STORE: EMBEDDED PERSISTENT STORAGE FOR CLOJURE PROGRAMMING LANGUAGE

Konrad Grzanek[1]

[1]IT Institute, Academy of Management, Lodz, Poland
kgrzanek@swspiz.pl

Abstract

Functional programming is the most popular declarative style of programming. Its lack of state leads to an increase of programmers' productivity and software robustness. Clojure is a very effective Lisp dialect, but it misses a solid embedded database implementation. A store is a proposed embedded database engine for Clojure that helps to deal with the problem of the inevitable state by mostly functional, minimalistic interface, abandoning SQL and tight integration with Clojure as a sole query and data-processing language.

Key words: Functional programming, Lisp, Clojure, embedded database

1 Introduction

Functional programming languages and functional programming style in general have been gaining a growing attention in the recent years. Lisp created by John McCarthy and specified in [8] is the oldest functional programming language. Some of its flavors (dialects, as some say [9]) are still in use today. Common Lisp was the first ANSI standardized Lisp dialect [13] and Common Lisp Object System (*CLOS*) was probably the first ANSI standardized object oriented programming language [14]. Apart from its outstanding features as a Common Lisp subset.

Various Lisps were used in *artificial intelligence* [11] and to some extent the language comes from AI labs and its ecosystem. Common Lisp was used as the language of choice by some AI tutors, like Peter Norvig (in [10]). But the whole family of languages address general problems in computer science, not only these in AI.

John Backus argues [3] that the functional style is a real liberation from the traditional imperative languages and their problems. Many other scientists and famous programmers confirm the fact of really hard to solve issues in programming related to C language features and it's derivatives. *Coders at Work* [12] by Peter Seibel are one of the sources of vital examples. Abelson and

Sussman in *Structure and Interpretation of Computer Programs* [9] describe in details two formal models behind computational processes, namely the *functional model* based on the λ-calculus by Alonzo Church [2] and the *environmental model* in which the notion of state, variable and environmental bindings are being introduced. The environmental model corresponds to the original idea of universal computational machines made by A.M. Turing [1]. The functional model is founded on the notion of pure *first-class* functions.

The notion of state as stated by Backus in [3] has been considered after him by many as the major cause of hard to solve problems appearing in the concurrent programming and in critical robust systems programming in general. This was and still is one of the reasons why functional languages like ML, Haskell [5] and Lisp, of course, are so adequate in some kinds of applications [4].

Development of purely functional programming style forced the development of effective *persistent data structures*. Works by Okasaki [6] and Bagwell [7] opened the way for languages like *Clojure* created by Rich Hickey [16, 15] to introduce *software transactional memory* as the most robust known way to eliminate concurrency problems emerging from using the state and variables in places where the ideal stateless functional style must be omitted.

Apart from that Clojure is a JVM (*Java Virtual Machine*) language. It offers full interoperability with Java and its libraries. That makes it a very interesting choice for people who want to write the mainstream software in a unique, robust way. Applying this language to solve common day-to-day programming problems as well as scientific ones requires having a solid database solution, including an *embedded database*.

2 Forces, environment and requirements

The idea to create an embedded database storage engine emerged from the author's previous works on complex software systems like the design patterns instances recognition tool described in PhD thesis [19]. There was also an embedded db – *the repository* – capable of storing database items [18]. Alas this successful solution has some severe limitations. The most important one is the static nature of Java (in terms of the system type) for which the *repository* was originally created. It practically closes any possible discussion on whether to use it or not in Clojure, as any Lisp, the classic dynamically typed system.

2.1 Embedded database

As stated in [17], for many research and production applications, an ideal database would have at least these characteristics:
- ability to manage thousands millions of objects, including complex ones,
- robustness
- resistance for heavy loads
- scalability due to large datasets
- tight integration with a high level programming language
- simple and expressive API

The integration with a programming language is of a special importance because of two reasons. First of all, it allows the programmer to omit using SQL and to go directly into coding the application logic and database queries using only one, native language. We avoid unnecessary programming languages diversity here. The other reason is the performance issue; separating application logic and database operations logic leads to a natural loss of performance. Database and application communication channel is the bottleneck here.

These requirements and considerations caused an immediate observation that choosing an embedded solution would be the most reasonable thing to do, if one thinks about creating a robust and highly scalable storage engine. We also rely on our previous experiences and research results with the *repository* storage engine described above.

2.2 Lack of state and persistence

Using pure procedures and assuming the lack of state as well as the timelessness of the computational model requires solid collections implementation every collection is immutable. In this model, the procedures that model functions in mathematics are side-effect free. This means that any collection passed to a procedure will not be modified. If the procedure wants to apply any modification to the original collection operand, it has to make a defensive copy of the collection and return this newly created object with all the modifications applied. In the traditional programming languages like Java, C++, Ada making a memory copy of the original is the only way to go. Unfortunately, the cost of such an operation is so high that in practice this technique cannot be applied as a general way of introducing immutability of collections. This is the major reason for the apparent absence of the idea of immutable collections in the mentioned languages and also in all languages considered mainstream programmers' tools.

Clojure takes another approach using so called *persistent collections*. The persistent collections are immutable, but they address the problem of pessimistic time and memory behavior by exploiting interesting algorithmic tech-

niques that are in general based on re-usability and sharing. These were presented in detail in [7] and earlier in [6].

Apart from the idea of persistent collections, the database problems exist. From now on one should assume that the word *persistent* holds its original meaning that refers to storing information in the persistent storages. This is what this paper really discusses. Unfortunately, in the case of the persistent storage, no good solutions exist that really could save the programmers from the notion of state, time and variables. This constatation comes from the characteristics of Turing machine's model underlying every computer; data being stored on disk to be re-read in the future simply must be written on the "tape". And further, writing on the tape of UTM is equal to using a variable and the assignment operation. It introduces inevitably the notion of state and time.

So there are no effective ways to omit falling into the statefulness when the databases are being considered. The only way to deal with this annoying problem is trying to remove some inherent pain by solid implementation and tight and elegant integration with the functional programming languages. Some approaches like monads [20], have a really great mathematical background but the way they hide the statefulness may be misleading to some programmers. Our conception assumes using a more traditional, yet powerful and readable semantic solution.

2.3 Existing solutions

There are several Lisp database solutions. For Clojure there is a contribution library packaged as *clojure.contrib.sql*. The library offers a very high level of abstraction different to the traditional relational database interface – JDBC (*Java Database Connectivity*). But we seek for an embedded database solution.

Common Lisp is the most mature technology in this respect. There is a mature and comprehensive module for Allegro Common Lisp (by Franz Inc.) called Allegro Graph [21]. This database offers a great scalability and tight language integration and is a base for some established Web 2.0 solutions provided by the company. Its major deficiency from the research and academic point of view is its commercial character.

When we talk about non-commercial ones, we find Elephant: A Persitent Object Database for Common Lisp [22]. This is a classic embedded non-relational database having a solid CL interface built around Common Lisp Object System. There are two problems in general related to using this database. First of all it is a purely Common Lisp solution, and the second one – it is CLOS dependent. We search for a slightly more lightweight engine and Clojure – oriented.

There are also two other ways to deal with the problem in Clojure. The Apache Derby [24] database may be embedded in the language using seam-

less Clojure - Java interoperability and the fact that Derby is a Java database. But Derby is still a traditional relational database with an SQL engine. This in turn causes the tight db-language integration suffering.

Another, and probably the most interesting alternative is Fleet DB Clojure binding [23]. This is a very promising attempt to build a very dynamic database framework. It suffers a little bit from the already defined query language (as stated before we want a "query-language IS the implementation language" solution) and its major problem is the fact that it's a RAM database.

Finally, we got to the point of specifying the set of the additional design and implementation requirements:

- The database should not be a relational one.
- Query language is the application logic language.
- Data should be stored on disk, not in the RAM memory.
- Engine should be suitable to store Clojure objects of any kind, except for lazy collections, which should be converted to non-lazy.
- Finally, some previous experiences with a very effective repository model and implementation should be used.

3 Design and API

The presented embedded database called *Store* is built with Berkeley DB Java Edition [25] as a low-level storage engine. This is a reminiscence of the previous author's works on the mentioned repository as described in [17], [18] and [19] . But Store's features relate to the low-level persistence layer only remotely. The source of its flexibility comes from its architectural design corresponding to the proposed layers of the abstraction described below.

3.1 Layers of abstraction

The following Fig. 1. presents abstraction layers of Store engine together with the whole Clojure/JVM environment. Custom implemented elements are marked bold. These layers will be described separately in the following subsections.

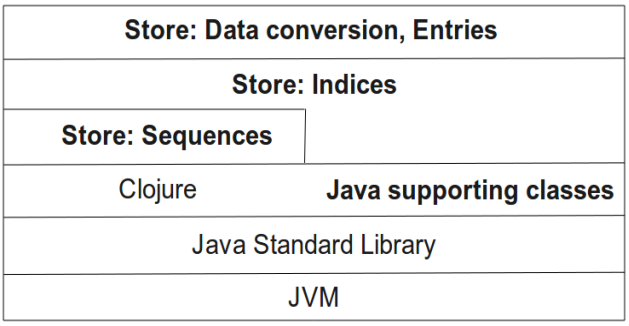

Figure 1. Store engine abstraction layers

3.2 Sequences

They represent named persistent streams of natural numbers. Their role is to provide unique auto-enumerated key values for stored objects. The basic construct for referring to a sequence is:

```
(stored-sequence name start step)
```

where start is 0 by default and step is 1. If a sequence of the given name does not exist, it will be created and registered in the engine's internals. An example use is like:

```
(gen (stored-sequence :test)) ==> 0
(gen (stored-sequence :test)) ==> 1
(gen (stored-sequence :test)) ==> 2
```

resulting in 0, 1, 2, 3, … in subsequent calls. Procedure ***gen*** is one of a few side-effect procedures in the whole framework. It returns the next number in the sequence and "moves" the sequence onto the next value. ***:test*** is Clojure *keyword* playing a role of the sequence's name.

Another procedure called recent returns the last generated number, but it is side-effect free; it does not change the sequence's state:

```
(recent (stored-sequence :test)) ==> 2
```

In the end, two procedures may be used to *delete a sequence* or *all sequences* respectively:

```
(drop-sequence (stored-sequence :test))
(drop-all-sequences)
```

3.3 Indexes

Indexes are the key storage mechanism and the key abstraction. We assume a very simple view on data stored in our databases at this level. Every index contains pairs of keys and values. There are two types of indexes:

1. Standard indexes. Keys and values are expected to be arrays of bytes and must be specified when performing data manipulation.
2. Auto-enumerated indexes. Keys are expected to be of type Long (long integer – *java.lang.Long*) and values are expected to be arrays of bytes. When inserting a new value entry, the key is never specified, but the index itself uses an internally assigned sequence to generate a new key.

Using the mechanism starts with creating a reference to an existing or a created on demand index. Just like in the case of sequences, the name is the encouraged way to make a reference.

```
(index :test)
```

By default the index is auto-enumerated. It may be specified explicitly by passing an additional parameter like below:

```
(index :test :auto)
(index :test true)
```

On the other hand, it is possible to ensure creating a standard index:

```
(index :test :non-auto)
(index :test false)
```

It is worth noting that choosing a selected option (like above) is possible only when creating a new index, that is requesting for a reference to an index of a previously not-specified name. When requesting for an existing index, the additional parameter value has to be omitted or correspond to the original settings of the selected index. Violating the original settings will cause an exception.

Indices, in opposition to the sequences, may (but do not have to) be explicitly closed after the use. The close operation is:

```
(close (index :test))
```

Another option for the programmers who want to use indexes within a more idiomatic Clojure use-and-release pattern is using a canonical ***with-open*** macro:

```
(with-open [i (index :test)]
  ...)
```

This is possible because (**index** …) is in fact a Java object of a type implementing a close() method.

All indices opened during the application run-time and not closed will be closed at the system shutdown.

Retrieving value for a given key is possible by calling a method get on the index reference:

```
(.get (index :test) <key>)

or in a more Java-ish way:

(.. (index :test) (get <key>))
```

A similar pattern exists for deleting key-value entry

```
(.delete (index :test) <key>)
(.. (index :test) (delete <key>))
```

and for storing (insertion or updates):

```
(.put (index :test) <key> <value>)
(.. (index :test) (put <key> <value>))
```

In the case of inserting into an auto-enumerated index, the key must not be provided, so that the above call reduces to:

```
(.put (index :test) <value>)
(.. (index :test) (put <value>))
```

An index may be deleted with all its data very similarly to the described way of deleting sequences:

```
(drop-index (index :test))
(drop-all-indices)
```

3.4 Store: entries and data conversion

Entries and store mechanism are the key way to refer to the data stored in an index. First of all, there is a Clojure *multi-method* [16] called ***entries.*** The method always returns a sequence of pairs <*key*, *value*> representing index content:

```
(entries (index :test))
```

When called with and index as an argument, the procedure returns a sequence of pairs each containing the key and value in the form of byte arrays. It is a very raw, low-level way to the access data.

To make the data to be more programmer-friendly, there exists an abstract notion of the highest abstraction level, namely ***store***. The store is the most effective and preferred way to work with the whole database engine. Again, like in the case of indices, one may refer to a named store giving its name. It must be mentioned that a store is only a very wrapper around an index of the same name as the store. The store references may be created in multiple ways:

```
(store (index :test))  # Refereing to an index
(store :test)          # Refereing to an index by name
```

When making a reference like above, the resulting store assumes the values in the index may be of any type serializable to Java *string* and capable of being ***read-from-string*** by the typical Clojure evaluation mechanism. Additionally when passing a name like :test to make a store reference means referring to an auto-enumerated index.

Using conversion to Java *string* and reading into Clojure object as the default serialization mechanism is enough in most cases, especially when storing composite Clojure objects: maps, vectors and sets. Nevertheless, sometimes it is better to specify the type of values or keys to achieve more fine-grained serialization:

```
(store <index-or-name> <value-type>) # auto-enumerated
(store <index-or-name> <key-type> <value-type>)
```

where *key-type* and *value-type* belong to one of **:str :boolean :byte :short :char :int :long :float :double :bigint :bitset**.

For example

```
(store (index :test :non-auto) :str :bigint)
```

creates a store around a standard non-auto-enumerated index to map strings onto java.lang.BigInteger objects. The same effect may be achieved by

```
(store :test :str :bigint)
```

Finally

```
(entries (store :test :str :bigint))
```

now makes a sequence of pairs <*String, BigInteger*>. There are also two convenience procedures to convert pairs into the more object-oriented form.

```
(itemize (entries (store ...)))

Returns a sequence of items {:id key, :data value}.
And further

(vectorize (entries (store ...)))
```

produces a sequence of vectors [*key value*].

3.5 Resource management

It is one of the key issues when creating a robust database API to provide a clean and easy to use resource management interface. In Store there is a simple ***doclean*** macro. Calling ***entries*** results in opening a database cursor in the implementation layer. And this call must be done in a body of (doclean …) form:

```
(doclean (entries (store ...))
(doclean (entries (index ...))
```

Omitting doclean causes an immediate signal:

```
Entries may be accessed only in a clean-up context.
  [Thrown class java.lang.IllegalStateException]
```

So there is no place for the programmer to introduce a resource leak. The macro evaluates its body and returns a result – the value of its last expression. This is a classic Lisp behavior. Moreover, it releases all the database cursors opened inside the body immediately after returning the result. ***doclean*** is exception-safe.

In the end, it should be noted that the pattern is a common mechanism for our Clojure extension library (it belongs to the custom *core* namespace).

3.6 Transactions

Currently there is no transactional interface in the Store engine. It may be added in the future because Store's implementation uses the low-level storage engine that fully supports transactions.

4 Implementation

The abstraction layers of Store were implemented using trusty, open and free technology.

The low-level storage engine behind Clojure is Oracle Berkeley DB Java Edition [25]. It is the same storage engine as the one used when implementing the mentioned *repository* [17], [18]. The engine is known for an excellent performance and reliability [26], [27]. It also supports transactions, making it possible to implement transactional API for Store in the near future.

Most parts of Store were implemented in Clojure. Few elements were moved to the Java level of implementation. It was mostly on the Clojure - Berkeyey DB surface.

5 Applications and observed performance

There was one major application using Store. It was an attempt to build a Wikipedia graph representation for some future research. The experiment was performed on a single PC machine running common low-cost hardware (2 cores, 2 GHz, 2MB RAM, HDD 5400 rpm, Ubuntu Linux). The experiment ended with **130 thousands** of Wikipedia **nodes** indexed and over **20 million** of **edges**. Visiting each node by an algorithm, including full deserialization takes 15-17 s. Also counting all edges takes 15-20 s. This is very optimistic time, even for the languages considered more performance-oriented.

The whole process of parsing Wikipedia pages and building all dependencies took about 70 hours. Inserting data into the database was no bottleneck. The decisive points in terms of the performance were the Internet communication (grabbing the Wikipedia web pages), parsing and some AI algorithms. During that time both algorithms (implemented also in Clojure) and Store were behaving perfectly well. There were no observable leaks of any resources (memory nor db cursors).

6 Conclusion

The presented embedded storage engine for the Clojure programming language offers a great scalability and robustness. It may be used both in research projects and in the industrial applications. Supposedly it may become the first step towards creating a more stateless solution exploiting Software Transactional Memory. Up till it helped to build a performance- and stability-demanding research application and this is its best showcase as a production-ready solution.

References

1. Turing A.M., 1936, *On computable numbers, with an application to the Entscheidungsproblem*, Proceedings of the London Mathematical Society 42 (2)
2. Church A., 1932, *A set of postulates for the foundation of logic. Annals of Mathematics*, Series 2, 33:346366
3. Backus J., 1978, *Can Programming Be Liberated from the Von Neumann Style? A Functional Style and It's Algebra of Programs*, ACM Turing Award Lecture (1977), Communications of the ACM (August 1978) vol. 2
4. Hudak P., 1989, *Conception, Evolution, and Application of Functional Programming Languages*, ACM Computing Surveys, Vol. 21, No. 3
5. Peyton Johnes S. L., 1987, *The Implementation of Functional Programming Language*, Prentice Hall International (UK) Ltd
6. Okasaki Ch., 1996, *Purely Functional Data Structures*, PhD thesis submitted to School of Computer Science Carnegie Mellon University Pittsburgh, PA 15213
7. Bagwell P., 2001, *Ideal Hash Trees*, Es Grands Champs vol. 1195
8. McCarthy J., 1960, *Recursive Functions of Symbolic Expressions and Their Computation by Machine*, CACM 3 (4), pp. 184-195
9. Abelson H., Sussman G. J., 1984, *Structure and Interpretation of Computer Programs*, ISBN 0-262-01077-1, MIT Press
10. Norvig, P., 1991, *Paradigms of Artificial Intelligence Programming: Case Studies in Common Lisp*, Morgan Kaufmann
11. Russel S. J., Norvig P., 2003, *Artificial Intelligence A Modern Approach Second Edition*, Pearson Education Inc., Upper Saddle River, New Jersey 07458
12. Seibel P., 2009, *Coders at Work*, Apress 1st edition
13. Graham P., 1993, *On Lisp - Advanced Techniques for Common Lisp*. Prentice Hall
14. Kiczales G., Rivieres J., Bobrow D.G., 1991, *The Art of the Metaobject Protocol*, MIT Press, ISBN 0-262-61074-4
15. *Clojure* Website, 2001, http://clojure.org
16. Halloway S., 2009, *Programming Clojure*, ISBN: 978-1-93435-633-3, The Pragmatic Bookshelf
17. Grzanek K., Grzybowski R., 2005, *Metody przechowywania danych w systemie rozpoznawania wzorców projektowych w oprogramowaniu,* Zeszyty Naukowe AGH, seria Automatyka, Vol. 9, Book 3, pp. 823-832
18. Grzanek K., Grzybowski R., 2007, *Implementation of the Repository for the Source Code Similarities Analysis System, Some New Ideas and Research Results in Computer Science*, Proceedings of the 2nd Polish and International PD Forum-Conference on Computer Science October 16-19, 2006 łódź, Smardzewice, Poland, Academic Publishing House EXIT, Warsaw, Vol I, Part D, pp. 373-386

19. Grzanek K., 2009, *Realizacja systemu wyszukiwania wystąpień wzorców projektowych w oprogramowaniu przy zastosowaniu metod analizy statycznej kodu źródłowego,* PhD Thesis, Wydział Inżynierii Mechanicznej i Informatyki, Politechnika Częstochowska
20. Moggi E., 1991, *Notions of Computation and Monads,* Information and Computation 93 (1)
21. *Allegro Graph*, 2010, website: http://www.franz.com/agraph/allegrograph/
22. *Elephant: A Persitent Object Database for Common Lisp*, 2010, Website: http://common-lisp.net/project/elephant/
23. *Fleet DB, Introduction*, 2010, http://fleetdb.org/docs/introduction.html
24. *Apache Derby*, 2010, website: http://db.apache.org/derby/
25. *Oracle Berkeley DB Java Edition*, 2010, website: http://www.oracle.com/database/berkeley-db/je/index.html
26. *Oracle Berkeley DB Java Edition vs. Apache Derby: A Performance Comparison*, 2010, http://www.oracle.com/technology/products/berkeley-db/pdf/je-derby-performance.pdf
27. *A Comparison of Oracle Berkeley DB and Relational Database Management Systems*, 2010, http://www.oracle.com/database/docs/Berkeley-DB-v-Relational.pdf