STORE REVISITED

Konrad Grzanek

IT Institute, Academy of Science, Łódź, Poland
kgrzanek@spoleczna.pl, kongra@gmail.com

Abstract

Building abstraction layers is the key do the creation of reliable, scalable and maintainable software. Large number of database models and implementations together with the requirements coming from agile and TDD methodologies make it even more tangible. The paper is an attempt to present features and abstraction layers of a transactional key → value persistent storage library in which the physical storage is fully transparent for a programmer and exchangeable on the run-time.

Key words: Databases, transactions, functional programming, Clojure, software architecture

1 Introduction

The database research and development is one of the key regions of both scientific and industrial activities. The advent of no-SQL databases, massive data storage and processing environments like Google BigTable, Hadoop as well as many other large and small database and storage solutions make it a great ecosystem to design and build large-scale data-processing software, but sometimes makes the design decisions harder than expected. It is not easy to choose a concrete storage solution when both the functional and non-functional requirements are hard to freeze at the initial phases of software design and development. Moreover the programmers would be so much satisfied being able to write business logic abstracting away from the details of a concrete storage. It is also very important in TDD and other agile methods.

The paper summarizes new features and some implementation details of a flexible ***Store*** library, a transactional and realization-transparent key → value storage abstraction being an enhanced version of a previously implemented solution [1]. It also gives an insight into ways of using the library and implementing new realizations.

2 Layers of Abstraction

Previous version of the ***Store*** library as described in [1] was designed and implemented with the following assumptions:

- Key → value store being the major data storage model. Effectively this meant abandoning the relational model for good.
- Query language is the application (business) logic language. Instead of SQL we use the operators and procedures of the high level programming language. This decision might appear costly with respect to joins, but the key → value store architecture does not support them anyway.
- ***A lack of transactions.***
- ***Only a single physical storage implementation using Berkeley DB Java Edition*** [3].

A diagram depicting the architecture of the original solution can be found in [1]. When approaching a ***Store*** renewal we decided to make the following improvements in various aspects of the whole:

- **Add the transaction processing**, making it an option if possible 1.
- Make no assumptions about the physical storage. There should be **various and pluggable storage realizations**.
- It should be possible to change the storage realizations on the run-time without stopping the application.
- All the key elements of the system (also the realizations) should be as loosely coupled as possible, reconfigurable on the run-time and provably safely managed (without any resource leaks).
- All the key elements of the system should exhibit solely (provably) correct multitasking behaviors.

The following Figure 1 presents the architecture of the new ***Store***[2].

1 In many transactions-supporting systems it is impossible to turn the transactions off. Instead an auto-commit mode may be used, like in the case of JDBC. Berkeley DB JE environments support turning the transactions on and off on the environment opening. When opened in a transactional mode, it also supports auto-commit [4].

2 Including Repo – a high level objects storage library. For more information see [2].

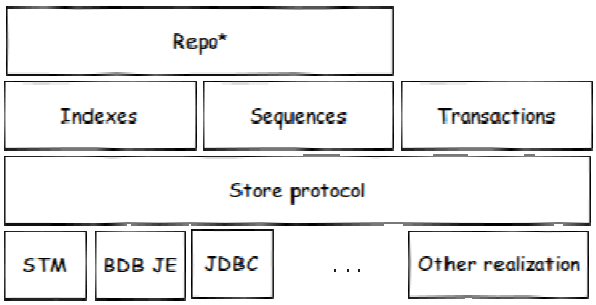

Figure 1. The architecture of the Store library

In the following sections all the elements of the architecture will be described with particular emphasis on the abstractions, but also with some implementation details. From now on we assume the following compilation unit header:

```
(ns <ns-name>
  (:use     [kongra.core])
  (:require [kongra.store :as S]))
```

3 Sequences

The nature of all objects representing the basic abstractions in ***Store*** was intended to be lightweight, i.e. their creation was supposed to be cheap in terms of CPU and memory usage. Being "lightweight" also means accessing their realization (possibly persistent, resource-heavy) is delayed for as long as possible and, the created abstraction does not hold any details of the realization within itself. This is essential with respect to the planned substitutions of realizations on the run-time.

An abstraction of a persistent, transactional sequence may be created using the following expression:

```
(S/sequence "<name>")
```

The created abstraction represents a sequence of step (delta) 1 and the initial value 1. To modify the settings another form may be used:

```
(S/sequence "<name>" <start> <step>)
```

where the parameters *start* and *step* are Clojure long integral values. The creation is pretty cheap indeed:

```
(microbench-repeat* 10 1e6 5 (S/sequence "<name>"))

Warming up for 5 [s] ...
Benchmarking...
Total runtime:  151.92165799999998 msecs
Highest time :  40.679912  msecs
Lowest time  :  8.573213  msecs
Average      :  12.833566624999996  msecs
```

That simple benchmark performed on a low-commodity machine[3] shows that the creation of 1 mln sequence handles takes circa 13 milliseconds. One is encouraged to use the form to access sequence abstraction as needed. In particular it is more elegant to use the *(S/**sequence** ...)* form in other S-expressions than to:

```
(def s (S/sequence …)
```

and then use the ***s*** symbol.

The following table shows all the sequence operators in ***Store*** with their semantics:

Table 1. Sequence operators and their semantics

Operator	Argument(s)	Meaning
next!	[seq]	Generates a next sequence value.
recent	[seq]	Returns the recently generated value.
drop!	[seq]	Deletes (drops) the passed sequence. This operation causes the underlying realization free all resources related with this sequence abstraction. The sequence abstraction is not deleted and may be used in further operations.

An example interactive sequence session (in Clojure REPL within a namespace ***stest***) can be seen below:

```
stest> (S/next!  (S/sequence "1st-seq" 0 1))
0
stest> (S/next!  (S/sequence "1st-seq" 0 1))
```

3 AMD E-450 netbook running in the *performance-on-demand* mode, 4GB of RAM, JVM settings: -Xms128m -Xmx2g -XX:MaxPermSize=256M -XX:+UseCompressedOops -XX:+UseParallelGC

```
1
stest> (S/next!  (S/sequence "1st-seq" 0 1))
2
stest> (S/recent (S/sequence "1st-seq" 0 1))
2
stest> (S/drop!  (S/sequence "1st-seq" 0 1))
OperationStatus.SUCCESS
stest> (S/next!  (S/sequence "1st-seq" 0 1))
0
stest> (S/next!  (S/sequence "1st-seq" 0 1))
1
stest> (S/next!  (S/sequence "1st-seq" 0 1))
2
```

Few things are worth mentioning here:

- There is no need to perform any explicit initialization of a sequence abstraction nor it's realization.
- The actual realization is fully transparent. The only realization-dependent value above is `OperationStatus.SUCCESS` – a result of an underlying operation of dropping the sequence in a Berkeley DB JE Store being the currently used *store*4 realization.
- No special steps have be taken between dropping the sequence and reusing it. This is a manifestation of the already mentioned overall lightweight nature of the **Store** abstractions.

Dropping a sequence that has never been referenced before also exhibits a desired behavior:

```
stest> (S/drop! (S/sequence "2nd-seq"))
OperationStatus.NOTFOUND
```

It is so in the case of a Berkeley DB JE realization and should be in all other realizations.

4 Indexes

An index is a key → value storage (mapping) abstraction. To get an access to an index one has to use an expression:

4 We use a capitalized name ***Store*** to refer to a library, and a lowe-case name *store* when talking about the possible realization(s) of Store abstractions, in fact – realizations of the kongra.store.Store *protocol* as will be described further.

```
(S/index "<name>")
```

or

```
(S/index "<name>" <key-type> <value-type>)
```

The second version evaluates to an index abstraction. Informing about *key-type* and *value-type* may be necessary in some realizations, when the serializations of key and values is non-trivial – JDBC or Berkeley DB JE realizations are the apparent examples here. The values of both *key-type* and *value-type* may be any Clojure or Java objects/values, as long as the realizations that expect them are able to recognize their meaning[5]. We strongly recommend using Java classes, Clojure *keywords* or *symbols*.

Like in the sequences case the actual realization of the index depends on the currently present *store*. There are the following index operations:

Table 2. Index operators and their semantics

Operator	Argument(s)	Meaning
get	`[index key]`	Returns a value for a given key stored in the index or nil, when no value present.
put!	`[index key val]` `[index & kvs]`	Puts the key → val entry into the index. Allows passing a sequence of key → val pairs to put them all in a single call (see example below).
del!	`[index key]` `[index & ks]`	Deletes a entry for the key. Allows passing a sequence of keys to delete them all in a single call.
drop!	`[index]`	Deletes (drops) the passed index. This operation causes the underlying realization free all resources related with this index abstraction. The index abstraction is not deleted and may be used in further operations.
clear!	`[index]`	Deletes all entries from the index.
entries	`[index]`	Returns a collection of [key value] pairs representing all entries of the index. The laziness of the returned collection depends on the *store* realization.

5 How to inform the realizations of the *key*- and *value-type* semantics is a question of the realizations, not the Store abstraction and lays beyond the scope of this paper.

To become familiar with these operations, please take a look at the following example:

```
stest> (def persons (S/index "persons" :long String))
#'stest/persons
stest> (S/get persons 77032403124)
nil
stest> (S/put! persons 77032403124 "Jan Kowalski")
OperationStatus.SUCCESS
stest> (S/get persons 77032403124)
"Jan Kowalski"
stest> (S/put! persons 77032403124 "Jan Kowalski"
                       77032403125 "Anna Kowalska")
nil
stest> (S/get persons 77032403125)
"Anna Kowalska"
stest> (S/put! persons 1 "Roman" 2 "Piotr" 3 "Jerzy")
nil
stest> (S/get persons 1)
"Roman"
stest> (S/get persons 2)
"Piotr"
stest> (S/get persons 3)
"Jerzy"
stest> (S/del! persons 1 2)
nil
stest> (S/get persons 1)
nil
stest> (S/get persons 2)
nil
stest> (S/get persons 3)
"Jerzy"
stest> (doclean (doall (S/entries persons)))
((3 . Jerzy))
```

The last expression (S/entries persons) returns a lazy sequence of entries stored in a Berkeley DB JE *store* in this case. But the iteration over the persistent entries requires opening a related *Database* cursor and so a *doclean* context is required to clean up all interconnected resources. In the case of other store realizations (e.g. Software Transactional Memory) this may not be necessary.

5 Transactions

Adding the transactions support was one of the two most important goals leading to the ***Store*** redesign and re-implementation. Table 3 summarizes the new transactional abstraction.

Table 3. Transactional operators and their meaning

Operator	Argument(s)	Meaning
`within-transaction`[6]	`[& body]`	Executes the body of code within a transaction, if there is a transaction running in the current scope. When no transaction present, runs within a new transaction.
`within-new-transaction`[6]	`[& body]`	Executes the body of code within a newly created transaction. Commits when the body executes without any errors/exceptions and rolls-back the transaction when there were errors/exceptions propagated.
`asserting-transaction`[6]	`[& body]`	Asserts the presence of a transaction and then executes the body. Raises an error when no transaction present.
`commit`	`[]`	Commits the currently running transaction. This operation should only be performed when absolutely necessary. A preferred way to go is auto-committing ***within-transaction*** or ***within-new-transaction***.
`rollback`	`[]` `[save-point]`	Aborts (rolls-back) the currently running transaction. As in the case of ***commit*** this operation should also be used only when absolutely necessary, e.g. when rolling back to a previously created save-point[7].

6 Implemented as Clojure macro [8]

7 Whether or not save-points functionality will be supported depends on a concrete underlying *store* realization.

set-savepoint	`[]`	Creates (sets) a new save-point[7].
release-savepoint	`[save-point]`	Releases (frees) a save-point[7].

Note that it is possible to use (***within-new-transaction*** …) inside a scope of an already running transaction. Syntactically this looks like nesting transactions however the actual support for nested transactions depends on a concrete underlying store realizations. Currently neither the Clojure STM nor the Berkeley DB JE realizations do not support this feature. One can expect a support for nested transactions from the JDBC realizations, as some RDBMs provide this feature[8] (like Oracle).

Anyway, the following good practices should be applied when using ***Store*** to implement a transactional system:

- By default use (***within-transaction …)*** to execute a body of code in a transactional context ensuring a presence of this context.
- Use (***asserting-transaction …)*** to execute a body of code in a transactional context, when the lack of a context is perceived as an erroneous state.
- Do not **commit** or **rollback** explicitly if possible
- Use save-points with care.
- Do not design with nested transactions8.

The following procedure written in a transactional way creates one million key → value pairs and puts them into an index.

```
(defn perftest
  []
  (let [s       (S/sequence "persons-seq" 0 1)
        persons (S/index "persons" Long String)]
    (S/within-transaction
     (dotimes [i 1e6]
       (S/put! persons (S/next! s)
                       (str "John Doe-" i))))))
```

achieving the following performance[9]:

```
stest> (time (perftest))
"Elapsed time: 32282.751323 msecs"
nil
```

8 Please, note that nesting transactions is perceived by some software and database engineers as a symptom of badly designed application.

9 The experiment was run on a previously specified machine with a Berkeley DB JE *store* realization opened with the following settings: ***transactional***, ***cache-size*** 512MB, ***cache-mode*** DEFAULT***, durability*** COMMIT_SYNC, ***lock-mode*** READ_COMMITTED (see [3], [4]).

The particular *store* realization[9] used to perform the simple benchmark above depends heavily on using the database cache:

```
stest> (room)
Used memory  : 580,949692 [MB]
Free memory  : 455,550308 [MB]
Total memory : 1036,500000 [MB]
Max memory   : 1820,500000 [MB]
nil
stest> (gc)
Used memory  : 139,630356 [MB]
Free memory  : 767,119644 [MB]
Total memory : 906,750000 [MB]
Max memory   : 1820,500000 [MB]
nil
```

The significant JVM heap usage after a garbage-collection (circa 140 MB) is a manifestation of a way the Berkeley DB JE manages, and – in fact – takes an advantage of using the cache.

Finally one final note on the *auto-commit* mode. Store realizations should support this mode. Only when it is impossible to be implemented, it may be permitted to skip this feature. Such a situation probably will never occur, as most transactional database or library providers offer some kind of an auto-commit. On the other hand, developers using the Store should be aware of the fact that running large amounts of operations in an auto-commit mode may decrease the overall systems' performance beyond the point of their usability.

6 The Store Abstraction

After presenting the high level abstractions of the ***Store*** library, it is time now to take a look at the core abstraction, namely the ***kongra.store.Store*** *protocol* (protocols as means of expression in Clojure are described in depth in [9]). The protocol definition looks as follows:

```
(defprotocol Store
  (sequence-drop!            [this seq])
  (sequence-next!            [this seq])
  (sequence-recent           [this seq])

  (index-drop!               [this index])
  (index-get                 [this index key])
  (index-put!                [this index key val]
                             [this index key val kvs])
  (index-del!                [this index key]
                             [this index key ks])
```

```
  (index-clear!              [this index])
  (index-entries             [this index])

  (tx-within-transaction     [this body])
  (tx-within-new-transaction [this body])
  (tx-asserting-transaction  [this body])
  (tx-commit                 [this])
  (tx-rollback               [this] [this savepoint])
  (tx-set-savepoint          [this])
  (tx-release-savepoint      [this savepoint]))
```

To provide a *store* (*kongra.store.Store* protocol) realization a provider has to do the following:

– introduce a new type whose object (or objects) would represent the realization,
– implement all the protocol methods.

A part of an example realization (with Berkeley DB JE) looks like below:

```
(defrecord ^:private JEStore [env])
(def INSTANCE (JEStore. (JE/env)))

(extend-protocol S/Store
  JEStore

  (sequence-drop!  [this s] …)
  (sequence-next!  [this s] …)
  (sequence-recent [this s] …)

  (index-drop! [this index]     …)
  (index-get   [this index key] …)

  (tx-within-transaction [this body] …)
  (tx-within-new-transaction [this body] …)

  (tx-asserting-transaction [this body]…)

  (tx-commit [this] …)

  (tx-rollback
    ([this]            …)
    ([this savepoint]  …))

  (tx-set-savepoint [this] …)

  (tx-release-savepoint [this savepoint] …))
```

The Berkeley DB JE *store* realization is the default one. Getting the current *store* realization is possible with a `(S/store)` call. As it was mentioned earlier, there are two ways to change the realization during the run-time. One is calling `(S/set-store! <store>)`. This causes a persistent, all-threads change of the default *store* realization. Another way is to use a dynamic binding of a store to a new one and executing a body of code within this newly established binding, e.g.:

```
(time (S/with-store STM/INSTANCE
                    (perftest)))
"Elapsed time: 8603.678825 msecs"
nil
```

Now, the *perftest* was executed with a Software Transactional Memory realization represented by ***STM/INSTANCE*** value. The operation results in much lower execution time (even considering the influence of database caching in the previous example) and larger memory consumption:

```
stest> (gc)
Used memory  : 297,057205 [MB]
Free memory  : 630,130295 [MB]
Total memory : 927,187500 [MB]
Max memory   : 1820,500000 [MB]
nil
```

Certainly the biggest win with this approach is the ability to change the target *store* realizations algorithmically on the run-time. It is the other most significant benefit coming from the ***Store*** redesign and re-implementation.

7 Parallelism of Abstractions and Realizations' Details

The most important problem arising within the ***Store*** library and the *store* realizations as seen from the point of view of potential users (programmers and system designers) is the complexity of configuration behind the particular realizations. With a naive approach we would be tempted to put as much configuration details into the *kongra.store.Store* protocol, in fact – trying to find a kind of a "greatest common denominator" for all possible configuration options. Even in the first look this seems a bad design. The new ***Store's*** implementer decided to go a completely different way, eliminating all configuration out of the ***Store*** abstractions and leaving them in current and future realizations. This convenient and flexible approach gives a rise of a problem of a loss of the abstract nature of source codes written using ***Store***. Codes using

the library should abstract away from the details of the realizations. One (and currently preferred) approach is to put all realization-dependent codes in a place dedicated to build a kind of a configuration context and the abstract business logic codes in all other places (procedures, modules) and then to connect them transparently using the dynamic variables and in-thread bindings. To get a sample how this can be done, please take look at the following piece of code:

```
(BDBJE/with-lock-mode BDBJE/LOCK-MODE-READ-UNCOMMITTED
     (BDBJE/with-lock-timeout 100 ;; msecs
       (perftest)))
```

where the locking mode present in the Berkeley DB JE wrapper library for Clojure (not even the *store* realization) is defined like below:

```
(def LOCK-MODE-DEFAULT           LockMode/DEFAULT)
(def LOCK-MODE-READ-COMMITTED    LockMode/READ_COMMITTED)
(def LOCK-MODE-READ-UNCOMMITTED LockMode/READ_UNCOMMITTED)
(def LOCK-MODE-RMW               LockMode/RMW)

(dyndef *lock-mode* nil)

(defn lock-mode
  []
  (dynval *lock-mode*))

(defn set-lock-mode!
  [lmode]
  (dynset! *lock-mode* lmode))

(defmacro with-lock-mode
  [lmode & body]
  `(binding [*lock-mode* ~lmode] ~@body))
```

8 Clojure STM Realization

The Software Transactional Memory is a modern approach to mutual-exclusion problems in multitasking environments. Clojure provides it's realization with references, atoms, agents, dynamic variables and persistent data structures [8], [9]. We decided to use STM for a *store* realization, even though STM lacks durability feature of a full ACID transactional system (all data is stored in RAM). It's unquestionable advantage is speed.

The major STM *store* realization features are:

– A *sequence* is a name → value pair within a mapping reference.
– An *index* is a mapping reference.

- No support for nested transactions, as the STM does not support them10.
- No support for save-points[10].
- No support for an explicit ***commits***[10].
- The explicit ***rollbacks*** are possible but not to the save-points10, only parameter-less.
- Data do not undergo any conversions on the read/write. The types declared on referring to an *index* are not used. This increases the operation speed, but eliminates type-safety by disabling any type-checks.

9 Berkeley DB JE Realization

The Berkeley DB Java Edition library [3], [4] offers full ACID stack, it also allows to operate in a non-transactional mode when properly configured on the opening-time. Our BDB JE *store* realization has the following properties:

- No support for save-points10, as in the case of the STM realization.
- Support for explicit commits and rollbacks, but not to the save-points[10].
- A support for nested transactions in the *store* realization layer. However the Berkeley DB JE library does not support this feature by now. Whether or not it will be supported is a question of the future.

10 A Discussion on Possible Future Realizations

The most promising, yet not existing realization of the ***Store*** library abstractions is the one using JDBC and relational databases. There are many ways a key → value storage may be implemented within a relational model. Readers are encouraged to take part in a research on the most effective ways of implementing this realization and also in design and programming phase.

Another interesting way to go is to use the Hadoop distributed data storage and processing engine. This direction is particularly important, as the massive storage and distributed data processing seems to be the central point in current and future database research and production use.

References

1. Grzanek K., 2010, *Store: Embedded Persistent Storage For Clojure Programming Language*, Journal of Applied Computer Science Methods No. 1 Vol. 2 2010, pp. 83

10 A Log4J warning is emitted on an attempt to use the feature.

2. Grzanek K., 2011, *Repo: High-Level Persistence Layer for Clojure*, Journal of Applied Computer Science Methods No. 2 Vol. 2 2011, pp. 5-16
3. *Oracle Berkeley DB Java Edition*, 2013, Website: http://www.oracle.com/technetwork/database/berkeleydb/overview/index-093405.html
4. *Oracle Berkeley DB Java Edition, Getting Started with Transaction Processing*, 2008, Release 3.3
5. DeCandia G., Hastorun D., Jampani M., Kakulapati G., Lakshman A., Pilchin A., Sivasubramanian S., Vosshall P., Vogels W., 2007, *Dynamo: Amazon's Highly Available Key-value Store*, SOSP'07, October 14–17, 2007, Stevenson, Washington, USA, pp. 205-220
6. Schütt T., Schintke F., Reinefeld A., 2008, *Scalaris: reliable transactional p2p key/value store*, Proceedings of the 7th ACM SIGPLAN workshop on ERLANG, pp. 41-48
7. Lim H., Fan B., Andersen D., Kaminsky M., 2011, *SILT: a memory-efficient, high-performance key-value store*, Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles, pp. 1-13
8. Halloway S., 2009: *Programming Clojure*, ISBN: 978-1-93435-633-3, The Pragmatic Bookshelf
9. Emerick Ch., Carper B., Grand Ch., 2012, *Clojure Programming*, O'Reilly Media Inc., ISBN: 978-1-449-39470-7