STUDIA INFORMATICA 2012
Volume 33 Number 2A (105)

Aleksander POHL
Jagiellonian University, Departament of Computational Linguistics

ROD – RUBY OBJECT DATABASE

Summary. Ruby Object Database is an open-source object database designed for storing and accessing data which rarely changes. The primary reason for designing it was to create a storage facility for natural language dictionaries and corpora. It is optimized for reading speed and easiness of usage.

Keywords: object database, Ruby, natural language processing

ROD – OBIEKTOWA BAZA DANYCH DLA JĘZYKA RUBY

Streszczenie. ROD (Ruby Object Database) jest otwartą, obiektową bazą danych zaprojektowaną do przechowywania i odczytywania danych, które rzadko ulegają zmianie. Podstawowym powodem jej utworzenia była chęć stworzenia bazy dla słowników oraz korpusów wykorzystywanych w przetwarzaniu języka naturalnego. Baza ta jest zoptymalizowana pod kątem szybkości odczytu danych oraz łatwości jej użycia.

Słowa kluczowe: obiektowa baza danych, Ruby, przetwarzanie języka naturalnego

1. Introduction

ROD (Ruby Object Database) is an open-source object database distributed under the MIT/X11[1] license and available at `http://github.com/apohllo/rod`. ROD is designed for storing and accessing data which rarely changes. It is an opposite of RDBMS as the data is not normalized, while "joins" are much faster. It is an opposite of in-memory databases, since it is designed to cover out of core data sets (10 GB and more). It is also an opposite of simple key-value stores, since it provides an expressive object-oriented interface.

[1] http://www.opensource.org/licenss/mit-license.php

The primary reason for designing it was to create storage facility for natural language dictionaries and corpora. The data in a fully fledged dictionary is interconnected in many ways, thus the relational model (joins) introduces unacceptable performance hit. The size of corpora forces them to be kept on disks. The in-memory databases like Redis [11] are not suited for large corpora. They would also require the data of a dictionary to be kept mostly in RAM, which is not needed (in most cases only a fraction of the data is used at the same time). And the last but not the least, the key-value stores however fast, provide an interface that is not expressive enough for Natural Language Processing tasks. That is why a storage facility which minimizes the number of disk reads and overcomes the defects of the mentioned storage systems was designed. The database is accessible via the Ruby language [8], which is both its data definition and data manipulation language. Thanks to its great expressiveness and true object-orientedness the data manipulation is done as easy as if a domain specific language was defined, still giving full access to a modern and very powerful programming language.

2. Motivation

The primary reason for designing ROD was to create a facility for storing linguistic data, which would be easily accessible from Ruby. This was motivated by the research in Natural Language Processing (NLP) and the lack of an object oriented database that would be suited for the specific NLP needs. The primary resources used in NLP are machine readable dictionaries and corpora – these resources have several features that are not very common in information technology in general.

First of all dictionaries rarely change – although natural languages evolve, this process is quite slow. Assuming that a dictionary has reached some maturity, it is not needed to update it more than several times a year. Similar situation concerns corpora – they are composed of texts, which are not modified after being incorporated. These types of resources tend to accumulate data rather than to change or remove their contents – in this respect they are similar to data warehouses.

However, the second feature of these resources makes them very different from data warehouses – that is the number of types of relations and number of relations that are registered for the data. If one wishes to build a decent dictionary for Polish one has to consider the following: the relation between a word form and its lemma (e.g. *dogs – dog***,** Polish has much more inflected forms than English – 14 in the case of a typical noun), the relation between a word and its senses (e.g. *grain – a small granular particle, a weight unit, a seed-like fruit*; Princeton WordNet 3.0 [2] registers 11 senses for *grain*), the different relations between senses of words (e.g. *hyperonymy*, *hyponymy*, *meronymy*, *holonymy*, *troponymy* etc.), the der-

ivational relations between words (e.g. *house – housing*) and similar. These might be further enriched with statistical data, so the data model is quite complex. In the case of corpora, the situation is similar – for a fully annotated corpus there would be many syntactic and semantic relations involved. If we used a relational database to store such a data, obtaining a full semantic or syntactic information, even for a single word form or a text segment would produce a very large number of table joins, causing an unacceptable performance. For this reason lexicographers tend to use SGML and (recently) XML to store dictionaries and textual data rather than relational databases.

The last important feature of these resources is their size. If it was relatively small, a good choice for such data would be in-memory databases like memcached[2] or Redis, which offer very good performance, even for highly interrelated data. But the size of dictionaries and corpora is often much larger than the available physical memory, even on modern machines. Size of a fully-fledged dictionary is two or three orders of magnitude larger than the number of entries (hundreds of thousands for words and millions for word forms), while sizes of the largest corpora are counted in tens or hundreds of gigabytes. On the other hand – the performance provided by in-memory databases is not really needed for these resources. Although it is good to have the largest dictionary and the largest corpus available, the data provided by them is never needed at-once. Usually only a small fraction of the data is processed at one moment, so it is not necessary to keep it in the operating memory. Only a good memory manager is needed: one that would keep the data that is often accessed in RAM and remove the rarely accessed data.

Such requirements are not very common in the other fields of information technology, thus there are not many general purpose storage systems that would suite them, the only exception being the graph databases. But on the other hand, the semi-structured graph model offered by these systems is too general for such needs – the data models of dictionaries and corpora, although evolve, are rather fixed, mostly due to the fact, that a change in the structure makes sense only if new data is available for the whole dictionary or corpus, obtaining which is usually very expensive.

Besides the features of the linguistic resources, the important factor taken into account when designing ROD was the language for accessing and processing the data. Instead of designing a new one like SQL or SPARQL, it was assumed that Ruby is a very good choice when it comes to navigate the data. If the database provided an object oriented interface with simple indexing, more complex queries could be easily expressed in that language due to its fully object oriented nature and simple, yet very powerful syntax.

[2] http://memcached.org/

3. Related Work

Designing a new data storage system should always have a good rationale, since there are so many storage system that there is a big chance that the problem at hand was already solved and it doesn't make sense to implement yet-another-home-made-storage-facility. On the other hand the number of the available solutions makes it quite hard to find the one that suites the best (which may change during the development of the client system). Thus the review of related work will focus on the systems used in Polish NLP as well as systems available for Ruby.

Finite state machines and finite state transducers are the primary means for obtaining taggings and lemmas from inflected word forms [1,10] (e.g. `pies+noun:plural:nominative` for *psy* (dogs)). The whole dictionary for a given language is transformed into finite state transducer, where each state transition corresponds to a letter in an analysed word form. The result of the analysis is a lemma or lemmas (in the case of ambiguity) of the word plus corresponding taggings. Since many word forms share some of the letter sequences, the information is much compressed and such a system works with very high performance characteristic for finite state machines.

For the specific task of providing lemma and tagging for a word form, finite state transducers seems to be the best option. But they fall short when it comes to provide more information about the word in question. The first problem is that the result of the analysis might be ambiguous – at least for Polish the lemma plus the tagging is not enough to distinguish words such as *rządy* (governments) and *rzędy* (rows), which both are nouns of the same gender and have the same lemma: *rząd*. The other problem is that if one wishes to obtain more than a lemma and a tagging (such as the senses of the word) the result of the analysis has to be parsed once again, which introduces significant processing cost. On the contrary in such a case an object database will return an object or objects that are distinguishable merely by their abstract ids and may provide any additional data via unified object interface (method call).

Another data stores used in NLP are engines build to store and efficiently query corpora, not only via key-words, but also via various features of the words. For example Poliqarp, a corpus engine build by IPI PAN [6], allows for storing large amounts of text and query them with Poliqarp query language by lemmas as well as by a specific part of speech or other morphosyntactic features such as gender, case or number.

Although Poliqarp provides quite expressive query language, its problem is similar to finite state transducers – it doesn't provide an object oriented interface to the data. So if a developer wishes to remember some result, he/she has to remember the query sent to the

server and the offset of the interesting result. This significantly impairs subsequent data retrieval performance, especially when the query returned many results. The other problem is that the data is transferred in the form of a semi-structured text so the result of the query has to be parsed, which further impairs its performance. As a result the system has to be augmented with another storage engine to remember the issued queries or the processed results, which makes it impractical as a standalone data storage solution.

The last interesting related work in the field of NLP is the access layer build on the top of PolNet – one of the two WordNets build for Polish. [3] describes the architecture of the POLINT-112-SMS system and the reasons for building a custom query language on the top of XML-based data store used to store the WordNet, which the POLINT system interacts with. The author argues that a direct integration of the WordNet with the system implementation language (Prolog is given as an example) would introduce high coupling between the NLP system and the storage system. But the author also indicates that the adoption of a generic solution such as SQL database, XML store or RDF store with SPARQL interface would yield a system which is less suited for NLP tasks, such as navigation over the WordNet structure or reasoning over the data – the queries would be much more verbose and less meaningful for the developers. So it would be harder to maintain the interoperability between the systems.

It is true that the general purpose data manipulation languages like SQL or SPARQL are more verbose than the language provided by the access layer. It is also true, that direct integration of the data with the system would introduce high coupling. Still it is not obvious that if an object oriented interface was provided, the queries expressed in the same language as the client system implementation language would not be concise enough. Most of the examples provided by the author are easily expressible in modern programming languages like Ruby or Python, so there is no need to create such a domain specific language. This solution is more powerful since it is much easier to write code in such general purpose languages, than extend syntax and implement semantics of a domain specific language.

Concerning Ruby and the solutions that bring the benefits of object-orientedness into the data storage world, there are many of them. However we compare only the most known and most used library, that is ActiveRecord [7], since it is the most popular object-relational mapper for Ruby. The important omission is Neo4j [9] – a graph database which seems to be most similar to ROD. The reason is that this database is available only for JRuby, that is a Ruby implementation for the Java Virtual Machine, while ROD is targeted at MRI – the primary Ruby implementation in C.

4. Database Design

4.1. Overview of implementation

The database features described in the "Motivation" section impose two primary constraints on its design – raw read performance and easiness of use provided by the Ruby interface. This language does not seem to be the best choice for implementing the fastest data storage engine, that's why the core of the database is implemented in C. Still its interface is pure Ruby, which mimics the well know object-relational mapper for Ruby – ActiveRecord used as the default ORM in Ruby on Rails framework. To bridge the gap between the fully object-oriented and dynamically typed Ruby and the procedural and statically typed C a RubyInline[3] library is used. It was created to allow developers replace slow, but critical Ruby code with a C implementation. Its name comes from the fact that the C code is written directly in a Ruby class so there is no need to maintain separate C files nor manually compile them. The C code is generated and compiled when the Ruby code containing it is run for the first time. As a result a shared library is created and linked in the run-time while the dynamic nature of Ruby allows for extending the already defined classes with new methods. The RubyInline library maintains the changes in the Ruby/C code, so the C code is re-complied only if the developer provides new methods or changes the C implementation.

This library allows ROD to generate C code providing access to database for each class that is intended to be stored in the database. As a result this ensures the highest performance provided by statically typed C while maintaining flexible interface of dynamically typed Ruby. It is not a surprise that this design introduces some restrictions concerning the types of Ruby values that might be stored in the database. But these restrictions are soften by the mechanisms provided by the database itself – the fact that is very easy to create a new type storable in the database as wall as a serialization mechanism, that allows for storing immediate Ruby values (if referential integrity does not have to be maintained).

4.1.1. Access method

The access method of the database is provided by the `mmap` system call available on most of the modern operating systems, Linux in particular. This call allows for mapping a file stored on a disk to a physical memory region. The pages containing the data are loaded only if the particular memory address is accessed. This greatly simplifies the implementation of the data access routines and it transfers the responsibility of memory management to the operating system. The only operations that have to be maintained by the database are growing the data file and its re-mapping. The OS is responsible for the rest – reading of the data, mainte-

[3] http://www.zenspider.com/ZSS/Products/RubyInline/

nance of the buffered pages and writing the data back to disk. So the primary memory manager of the database is just the memory manager of the OS. Such a solution should suite needs of most of the developers. If it is not the case, it is usually possible to select a different memory manager of the whole OS.

4.2. Data model

The primary storage unit of ROD is a Ruby object, but the core implementation language is C, so the data model of the database is mapped directly to the C data types. This means that any Ruby object apt for being persisted in ROD has to be represented as a C struct. Thanks to the RubyInline library this struct is defined at run-time by calling Ruby methods and it is used to represent the attributes and associations of the persisted object. There are several base rules of mapping between Ruby objects and C structs. The first is that the name of the attribute or association in the class is used as the name or prefix of the names of corresponding fields in the C struct. The second is that the name of the struct is a transformation of the fully qualified name of the Ruby class, with colons replaced by underscores. The last is that the C structs of a single class constitute a continuous array in the memory and are uniquely identified by their index (that is the offset in the corresponding memory-mapped file). This index is the database identifier of the object, it is named `rod_id` and its smallest value is 1.

4.2.1. Attributes

The database defines several types of attributes: atomic, fixed length attributes; atomic, variable length attributes and complex attributes. Atomic, fixed length attributes are these attributes that might be directly mapped to atomic C data types – such as Ruby `Fixnum`, `Integer` and `Float`. In fact at present only the following C types are used: `int`, `unsigned long` and `double`. The first type is used to map values of Ruby `Fixnum` type, the second – positive values of Ruby `Integer` type that are smaller or equal to the maximum value of unsigned long and the third – values of Ruby `Float` type, with the restrictions imposed by the `double` type. If there is such an attribute in the persisted class, a corresponding field is created in the C struct and the value is stored using Ruby built-in Ruby-to-C mapping macros and read using built-in C-to-Ruby macros.

Ruby strings are represented by atomic, variable length attributes, which are stored in a flat file. Since Ruby strings may contain non-string terminating zeros, they are identified by their offset in the file and their length. These two values are represented in the C struct as `_offset` and `_length` fields respectively, prefixed with the name of the attribute.

Complex attributes have to be further divided into two groups: immediate values, that do not have to preserve referential integrity, such as Ruby arrays of integers or hashes of strings

and complex Ruby objects that have to preserve referential integrity. The values of the first type are marshaled using Ruby built-in marshal function or encoded using a JSON[4] format (the method is left as a choice for the user). They are stored in the same way as strings, with the exception that strings are always UTF-8 encoded before being stored. This also applies to values of atomic Ruby types besides `Integer`, `Fixnum` and `Float` such as `Symbol`.

If the complex values are supposed to preserve referential integrity, they have to be defined by classes that use ROD storage mechanism and such attributes have to be transformed into singular associations. In most circumstances this should not be a problem, since Ruby incorporates the uniform accessor pattern, so from the point of view of object's interface attributes are indistinguishable from singular associations.

4.2.2. Singular Associations

Singular associations that is 1-to-1 and n-to-1 associations from the point of view of the class on the left are treated as follows: the C struct defines an unsigned long field with the name of the association as prefix and `_id` suffix, that stores the `rod_id` of the referenced object. If the association is empty, the value of the field is 0 (since the smallest valid `rod_id` is 1). In the default scenario the type of the object is guessed form the name of the association, that is if there is a *user* associated with one *address* and the association's name is *address* then the guessed Ruby class is `Address`. This might be changed with an option `class_name`, which directly indicates the name of the association's class. But this might be further changed with a `polymorphic` option. Polymorphic association indicates that the values of the association don't have one type. In such a case, the struct has an additional field of `unsigned long` type with `__class` suffix, which holds most significant bits of the SHA1 checksum of the name of the class of the associated object (the `class_id` of the class). The pair `(rod_id,class_id)` uniquely identifies the object in question.

4.2.3. Plural Associations

Plural associations that is 1-to-n and n-to-m associations from the point of view of the class on the left are treated similar to the variable length atomic attributes. The C struct stores their offset and count as `_offset` and `_count` fields, prefixed with the name of the association. The offset of the association indicates its index in an auxiliary memory-mapped file used to store all plural associations of the database. This file contains C structs that have a `rod_id` field capturing the `rod_id` of a single associated object. A continuous array of such structs is used to represent one plural association. As a result navigation via these association

[4] http://www.json.org/

is vary fast, since in most circumstances the structs referring to the objects in one association are loaded during one disk look-up, which is unlike to happen in relational databases.

The types of the objects in plural association are also guessed from its name, but the same as in ActiveRecord the name of the association is in plural, while the guessed class is in singular, e.g. if there is a *user* that has many *items*, the guessed Ruby class is `Item`. The same as in singular association the class may be given explicitly or the association may be polymorphic. In the second case the C struct besides the `rod_id` of the object stores the `class_id` of its class, which is computed the same way as in the case of singular associations.

4.2.4. Indexing

Attributes and singular as well as plural associations might be indexed. This seems to be strange in the case of associations, since in most circumstances it is easier to create a bidirectional association instead of indexing it on one side (which from the point of view of the data is equivalent). However, since the database is not normalized, an index on the association allows for maintaining a consistent bidirectional association without any extra effort from the user of the database. This feature has another, maybe even more important application – it allows for maintaining bidirectional associations across separate databases. While it is quite easy to create associations from one database to another and use them simultaneously, it is usually not feasible (or convenient) to define them as bidirectional (the databases have to form a partially ordered set). There are some use cases in NLP, which use such a configuration and while there is no enough space for explaining such a scenario in details, it turns out that indexed associations are very useful in such a case.

There are three options for indexing the objects: the first (named `flat`) that uses Ruby hash table to map the attribute/association values to the corresponding object, the second (named `segmented`) that uses the same method but splits the hash keys into buckets and the third one (named `hash`) which uses Berkeley DB [4] Hash access method. In the first two cases the resulting hash/hashes are kept in memory when the database is open and are marshaled and stored on the disk when the database is closed. The only difference is that the second method have better observed start-up time, since the whole index do not have to be read at-once – only the part of the hash that contains the keys of a particular bucket. The third method is without comparison in performance both in the case of loading and writing the index into disk, but imposes an additional dependency on the library, that is the Oracle Berkeley DB. Since the Berkeley DB is distributed under GPL and commercial licenses, in some circumstances it might be better to use pure ROD which is distributed under much more permissive MIT/X11 license.

4.2.5. Metadata

Besides the files that store the structures corresponding to the Ruby objects, variable length attributes and associations, the database has an additional file in YAML[5] format, that contains its metadata. The metadata cover the version of the ROD library, the database creation and last modification time, the number of objects stored for each class and the structure of each of the persisted classes. The last information allows for detecting if the run-time data model is the same as the data model in the database. It also allows for generating the run-time model (that is the Ruby classes) as well as migrating from one data model to another.

5. Database Interface

As described in the "Motivation" section the database was designed for the Ruby programming language. This means that both the data definition language and the data access language are just subsets of that language. Although this language is not as popular as Java or even Python, its syntax is so simple, that even for an unacquainted person it should look almost like a pseudo-code. It is hoped that most of the code will be intelligible with the minimal comments provided.

5.1. Data Definition Language

In order to be persisted in ROD, a class have to inherit from the `Rod::Model` class (the `Rod::` in the class name is its namespace). It also have to indicate the database class that inherits from the `Rod::Database` class, which represents the ROD database. This might be omitted if the database was indicated in its parent class. Thus a minimal code for a class is as follows:

```
require 'rod'

class Database < Rod::Database
end

class User < Rod::Model
  database_class Database
end
```

The first line loads the ROD library. The first class definition, which starts with the `class` and ends with the `end` key-words, defines the `Database` class, while the second defines the `User` class that uses the `Database` class. The < sign indicates the inheritance relation. The

[5] http://www.yaml.org/

`database_class` call on the `User` class is used to indicate the database class associated with the class.

Attributes are defined with the field method, singular associations with `has_one` and plural associations with `has_many` method. All of them treat the first argument as the name of the attribute/association. The attribute also have to define its type as a Ruby symbol (which starts with a colon) and might be one of: `:string` (for strings), `:integer` (for integers), `:float` (for floating point numbers), `:ulong` (for unsigned long integers), `:object` (for marshaled values) and `:json` (for values stored in JSON format). The options available for the attributes and associations are provided as key – value pairs (where the key is separated from the value by =>). For instance if one needs to define a flat index for one of the attributes, one has to provide `:index => :flat` option for the `field` method call.

A more sophisticated example looks as follows:

```
require 'rod'

class Database < Rod::Database
end

class Model < Rod::Model
  database_class Database
end


class User < Model
  field :name, :string
  field :surname, :string, :index => :flat
  has_one :account,
    :class_name => 'DatabaseAccount'
  has_many :addresses
end
```

```
class DatabaseAccount < Model
  field :login, :string,
    :index => :flat
  field :password, :string
  has_one :user
end


class Address < Model
  field :street, :string
  field :number, :integer
  field :city, :string
  has_one :user
end
```

The `Model` class is defined only to prevent all the other classes to repetitively define their databases. This code defines the `User` class that has `name` and `surname` attributes of the `String` type, where the second attribute is indexed with `flat` index as well as singular association `account` pointing to an object of the `DatabaseAccount` class and plural association `addresses` pointing to the objects of the `Address` class; the `DatabaseAccount` class that has `login` and `password` attributes, both of the `String` type and singular association `user` pointing to the object of the `User` class; the `Address` class, that has `street`, `number` and `city` attributes of the `String`, `Integer` and `String` types respectively and singular association `user` pointing to the object of the `User` class.

It should be restated that the above code examples are all valid Ruby programs – the data definition language is just a subset of Ruby, not its super-set. The calls to `field`, `has_one` and `has_many` methods are just ordinary method calls whose callee are the respective classes.

5.2. Data Manipulation Language

When the class defines the respective attributes and/or associations, the following data manipulation methods become available. The class constructor (`new`) may be called without parameters, which creates a new, empty object; it may be called with a hash of key-value pairs that initialize the respective attributes/associations with the values given and it may be also called with the `rod_id` of an object that has been stored in the database. In the last case the first access to the attributes or associations causes the object to be loaded from the database. Furthermore the values of the respective attributes and singular associations might be read and modified via getters and setters whose names are the same as the name of the attribute or association. In the case of plural association there are also a getter and a setter which allow for reading and writing a whole collection of objects. This collection might be read and modified like an array of objects.

The newly created object as well as an object retrieved from the database is always in a detached state, that is any changes made to it are only propagated to the database when a `store` method is called on it. If the object is fresh, the call to this method creates new entry in the database and sets the `rod_id` of the object. If the object was retrieved from the database, the call propagates to the database only the values that have changed (dirty tracking).

There are finders created for the attributes and associations that are indexed for a given class. The finders start with `find_by_` and `find_all_by_` prefix and are suffixed with the name of the attribute. The call of the first type returns at most one object or `nil` (which is also an object in Ruby by the way) in case it was not found, while the second returns a collection of objects, which is empty in case no such objects were found. Both these finders are class methods, so the query looks as follows: `User.find_by_surname('Smith')`. So far the library does not provide specific mechanism to query by attributes and associations that are not indexed. However the `Rod::Model` class includes the `Enumerable` mix-in which offers a general purpose query mechanism.

The following code shows how the data manipulation language works (assuming we have access to the data model defined in the previous listing) – the explanation is given in comments which start with the # sign:

```
# creation of an object with the initialized attributes
user = User.new(:name => 'Fred', :surname => 'Smith')
# storing the object in the database
user.store
# creation of a fresh object
user = User.new
# setting the values of the attributes via setters
user.name = 'Fred'
user.surname = 'Smith'
# storing the object in the database
user.store
# initializing the user with rod_id = 10
```

```
user = User.new(10)
# reading the values of the attributes via getters
user.name
user.surname
# a singular association example
account = Account.new(:login => 'fred', :password => 'querty')
user.account = account
user.store
account.user = user
account.store
# plural association example
address1 = Address.new(:street => 'Straight', :number => 10,
   :city => 'Warsaw')
address1.user = user
address2 = Address.new(:street => 'Short', :number => 20, :city => 'Krakow')
address2.user = user
user.addresses << address1
user.addresses << address2
user.store
address1.store
address2.store
# searching an address with 'Short' street for a given user using general
# query mechanism
user.addresses.find{|address| address.street == 'Short'}
# searching a user with 'Smith' surname using the built-in query mechanism
User.find_by_surname('Smith')
# searching for all users with 'Smith' surname
User.find_all_by_surname('Smith')
```

Two peculiarities should be noted regarding the data manipulation. The first concerns the bidirectional associations – unlike in object-relational mappers, the objects on both sides have to be set independently. If not, only one direction is persisted. The second peculiarity concerns associations between fresh objects. The association is persisted only if both of the objects received the `store` method call. However it does not matter which of the objects was stored first, as long as both of them were stored in the same open-close database cycle.

6. Results

The primary goal of the database is to be used as a storage engine for NLP tasks. As such it is used in three libraries: `rlp-grammar`[6] providing access to an inflectional dictionary of Polish, `rlp-semantics`[7] providing access to a semantic dictionary of Polish and `rlp-corpus`[8] providing access to a corpus with Polish texts. All these libraries are building blocks of a framework for information extraction from Polish texts. Although there are many missing features in ROD, it proved to be quite stable and offering a much better performance than a relational database and providing more uniform interface than the other solutions available for Polish.

[6] https://github.com/apohllo/rlp-grammar
[7] https://github.com/apohllo/rlp-semantics
[8] https://github.com/apohllo/rlp-corpus

ROD was compared with SQLite [5] working with ActiveRecord ORM. The test used to compare the libraries works as follows – a text in polish is scanned and each text segment is looked up in an inflectional dictionary. If the segment belongs to the dictionary, it is checked if the word form is unambiguous, that is it belongs to only one flexeme. If this is true it is checked that the flexeme has more word forms and that it is a noun. In such a case the occurrence of the word form is counted (to make sure that the implementations work the same). This test case represents a quite simple text processing task – in real applications much more data about given word form and its flexemes usually have to be fetched from the database. In Ruby the kernel of the test looks as follows (this is exactly the same code for ROD and ActiveRecord – the only difference is the WordForm class on which the first call is made):

```
form = WordForm.find_by_value(word)
if form && form.flexemes.count == 1 &&
      form.flexemes.first.word_forms.count > 1 &&
  form.flexemes.first.taggings.any?{|t| t.tags.map(&:value).include?("subst")}
  count += 1
end
```

The tables 1 and 2 show the performance comparison between ROD and SQLite used with ActiveRecord. Full code of the benchmark is available at `http://github.com/-apohllo/rod-benchmark`.

The results should be interpreted as follows: two texts file were used as input for the benchmark: `text_1.txt` and `text_2.txt`. For each text 10 iterations of the benchmark were run (the table shows only the 3 first result, the whole data is available under the address of rod-benchmark). The time measured is the *real* value returned by the `time` command under Linux. All the benchmark runs were performed within the scope of one process. As a result ROD performance seems to be quite varying, but in fact the first large number (11 seconds for the 0^{th} iteration for the `text_1.txt` file) is due to the fact, that the relevant dictionary file pages are load into memory. Then cache of objects is used, so the test runs quite fast (around 1 second). When a new text is processed (which is larger than the first one) some new objects are created in the first run, but in general the test runs much faster than the first one (around 3 seconds). In all cases this gives significant performance improvements compared to ActiveRecord.

Table 1

SQLite 2.8.16 performance with ActiveRecord 3.1.1

File	Iteration	Time [s]	Difference [s]	Speed up [times]
text_1.txt	0	33.65	-	-
text_1.txt	1	33.66	-	-
text_1.txt	2	33.54	-	-
text_2.txt	0	42.48	-	-
text_2.txt	1	41.87	-	-
text_2.txt	2	42.47	-	-

Table 2

ROD 0.7.1 performance compared with SQLite

File	Iteration	Time [s]	Difference [s]	Speed up [times]
text_1.txt	0	11.32	-22.33	x 2.97
text_1.txt	1	0.99	-32.67	x 34.00
text_1.txt	2	1.08	-32.46	x 31.05
text_2.txt	0	2.79	-36.69	x 15.22
text_2.txt	1	1.27	-40.60	x 32.97
text_2.txt	2	1.27	-41.20	x 33.44

However, the results presented in tables 1 and 2 should be treated only as illustrative – they are not meant as a full performance comparison of ActiveRecord and ROD libraries. This only shows that ROD might have much better performance in scenarios covering Natural Language Processing. It should be noted that no fine tuning (besides proper indexing of fields that are involved in the presented algorithm) of ActiveRecord or SQLite was done in order to improve their performance in this task. There is also no comparison in the size of memory used by the libraries, which is an important factor when comparing such libraries. But on the other hand the reader has full access to the benchmark code as well as the library and is advised to perform his/her own tests.

7. Conclusions and Perspectives

Although the performance gains provided by Ruby Object Database are important in the case of NLP tasks, there are many drawbacks of using ROD: lack of normalization, no transactions, data modification anomalies, no support for removal of objects, data portability problems, tight coupling between the database and the Ruby language, batch updating of the data, missing support for Windows platform and more. As such the database is not mature yet. Some of the deficiencies will be addressed in the next version of the library, which will based on Oracle Berkeley DB. Among the other benefites this will provide full support for ACID transactions, data portability and this will lose the coupling between the database and Ruby opening possibility for implementations in other programming languages.

BIBLIOGRAPHY

1. Daciuk J.: Incremental Construction of Finite-State Automata and Transducers, and their Use in the Natural Language Processing. 1998.

2. Fellbaum C.: WordNet. Theory and Applications of Ontology. Computer Applications, 2010, p. 231÷243.
3. Kubis M.: An Access Layer to PolNet-Polish WordNet. Human Language Technology. Challenges for Computer Science and Linguistics, 2011, p. 444÷455.
4. Olson M., Bostic K., Seltzer M.: Berkeley DB. Proceedings of the FREENIX Track: 1999 USENIX Annual Technical Conference, 1999, p. 183÷192.
5. Owens M.: The denitive guide to SQLite. Apress, 2006.
6. Przepiórkowski A.: Korpus IPI PAN. Wersja wstępna. Instytut Podstaw Informatyki PAN, 2004.
7. Tate B. A., Hibbs C.: Ruby on Rails: Up and Running. O'Railly Media, 2006.
8. Thomas D., Fowler C., Hunt A.: Programming Ruby. Pragmatic Bookshelf, 2004.
9. Vicknair C., Macias M., Zhao Z., Nan X., Chen Y., Wilkins D.: A comparison of a graph database and a relational database: a data provenance perspective. Proceedings of the 48th Annual Southeast Regional Conference ACM, 2010, p. 42.
10. Woliński M.: Morfeusz-a practical tool for the morphological analysis of Polish. Intelligent information processing and web mining, 2006, p. 511÷520.
11. Zawodny J.: Redis: Lightweight key/value Store That Goes the Extra Mile. Linux Magazine, http://www.linux-mag.com/id/7496/, 2009.

Wpłynęło do Redakcji 16 stycznia 2011 r.

Omówienie

W artykule przedstawiono ROD – obiektową bazę danych dla języka Ruby do zastosowania w zagadnieniach przetwarzania języka naturalnego. Baza ta została zaprojektowana z uwzględnieniem specyficznych wymagań stawianych elektronicznym słownikom oraz korpusom tekstów stosowanych w tej dziedzinie wiedzy. W artykule przedstawiono specyficzne wymagania stawiane zasobom tego rodzaju oraz przedstawiono kilka alternatywnych implementacji stosowanych w szczególności w kontekście języka polskiego.

Istotna część artykułu poświęcona jest sposobowi implementacji przedstawianej bazy danych. W szczególności omówiono mechanizm persystencji danych oraz mechanizm odwzorowania obiektów języka Ruby na struktury bazy danych, ze szczególnym uwzględnieniem atrybutów, związków jeden-do-wiele i wiele-do-wiele oraz mechanizmu indeksowania. W dalszej części artykułu przedstawiono język definicji danych oraz język manipulacji danymi.

W tabelach 1 i 2 pokazano porównanie wydajności opisywanej bazy danych z relacyjną bazą danych SQLite, współpracującą z biblioteką ActiveRecord. Jakkolwiek dane te mają charakter wyłącznie ilustratywny, pokazują, że zastosowanie odmiennego sposobu przechowywania danych niż baza relacyjna pozwala uzyskać znaczące przyspieszenie w zagadnieniach przetwarzania języka naturalnego. Z tego względu prace nad bazą danych będą kontynuowane w celu zwiększenia zakresu jej możliwych zastosowań.

Address

Aleksander POHL: Jagiellonian University, Departament of Computational Linguistics, Łojasiewicza 4 st., 30-348 Krakow, Poland, aleksander.pohl@uj.edu.pl.