 2006, pp. 856–861.

[40] G. Weikum and G. Vossen, *Transactional Information Systems: Theory, Algorithms, and the Practice of Concurrency Control and Recovery*, Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2002. ISBN 9780080519562.

[41] L. Feigenbaum, G.T. Williams, K.G. Clark and E. Torres, SPARQL 1.1 Protocol, Technical Report, W3C, 2013, Accessed 2018-01-08. https://www.w3.org/TR/sparql11-protocol.

[42] S. Speicher, J. Arwe and A. Malhotra, Linked Data Platform 1.0, Technical Report, W3C, 2013, Accessed 2018-01-08. http://www.w3.org/TR/ldp/.

[43] G. Hillairet, F. Bertrand and J.Y. Lafaye, Rewriting Queries by Means of Model Transformations from SPARQL to OQL and Vice-Versa, in: *Theory and Practice of Model Transformations*, R.F. Paige, ed., Springer Berlin Heidelberg, Berlin, Heidelberg, 2009, pp. 116–131. ISBN 978-3-642-02408-5.

[44] R.G.G. Cattell and D.K. Barry, *The Object Data Standard: ODMG 3.0*, Morgan Kaufmann, 2000. ISBN 1-55860-647-5.

[45] C. Stadler and J. Lehmann, JPA Criteria Queries over RDF Data, in: *Workshop on Querying the Web of Data co-located with the Extended Semantic Web Conference*, 2017. http://jens-lehmann.org/files/2017/quweda_jpa.pdf.

[46] M. Zimmermann, Owl2Java – A Java Code Generator for OWL, 2009, Accessed 2018-01-02. http://www.incunabulum.de/projects/it/owl2java/.

[47] M. Quasthoff, H. Sack and C. Meinel, Can Software Developers Use Linked Data Vocabulary?, in: *Proceedings of International Conference on Semantic Systems 2009 (i-semantics 2009)*, 2009.

[48] D. Allemang and J. Hendler, *Semantic Web for the Working Ontologist: Effective Modeling in RDFS and OWL*, 2nd edn, Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2011. ISBN 9780123859655, 9780123859662.

[49] J.J. Carroll, C. Bizer, P. Hayes and P. Stickler, Named Graphs, Provenance and Trust, in: *Proceedings of the 14th International Conference on World Wide Web*, WWW '05, ACM, New York, NY, USA, 2005, pp. 613–622. ISBN 1-59593-046-9. doi:10.1145/1060745.1060835. http://doi.acm.org/10.1145/1060745.1060835.

[50] A. Zimmermann, RDF 1.1: On Semantics of RDF Datasets, 2014, Accessed 2018-01-04. https://www.w3.org/TR/rdf11-datasets/.

[51] S.S. Sahoo, O. Bodenreider, P. Hitzler, A. Sheth and K. Thirunarayan, Provenance Context Entity (PaCE): Scalable Provenance Tracking for Scientific RDF Data, in: *Proceedings of the 22Nd International Conference on Scientific and Statistical Database Management*, SSDBM'10, Springer-Verlag, Berlin, Heidelberg, 2010, pp. 461–470. ISBN 3-642-13817-9, 978-3-642-13817-1. http://dl.acm.org/citation.cfm?id=1876037.1876075.

[52] M. Quasthoff and C. Meinel, Tracing the Provenance of Object-Oriented Computations on RDF Data, in: *Proceedings of the 2nd Workshop (in conjunction with 7th ESWC) on Trust and Privacy on the Social and Semantic Web (SPOT 2010)*, 2010.

[53] P. Křemen, B. Kostov, M. Blaško, J. Ahmad, V. Plos, A. Lališ, S. Stojić and P. Vittek, Ontological Foundations of European Coordination Centre for Accident and Incident Reporting Systems, *Journal of Aerospace Information Systems* **14**(5) (2017), 279–292, https://doi.org/10.2514/1.I010441.

[54] L. Saeeda, Iterative Approach for Information Extraction and Ontology Learning from Textual Aviation Safety Reports, in: *The Semantic Web*, E. Blomqvist, D. Maynard, A. Gangemi, R. Hoekstra, P. Hitzler and O. Hartig, eds, Springer International Publishing, Cham, 2017, pp. 236–245. ISBN 978-3-319-58451-5.

[55] S. Oaks, *Java Performance: The Definitive Guide*, 1st edn, O'Reilly Media, Inc., 2014. ISBN 1449358454, 9781449358457.

[56] B. Kostov, J. Ahmad and P. Křemen, Towards Ontology-Based Safety Information Management in the Aviation Industry, in: *On the Move to Meaningful Internet Systems: OTM 2016 Workshops: Confederated International Workshops: EI2N, FBM, ICSP, Meta4eS, and OTMA 2016, Rhodes, Greece, October 24–28, 2016, Revised Selected Papers*, I. Ciuciu, C. Debruyne, H. Panetto, G. Weichhart, P. Bollen, A. Fensel and M.-E. Vidal, eds, Springer International Publishing, Cham, 2017, pp. 242–251. ISBN 978-3-319-55961-2. doi:10.1007/978-3-319-55961-2_25. https://doi.org/10.1007/978-3-319-55961-2_25.

[57] G. Guizzardi, Ontological Foundations for Structural Conceptual Models, PhD thesis, University of Twente, 2005.

[58] E. Oren, B. Heitmann and S. Decker, ActiveRDF: Embedding SemanticWeb data into object-oriented languages, *Web Semantics: Science, Services and Agents on the World Wide Web* **6**(3) (2008), ISSN 1570-8268. http://www.websemanticsjournal.org/index.php/ps/article/view/143.

[59] J. Frohn, G. Lausen and H. Uphoff, Access to objects by path expressions and rules, in: *Proceedings of the International*

Conference on Very Large Data Bases (VLDB), 1994, pp. 273–284.

[60] G. Stevenson and S. Dobson, Sapphire: Generating Java Runtime Artefacts from OWL Ontologies, in: *Advanced Information Systems Engineering Workshops*, C. Salinesi and O. Pastor, eds, Springer Berlin Heidelberg, Berlin, Heidelberg, 2011, pp. 425–436. ISBN 978-3-642-22056-2.

[61] P. Mika, Social Networks and the Semantic Web: The Next Challenge, *IEEE Intelligent Systems* **20**(1) (2005). http://www.cs.vu.nl/~pmika/research/papers/IEEE-TrendsAndControversies.pdf.

[62] F. Chevalier, AutoRDF - Using OWL as an Object Graph Mapping (OGM) Specification Language, in: *The Semantic Web*, H. Sack, G. Rizzo, N. Steinmetz, D. Mladenić, S. Auer and C. Lange, eds, Springer International Publishing, Cham, 2016, pp. 151–155. ISBN 978-3-319-47602-5.

[63] J. von Malottki, JAOB (Java Architecture for OWL Binding), 2008, Accessed 2018-01-02. http://wiki.yoshtec.com/jaob.

[64] J. Tao, E. Sirin, J. Bao and D.L. McGuinness, Integrity Constraints in OWL, in: *AAAI*, M. Fox and D. Poole, eds, AAAI Press, 2010.

[65] K. Wenzel, KOMMA: An Application Framework for Ontology-based Software Systems, *Semantic Web – Interoperability, Usability, Applicability* (2010).

[66] A. Alishevskikh, RDFBeans, 2017, Accessed 2018-01-02. https://rdfbeans.github.io.

[67] M. Völkel and Y. Sure, RDFReactor - From Ontologies to Programmatic Data Access, in: *Poster and Demo at International Semantic Web Conference (ISWC) 2005, Galway, Ireland*, 2005.

[68] P. Ježek and R. Mouček, Semantic framework for mapping object-oriented model to semantic web languages, *Frontiers in Neuroinformatics* **9**(3) (2015).

[69] T. Cowan and D. Donohue, Jenabean, 2010, Accessed 2018-01-04. https://code.google.com/archive/p/jenabean/.

[70] Ben Lavender, Spira: A Linked Data ORM for Ruby, 2010, Accessed 2018-03-20. https://github.com/ruby-rdf/spira.

[71] Cosmin Basca, SuRF, 2009, Accessed 2018-03-20. https://github.com/cosminbasca/surfrdf.

[72] B.H. Szekely and J. Betz, Jastor Homepage, 2006, Accessed 2018-01-02. http://jastor.sourceforge.net.

[73] H. Story, Semantic Object (Metadata) MappER, 2009, Accessed 2018-01-02. https://github.com/bblfish/sommer.

[74] D. Toti and M. Rinelli, RAN-Map: a system for automatically producing API layers from RDF schemas, *Journal of Ambient Intelligence and Humanized Computing* **8**(2) (2017), 291–299, ISSN 1868-5145. doi:10.1007/s12652-016-0394-z. https://doi.org/10.1007/s12652-016-0394-z.

[75] S. Fernández, D. Berrueta, M.G. Rodríguez and J.E. Labra, TRIOO: Keeping the semantics of data safe and sound into object-oriented software, in: *ICSOFT 2010 - Proceedings of the 5th International Conference on Software and Data Technologies*, Vol. 1, 2010, pp. 311–320.

[76] P.-H. Chiu, C.-C. Lo and K.-M. Chao, Integrating Semantic Web and Object-Oriented Programming for Cooperative Design, *Journal of Universal Computer Science* **15**(9) (2009), 1970–1990.

[77] N. Drummond, A. Rector, R. Stevens, G. Moulton, M. Horridge, H.H. Wang and J. Seidenberg, Putting OWL in Order: Patterns for Sequences in OWL, in: *OWL Experiences and Directions (OWLEd 2006)*, Athens Georgia, 2006.

[78] Oracle, Java Garbage Collection Basics, 2016, Accessed 2018-01-04. http://www.oracle.com/webfolder/technetwork/tutorials/obe/java/gc01/index.html.

[79] I.R. Forman and N. Forman, *Java Reflection in Action (In Action Series)*, Manning Publications Co., Greenwich, CT, USA, 2004. ISBN 1932394184.

[80] M. Schmidt, M. Meier and G. Lausen, Foundations of SPARQL Query Optimization, in: *Proceedings of the 13th International Conference on Database Theory*, ICDT '10, ACM, New York, NY, USA, 2010, pp. 4–33. ISBN 978-1-60558-947-3. doi:10.1145/1804669.1804675.