**Temporal in memory data store challenge – solution**

**Clarification of requirements**

The specification included several flaws (presumably arranged on purpose, to check if candidate is capable enough to spot holes and react) so I first needed to make some ad-hoc decisions. All of them are enumerated here:

* GET command may query for timepoint where no value is defined, as would happen in the following interaction:

CREATE 1 10 xyz

*OK xyz*

GET 1 5

This situation is not covered by the spec. I decided to signal error (by using prefix “ERR”) in this case, so my implementation will behave like:

CREATE 1 10 xyz

*OK xyz*

GET 1 5

*ERR value is not defined for this timepoint*

* It is possible to delete all history entries. After successful deletion - is the identifier going to be still registered or not ? Example interaction:

CREATE 1 10 xyz

*OK xyz*

DELETE 1

*OK xyz*

CREATE 1 10 abc

What the response should be ? This situation is not covered by the spec. I decided to never de-register an identifier. So even if the history gets empty, the identifier is still there. In other words my “DELETE 1” command operates like “remove all history entries for identifier 1, but leave the identifier registered”. My implementation will behave like this:

CREATE 1 10 xyz

*OK xyz*

DELETE 1

*OK xyz*

CREATE 1 10 abc

*ERR A history already exists for identifier 1*

* UPDATE command has to return previous value seen for updated time point. But what should happen if there was no value seen at this point ? Example script:

CREATE 1 10 xyz

*OK xyz*

UPDATE 1 5 abc

I decided to return empty string as a “value” seen previously. So the behaviour will be:

CREATE 1 10 xyz

*OK xyz*

UPDATE 1 5 abc

*OK* <------- NOTICE EMPTY STRING AFTER “OK”

* DELETE command behaviour also was underspecified for corner cases where no reasonable value can be returned. I decided to apply the same approach as for UPDATE, i.e. I use empty string where there is no previously visible value.

Example 1:

CREATE 1 10 xyz

*OK xyz*

DELETE 1

*OK xyz*

DELETE 1

OK <------- NOTICE EMPTY STRING AFTER “OK”

Example 2:

CREATE 1 10 xyz

*OK xyz*

DELETE 1 5

*OK* <------- NOTICE EMPTY STRING AFTER “OK”

DELETE 1 5

OK <------- NOTICE EMPTY STRING AFTER “OK”

* LATEST command can be invoked against empty history. In such case I signal an error. Example:

CREATE 1 10 xyz

*OK xyz*

DELETE 1

*OK xyz*

LATEST 1

*ERR history is empty*

**Small improvements (that made life easier)**

* I allow running the app in two modes:
  + batch mode (this is precisely what was defined in requirements)
  + interactive mode (allows the user to interactively type commands and observe results)

Implementation-wise differences between modes are very minimal: interactive mode just adds simple prompt character printing (>) and actually waits for the user as it types new command. In batch mode the program exits where there is no more commands (as was required).

* I ignore uppercase/lowercase for command keywords, so CREATE, create, CrEaTe and CREate will work.
* I added LIST command. It shows the whole history for given identifier (very useful for testing in interactive mode). Example:

CREATE 1 10 x

*OK x*

UPDATE 1 20 y

*OK y*

UPDATE 1 15 z

*OK z*

LIST 1

*OK 10->x,15->z,20->y*

* I added two config parameters that allow adjusting the performance. For explaining these parameters, check next chapter.

**Main idea of the solution and**

Implementation is based on a (possibly large) hash map:

identifier -------------> history container

For implementing history containers I had a strong hesitation which implementation should I choose:

* array buffer with binary search approach (which I considered “elementary”)
* use SortedMap implementation that is available in Scala standard library

Further investigation on the subject of SortedMap implementations revealed that Scala interestingly “diverged” from its java roots in this aspect, and you have:

* java.util.TreeMap, which is a mutable, JDK implementation of SortedMap (internally based on red-black trees)
* scala.collection.immutable.TreeMap, which is immutable pure Scala impelementation (internally also based on red-black trees)

There is no mutable SortedMap in Scala, and there is no immutable SortedMap in JDK. Moreover, java provides a very interesting extension of SortedMap called “NavigableMap”, which looked quite useful. So finally I decided to implement all 3 approaches and compare them.

**Startup parameters explained**

There are two startup parameters (both are mandatory):

* the initial size of main hash table (bigger values means less frequent resizing, means better performance); a rule of thumb here is to provide the best estimate of number of identifiers that are going to be stored in the buffer
* the implementation of HistoryContainer to be used for storing history of each identifier; there are 3 accepted values here:
  + binary-search
  + scala-treemap
  + java-treemap

**Prerequisites**

I tested this solution against Scala 2.11.7 so I recommend this version. I was testing also against Scala 2.11.1 and (quite interestingly) the application was not working (due to a bug in Scala base library).

Scala 2.11.7 does not work with Java 8. For this reason latest release of Java 7 is required.

**Running the solution (interactive mode)**

The entry point to interactive mode is class:

com.heuriqo.daiwatest.TemporalBufferAppInteractiveMode

Assuming scala executable is in search path and solution.jar is placed in current directory, the following command will launch the program:

scala – classpath solution.jar 100 100 binary-search

Please notice the configuration parameters passed to the program:

* first parameter is the initial size of “identifiers--->containers” hashmap, we passed value **100**
* second parameter is implementation of history container to be used, we selected **binary-search** implementation

**Running the solution (batch mode)**

The entry point to interactive mode is class:

com.heuriqo.daiwatest.TemporalBufferAppBatchMode

For running in batch mode, it is usually useful to use a file containing commands and redirect output to another file.

Assuming:

* scala executable is in search path
* solution.jar is placed in current directory
* text file with commands is named script.txt and is also placed in current directory

... the following command will launch the program and run the script (I am using here Ubuntu syntax for redirecting input/output):

scala – classpath solution.jar com.heuriqo.daiwatest.TemporalBufferAppBatchMode 100 java-treemap < script.txt > result.txt

In this example invocation we instructed the program to initialize main hash table size for 100 identifiers and to use java-treemap implementation of history containers.

**Browsing source code and building (if needed)**

Source code is well documented (scaladoc).

Directories structure:

src

main

java

resources

scala

com

heuriqo

daiwatest <--------- here you with find both classes with ‘main’ method

datastore <------- data layer implementation

interpreter <----- commands-processing implementation

test

java

resources

scala

com

heuriqo

daiwatest

integration <--------- one integration test is here

performance <--------- HistoryContainer benchmarks are here

unit <--------- unit tests are here

The test suite is pretty comprehensive (as I assumed ability to provide good test coverage was part of the exercise).

Unit tests:

* TemporalDatastoreSpec – covers the API of temporal datastore
* CommandsParserSpec – covers the API of commands parser
* HistoryContainerSpec – covers the API of history containers

Integration tests:

* there is only one integration test implemented, which actually directly follows the interaction presented in the task description

Performance tests:

* there is a small suite of benchmarks measuring performance of all 3 implementations of history containetrs

**Compiling source code (if needed)**

Best way for compiling source code is to use SBT: http://www.scala-sbt.org/

SBT project definition (build.sbt file) is included in sources. There is only one external dependency, which is for using ScalaTest library (for unit tests). SBT will resolve this dependency for you.

Using SBT you can also run all unit tests.

**Comparing performance of algorithms**

Maybe the most interesting part is the performance comparison of history containers, I made.

First let me precise the scope and limitations of my tests:

* I was testing only history containers in isolation (= outside the context of temporal datastore)
* I was not measuring memory consumptions, I focused on performance
* I was using System.nanoTime for measuring performance of individual API calls (which is not perfect for reasons discussed in java documentation)

Another important issue is the pattern of request stream, given HistoryContainer is exposed against. Saying it is “random” is of course far not enough, because there are several aspects on data level that may lead to completely different results. Just to point out some of them:

* relative frequencies of query vs clear vs update requests
* probabilistic distribution of time point for query / clear / update requests
* typical size of history entries collection

In my benchmarks I fixed some ad-hoc model of how I generate sequences of API calls against given HistoryContainer instance:

* queries are 2 times more frequent than inserts
* I am generating always 2 millions of API calls
* deletes are rare

For details go to classes in package **com.heuriqo.daiwatest.performance**.

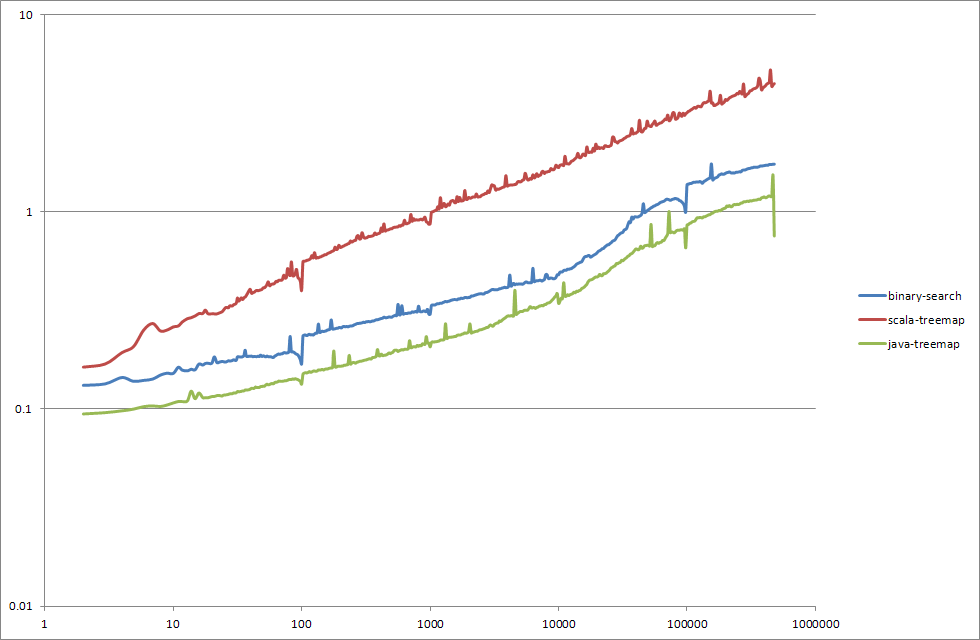
The following diagrams give some insight on the performance results I measured.

X axis shows the size of HistoryContainer (number of entries) at the moment of API call.

Y axis shows average time of API call (in microseconds).

UPDATE is one API call, but I decided to separate the results by the type of update operations: INSERT means that a new timepoint was added to history, UPDATE means that a value was overriden.

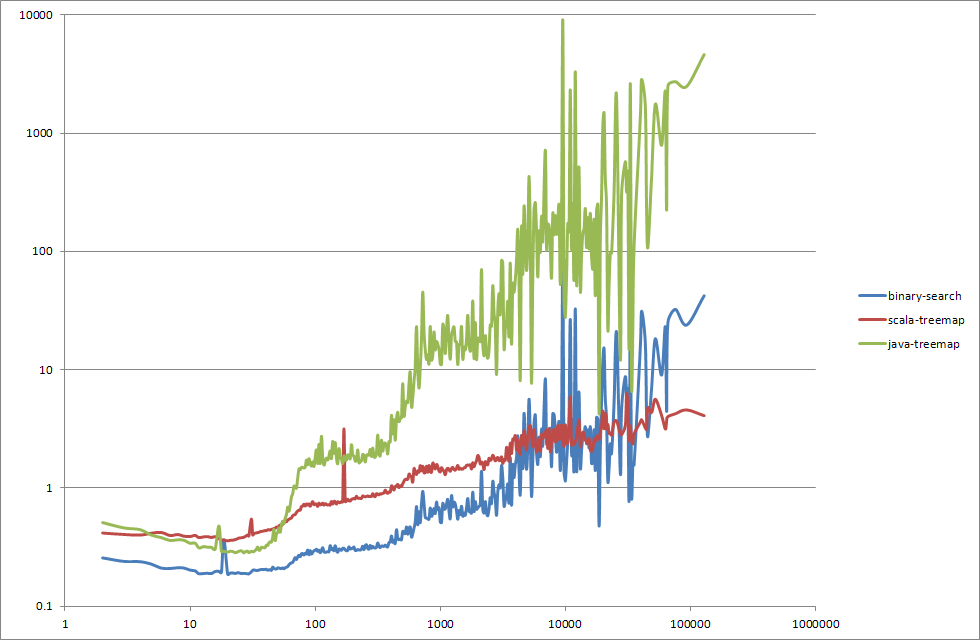
**Benchmark results - QUERY**



**Benchmark results - INSERT**

**Benchmark results – UPDATE**

**Benchmark results – CLEAR**



**Benchmark – mixed test**

Class BatchProcessingSimulation provides a convenient environment for testing a long sequences of API calls. Example output produced by this class:

number of API operations: 10000000

timepoints range: 10000

queries-to-updated ratio: 2.0

clear-from invoked with probability: 1/10000

starting benchmark for: array buffer implementation, initial size = 10

number of operations = 10000000

results:

total time [seconds] = 5.602

number of entries in container when test finished: 2294

insert: invocations = 1825185 averageTime [micro-seconds] = 0.602606313332621

update: invocations = 1505638 averageTime [micro-seconds] = 0.41654879658988414

clear from: invocations = 1003 averageTime [micro-seconds] = 2.192437686939183

query: invocations = 6668174 averageTime [micro-seconds] = 0.42132197840068364

starting benchmark for: array buffer implementation, initial size = 1000

number of operations = 10000000

results:

total time [seconds] = 5.527

number of entries in container when test finished: 2294

insert: invocations = 1825185 averageTime [micro-seconds] = 0.5905546555554643

update: invocations = 1505638 averageTime [micro-seconds] = 0.4151664928754455

clear from: invocations = 1003 averageTime [micro-seconds] = 2.164035892323031

query: invocations = 6668174 averageTime [micro-seconds] = 0.41313546182208205

starting benchmark for: scala (immutable) TreeMap implementation

number of operations = 10000000

results:

total time [seconds] = 16.279

number of entries in container when test finished: 2294

insert: invocations = 1825185 averageTime [micro-seconds] = 1.7576341154458315

update: invocations = 1505638 averageTime [micro-seconds] = 1.8546809551831185

clear from: invocations = 1003 averageTime [micro-seconds] = 3.0105004985044865

query: invocations = 6668174 averageTime [micro-seconds] = 1.3662937490833322

starting benchmark for: java (mutable) TreeMap implementation

number of operations = 10000000

results:

total time [seconds] = 4.981

number of entries in container when test finished: 2294

insert: invocations = 1825185 averageTime [micro-seconds] = 0.4988593243972529

update: invocations = 1505638 averageTime [micro-seconds] = 0.49378755052675344

clear from: invocations = 1003 averageTime [micro-seconds] = 105.33473280159521

query: invocations = 6668174 averageTime [micro-seconds] = 0.3203449932470268

**Benchmark – conclusions**

Surprisingly good results for java-treemap came (in my opinion) from the fact that NavigableMap interface was especially useful for us. However this implementation provides really poor performance of clear operation (in this can’t be improved I am afraid).

Binary search appears to be quite good if history size is below 8000 entries.

SortedMap interface in Scala flavour is somewhat poor and expensive workarounds were needed to overcome this – hence the scala is generally behind others. Biggest advantage is stable behaviour of this implementation among wide spectrum of conditions (and excellent clear performance !). For sure is it also most promising for parallel processing (outside of the scope of this exercise). And I am pretty sure that some more deep hacking around scala implementation (inspired by NavigableMap interface) would provide possibly best results.