CHAPTER 14

Further AllegroCache

Searching the Database

This chapter builds on the last one and continues our exploration of AllegroCache. We've already covered in some detail the various ways you can get information into the database and modify it once it's there; now we really ought to look at how to get your data out. After that we'll move on to a brief trot through some more advanced topics including database administration.

Cursors

We've already met the macro `doclass` which iterates over every instance of a class in your local cache:

```
(let ((sentences nil))
  (doclass (sentence 'sentence)
    (push (sentence-text sentence) sentences))
  sentences)
=>
("We have to look at how to get your data out."
 "Then we'll take a brief look at some more advanced topics.")
```

Here's another way of doing it.

```
CL-USER(31): (setf cursor (create-class-cursor 'sentence))
#<DB.ALLEGROCACHE::CLASS-CURSOR @ #x214a7e1a>
CL-USER(32): (next-class-cursor cursor)
#<SENTENCE oid: 1011, ver 5, trans: 8,  not modified @ #x2146dcca>
CL-USER(33): (next-class-cursor cursor)
#<SENTENCE oid: 1012, ver 5, trans: 8,  not modified @ #x2146dce2>

;; The following NIL means there are no more sentences and the search is
;; over:

CL-USER(34): (next-class-cursor cursor)
NIL
CL-USER(35): (free-class-cursor cursor)
```

```
NIL
CL-USER(36):
```

When you don't need the cursor any more, you should call `free-class-cursor` to release its resources.

Exercise

Working from the example above, write a function which uses a cursor to collect all the instances of a class. How will you guarantee that `free-class-cursor` will be called on the way out?

At first sight, all we've done is add unnecessary baggage: it seems unlikely that your cursor function will look as elegant as using `doclass` would. However class cursors have a couple of advantages which are worth consideration. The first is that they can return an *object identifier* rather than the object itself:

```
CL-USER(41): (next-class-cursor cursor :oid t)
1011
CL-USER(42):
```

We'll come back to this and its uses later. The other gain is more immediately noticeable: we can pass the cursor around as a Lisp value.

```
(defmacro with-class-cursor ((fun-var class) &body body)
  (let ((cursor-var (gensym "CLASS-CURSOR-")))
    `(let ((,cursor-var (create-class-cursor ,class)))
       (unwind-protect
           (catch ',cursor-var
                  (flet ((,fun-var ()
                                   (or (next-class-cursor ,cursor-var)
                                       (throw ',cursor-var nil))))
                    ,@body))
         (free-class-cursor ,cursor-var)))))

(with-class-cursor (next-sentence 'sentence)
   (loop (print (sentence-text (next-sentence)))))
```

Exercise

The symbol stored in `cursor-var` is used two different ways. Is there anything wrong with this? Explain.

We can now go one step beyond `doclass` and free ourselves from the lexical scope of the loop, as in this example from a fictitious GUI:

```
(make-instance 'menu-item
               :text "View next sentence"
```

```
                    :callback (lambda ()
                                (view-text (sentence-text (next-sentence)))))
```

If there are no instances of your class in the cache, `create-class-cursor` is allowed to return `nil` and your code should check for that.

Exercise

Modify `with-class-cursor` to bind `fun-var` to a `lambda` which is still safe in these circumstances.

Indexes

An *index* is a simple way of filtering and ordering the results of a search. We have to flag the slot(s) which we want to be indexed; as the following example shows we needn't get around to doing this until after creating the objects concerned.

```
;; Modify previous class definition, specifying :INDEX :ANY-UNIQUE for the
;; TEXT slot.

(defclass sentence ()
  ((text :initarg :text :accessor sentence-text :index :any-unique)
   (next :initform nil  :accessor sentence-next))
  (:metaclass persistent-class))

;; Write the new index to the database. This will update existing instances
;; and we can keep on working with them.

(commit)
```

Note that when a slot is indexed as `:any-unique`, its value cannot appear in more than one instance of that class. To drop that restriction, specify `:index :any`.

Let's try out some searches. Give the function `retrieve-from-index` a class, the name of an indexed slot, and a value for that slot; it'll return the object which has this slot-value (or `nil` if there's no matching object).

```
(retrieve-from-index 'sentence 'text "Hello, World.")
=>
NIL

(retrieve-from-index 'sentence
                     'text "We have to look at how to get your data out.")
=>
#<SENTENCE oid: 1011, ver 6, trans: 19,  not modified @ #x214452fa>
```

We can put this index to use, building a set of objects which links Lisp symbols to the sentences which refer to them and ensuring that the sentence references are unique (see the call to `retrieve-from-index` in function `make-links` below).

```
;; Warning! I'm only showing a small part of my data here.
;; The full list is available via http://lisp-book.org/

(defparameter *chapter-13-links*
  '(("doclass" .
     "It's the macro doclass which has the following syntax.")

    ("doclass" .
     "Doclass returns nil; you can use (return ...) to terminate the loop
      early and return more interesting values.")

    ("return" .
     "Doclass returns nil; you can use (return ...) to terminate the loop
      early and return more interesting values.")

    ...
    ))

;; This function uses the index on TEXT for the class SENTENCE.

(defun make-links (links)
  (loop for (term . sentence-text) in links do
        (let ((sentence (or ;; Is this sentence already in the cache?
                            (retrieve-from-index 'sentence
                                                 'text sentence-text)

                            ;; If not, make a new one.
                            (make-instance 'sentence
                                           :text sentence-text))))

          ;; Make a persistent object linking the term ("doclass", etc) to
          ;; the sentence.

          (make-instance 'link
                         :term term
                         :sentence sentence)))

  ;; End of the loop - commit the changes.

  (commit))

;; Persistent class indexed on slot named TERM.

(defclass link ()
  ((term     :reader link-term     :initarg :term :index :any)
   (sentence :reader link-sentence :initarg :sentence))
  (:metaclass persistent-class))

(make-links *chapter-13-links*)
```

Exercise

In `make-links`, why does `retrieve-from-index` take `'text` and `make-instance` take `:text`?

We use the index which we defined in the class `link` to retrieve data. Because it's an `:any` index, each slot value (such as `"doclass"`) may be used in more than one instance. If we specify `:all t` to `retrieve-from-index` we'll get back a list of all these objects.

Recall from our raw data (`*chapter-13-links*` above) that one of our sentences was linked to by both `"doclass"` and `"return"`. We expect the search results on either of these terms to include that sentence (and it's the one which prints here with `oid: 2366`).

```
(retrieve-from-index 'link 'term "doclass" :all t)
=>
(#<LINK oid: 2367, ver 5, trans: 101,  not modified @ #x21281922>
 #<LINK oid: 2365, ver 5, trans: 101,  not modified @ #x2128190a>)

(mapcar 'link-sentence *)
=>
(#<SENTENCE oid: 2366, ver 6, trans: 101,  not modified @ #x21281742>
 #<SENTENCE oid: 2364, ver 6, trans: 101,  not modified @ #x2128172a>)

(mapcar 'link-sentence (retrieve-from-index 'link 'term "return" :all t))
=>
(#<SENTENCE oid: 2366, ver 6, trans: 101,  not modified @ #x21281742>)
```

Exercise

Persistent classes can have any number of indexed slots. Try out a class with two indexes.

Object Identifiers

Associated with every object in the cache there is a number which serves as a unique identifier. Output generated by the default `print-object` method for persistent objects includes this value, for instance `oid: 2366` in the example above. You can obtain an object's identifier with the function `db-object-oid` and given an object's class and its identifier you can use `oid-to-object` to find the object:

```
CL-USER(51): (retrieve-from-index 'link 'term "return")
#<LINK oid: 2369, ver 5, trans: 101,  not modified @ #x21284c4a>
CL-USER(52): (db-object-oid *)
2369
CL-USER(53): (oid-to-object 'link 2369)
#<LINK oid: 2369, ver 5, trans: 101,  not modified @ #x21284c4a>
CL-USER(54):
```

Most of the search functions described in this chapter take a `:oid` keyword parameter; when this is non-`nil` the search returns identifiers rather than the objects to which these belong. Identifiers could be the basis for making "join" queries more efficient: retrieve the identifiers of instances satisfying one constraint, do this again with a second constraint, use `intersection` to combine the queries and finally `oid-to-object` to get to the objects themselves. See however “Expression Cursors” on page 7 later in this chapter for a better approach.

Exercise

The Allegro CL function `arglist` returns the argument list of a symbol’s function definition. For example, `(arglist 'create-expression-cursor) => (CLASS DB.ALLEGROCACHE::EXPRESSION &KEY DB.ALLEGROCACHE::DB)`. Look up how the macro `cl:do-external-symbols` works and use it to apply `arglist` to all `fboundp` external symbols in the `DB.ALLEGROCACHE` package. Come up with a list of functions which take the `:oid` keyword. While you’re at it, find out which functions take `:db`.

Exercise

In the example call to `arglist` above we get `DB.ALLEGROCACHE::DB` rather than `DB`? What’s going on? Does it matter?

Object identifiers are a valuable debugging resource. Bear this in mind if you define `print-object` methods on your own persistent classes.

Exercise

`print-object` specializes on the class of the object you’re printing. Even though the persistent class definitions we’ve used have not specified any superclasses, what would you now expect `(:metaclass persistent-class)` to have done? How can you check this?

Exercise

Suppose you know an object’s identifier but not its class. Use a class from which all objects are bound to inherit and the function `oid-to-object*` (which descends through subclasses) to find that object and hence its class.

Sorting by Index Values

The function `retrieve-from-index-range` lists objects with a range of values for a given index (as opposed to objects with one specific value). It returns a sorted list of all instances of the class whose values for that slot are in the range (including the start, excluding the end). So here we retrieve every `link` whose `term` slot begins with the character `#\c`:

```
(mapcar 'link-term (retrieve-from-index-range 'link 'term "c" "d"))
=>
```

```
("cl:open" "close-database" "close-database" "commit" "commit" "commit"
 ...)
```

For both the filtering and the sorting operations, numbers (integers and floats in increasing numerical order) come before strings. The target slot may take "other" values in which case it'll be sorted ahead of both numbers and strings. With one exception such other values may not be specified for the `start-` and `end-value`. The exception is that you can pass `nil`, which means "don't limit the values". So the following call:

```
(retrieve-from-index-range 'link 'term nil nil)
```

returns a list of every `link` in the cache, sorted by `term` values.

Alternatively you can use an *index cursor* and combine the filtering and sorting of indexes with the flexibility which we met before with class cursors. Create one with `create-index-cursor` (this takes a class and and an indexed slot-name; specify start and end values if you wish with the keywords `:initial-value` and `:limit-value`). Exactly as with class cursors, you have `next-index-cursor` to return the next object and `free-index-cursor` to free resources at the end.

Exercise

Without looking at the AllegroCache Reference Manual, use a cursor to implement `retrieve-from-index-range`.

Expression Cursors

In this context an *expression* allows you to specify either a value or a range of values for a given slot, or to make logical combinations of other expressions using `(and ...)` or `(or ...)`. This allows you to join together queries on different slots in a single object.

Purely to illustrate these features, we'll add another slot to the class `link`:

```
(defclass link ()
  ((term     :reader link-term     :initarg :term :index :any-unique)
   (sentence :reader link-sentence :initarg :sentence)
   (text     :accessor link-text))
  (:metaclass persistent-class))

(doclass (x 'link)
  (setf (link-text x) (sentence-text (link-sentence x))))

(commit)
```

To find all `link`s whose `term` is the string `"rollback"` and whose `text` starts with the letter `#\A` we could build a cursor as in the following example. Once more, we'll use `next-index-cursor` to return the cursor's next object and `free-index-cursor` to release resources at the end.

```
(let* ((expression '(and (= term "rollback")
                         (:range text "A" "B")))
       (cursor (create-expression-cursor 'link expression))
```

```
        (links nil))
    (loop (let ((link (or (next-index-cursor cursor)
                          (return))))
            (push link links)))
    (free-index-cursor cursor)
    (mapcar 'link-text links))
  =>
  ("Any uncommitted changes will be dropped and the objects concerned rewound
    to their state at the last rollback or commit."
   "Alternatively, you can call rollback and all your changes (since the last
    commit or rollback) will be abandoned.")
```

As this example shows, expression cursors work on slots which haven't been indexed. However running on indexed slots is faster and you should order your expressions so that indexed values are tested first. (Why?)

Other Storage Classes

In the last chapter we glossed over AllegroCache's answer to hash tables, and how it stores non-persistent objects. Let's come back to these now.

Maps

Think of an AllegroCache *map* as a persistent hash table. To create a map, make an instance of the class `ac-map-range`. Use the accessor `map-value` to get values in and out; use the functions `map-map`, `map-count` and `remove-from-map` the same way as you'd use Common Lisp's `maphash`, `hash-table-count` and `remhash`.

```
(let ((cursor (create-class-cursor 'link))
      (map (make-instance 'ac-map-range)))

  ;; Use the cursor to walk over all instances of LINK and
  ;; insert them into the map.

  (loop for link = (next-class-cursor cursor)
        while link
        do
        (push (link-sentence link)
              (map-value map (link-term link))))
  (free-class-cursor)

  ;; You should commit the map and its values before using it.

  (commit)

  (values (map-count map)

          ;; What are the sentences linked by "persistent-class"?

          (map-value map "persistent-class")

          ;; Return one of the terms and count how many times it occurs.
          (block nil (map-map (lambda (term sentences)
                                (return (cons term (length sentences))))
                              map))))
=>
31
(#<SENTENCE oid: 2354, ver 6, trans: 102,  not modified @ #x21269462>)
("*allegrocache*" . 2)
```

Like `gethash`, `map-value` returns a second value which tells us whether or not the value was found. Unlike `gethash`, the arguments to `map-value` are in the more canonical order for Lisp accessors: first the map, then the value.

Maps also support `retrieve-from-map-range` (like `retrieve-from-index-range`) and cursors (like index cursors: use `create-map-cursor`, `next-map-cursor` and `free-map-cursor`).

Encoding and Decoding

Although it's easiest to work with CLOS instances which are persistent, you can also add non-persistent objects to your cache. There are three hoops through which you have to jump to make this work.

- You only ever need to define each persistent class once; the cache remembers the definitions for you and hands them over to future connections. Non-persistent classes are not stored in the cache and you'll have to repeat their definitions in each Lisp image which refers to them.
- You'll have to define methods on `encode-object` in the image which stores your non-persistent instances. These methods should take one argument (the object which is to be stored; you'll want to specialize on this) and should return a list of values from which that object can be reconstructed.
- You'll have to define methods on `decode-object` in every image which retrieves non-persistent objects. This time each method takes an instance of the class to be reconstructed and the list of values generated elsewhere by `encode-object`. Your method has to make a *new* instance of the class concerned, fill in its instance slots, and return it.

For example, suppose we have the following non-persistent class definition: `(defclass word () (text id previous next))` where the `text` slot is known always to be a string and the other slots are numbers. Then we might define:

```
(defmethod encode-object ((self word))
  (loop for slot in '(text id previous next)
        collect (slot-value self slot)))

(defmethod decode-object ((self word) values)
  (let ((word (make-instance (class-of self))))
    (loop for slot in '(text id previous next)
          as value in values
          do
          (setf (slot-value word slot) value))
    word))
```

Exercise

Why not just `(make-instance 'word)`? What two things did we gain by passing a throw-away instance as first argument?

Exercise

What might you do about encoding a `word` when some of its slots are unbound?

Exercise

Suppose instead that `previous` and `next` are going to be other instances of the class `word`. What's the problem with this? Sketch a solution involving maps (or anything else that comes to mind). You'll need a fairly ugly hack to deal with pointers to `word`s which haven't been retrieved yet.

So why not just make everything persistent? Playing devil's advocate,

You might not be able to:
: There might be reasons why you cannot make your object class an instance of `persistent-class`. (Although we weren't planning to talk about `defstruct` we'll mention here that it generates classes which cannot inherit from `standard-class` and in particular not from `persistent-class` either.)

You might not want to:
: You might have a class with very many instances most of which you don't need in the cache, and you might not want to take the performance hit of very many unnecessary transactions.

Moving On

There are two outstanding topics to cover here: database administration and alternative approaches to persistence. Before we get to these, we need to issue:

A Warning

AllegroCache connections are not *thread-safe*: the consequences are undefined (but probably not good) if two threads "use the same connection" at the same time. Considering the space of all libraries, this problem is not altogether uncommon and will hit you from time to time whether or not they were written in Lisp. We haven't covered multi-threaded use of Lisp yet; it's a major topic and the subject of the next chapter. At the end of that we'll be able to come back briefly to AllegroCache and deal with this difficulty. For now, note that "using a connection" includes: creating, deleting, or accessing the slots of, a persistent object; using an index; and calling `commit` or `rollback`.

Administrative Matters

We don't have space to go into much detail here. This section only summarises the Adminstrator's Guide and parts of the Reference Manual, both of which you'll find by visiting the AllegroCache website and clicking on *Documentation*.

http://franz.com/products/allegrocache/

The first point to make is that the AllegroCache database is implemented using structures known as *b-trees*. Think of a b-tree as a large, efficient, sorted, and (in this case) file-based hash table whose keys and values are arrays of numbers[*]. In case you want to work with b-trees directly (not something you should ever need to do with an AllegroCache database), documentation for the API is accessible via the URL above.

The b-trees live in files of type `.bt` stored in the locations handed to `open-file-database` and `start-server`. Also in that location you'll find one or more numbered files with extension `.dat`, for instance `ac000000.dat`. These are the database's *log files* and it's possible to recreate the database from them, either in its most recent form or winding back to some previous state. The server writes to the end of one log file at a time; when that's full it moves on to the next one and doesn't modify the old file any more. If the space taken up by these files becomes excessive over time, you can use a compression utility to generate a set of smaller logs. If you want to backup your database, save the log files.

Another use for log files is rebuilding the database after an upgrade. This is straightforward and full instructions are given in the Adminstrator's Guide.

Finally, two points about efficiency. The first is that you can use the accessor `object-class-size` to tune the size of your local cache, on a per-class basis. The other is that if you plan to commit a large number of objects you should put AllegroCache into *bulk loading mode* by calling `(commit :bulk-load :start)` beforehand; when you're finished call `(commit :bulk-load :end)`.

Alternatives

Now that we've covered one approach to persistence in some detail, it's worth glancing at a couple more. These are both open-source, free to use, and not commercially supported.

[*] To be precise: simple vectors of `(unsigned-byte 8)`.

Arthur Lemmens's *Rucksack*, released under an MIT-style license, runs on a variety of Lisp implementations: Allegro, LispWorks, SBCL, Clozure CL. The documentation is less thorough than that of AllegroCache: external symbols have documentation strings and there's a short tutorial to get you started. General coding principles should be recognizable now that you've seen how to drive AllegroCache and although it has fewer high-level utilities you should be able to see from the following that it wouldn't be hard to concoct your own:

```
(defun retrieve-from-index (class slot value)

  ;; Macro with-rucksack guarantees to close connection on the way out

  (with-rucksack (*rucksack* *rs-directory*)

    ;; Macro with-transaction attempts a COMMIT on exit and guarantees
    ;; that if the COMMIT fails it'll perform a ROLLBACK instead.

    (with-transaction ()

      ;; Even without calling (documentation 'rucksack-map-slot 'function)
      ;; it should be clear what this call does.

      (rucksack-map-slot *rucksack* class slot
                         (lambda (instance)
                           (return-from retrieve-from-index
                             instance))
                         :equal value)))
  nil)
```

Exercise

What does the above call to `rucksack-map-slot` do? Implement `allegrocache-map-slot`.

A notable weakness with Rucksack is that it does not support an equivalent to AllegroCache's client/server mode.

You can access the Rucksack repository, documentation, a talk about how it's implemented, and a users' mailing list by visiting:

http://common-lisp.net/project/rucksack/

Also worth a visit is *Elephant* which runs on SBCL, Allegro, LispWorks, Clozure CL and CMUCL. Note that it's licensed under the more restrictive *LLGPL* (see Appendix B). The implementation isn't 100% Lisp: it uses a serializer written in C and you have a choice of connecting to one of *Berkeley DB*, *SQLite* and *PostgreSQL* for the backing store. Once more this is an API which should look familiar and this time there's extensive documentation to go with it. Elephant supports multiple connections and although its authors believe this will work over a network provided the underlying database supports it, they haven't heard of anyone trying that out yet.

The Elephant project homepage is at:

> *http://common-lisp.net/project/elephant/*

and you'll find a comparison between Rucksack and Elephant here:

> *http://bc.tech.coop/blog/060604.html*

We conclude our look into persistence for Lisp with a note about more traditional database interfaces. Some of these are driven by sending SQL control strings to a database connection object and receiving lists of strings in return:

```
(sql "select term from link where term like 'c%';" :db *db-connection*)
=>
(("cl:open") ("close-database") ("close-database") ("commit") ("commit")
 ...)
;; Second return value is a list of "headings"
("term")
```

Exercise

What does the function `sql` gain by returning a list of lists rather than a single flat list ("cl:open" "close-database" ...)?

Other interfaces also support an object-oriented approach. The invocations are reminiscent of SQL but the results, as for queries in the persistence libraries - are lists of CLOS objects.

```
(select 'link
        :where [like [slot-value 'link 'term] "c%"])
=>
((#<db-instance LINK 8067092>) (#<db-instance LINK 8069536>)
 (#<db-instance LINK 8069176>) ...)
```

For more details, see the *Allegro ODBC* chapter in the ACL documentation, *Common SQL* in either the LispWorks documentation or my own tutorial on the subject at:

> *http://www.lispworks.com/documentation/sql-tutorial*

and finally—open-source and similar to the LispWorks implementation—*CLSQL* at:

> *http://clsql.b9.com/*

Using any of these SQL interfaces has the advantage that you can share your data with other applications very easily. On the other hand you'll lose at least some of the power which tightly integrated persistence can offer you. As to which of these will give you the greater benefit: this is between you and your project requirements.

What next?

That's it for persistence. It's been useful as an example of a language extension which comes "inside the box" for just one implementation, although it's available for others —arguably in a weaker form—under open-source licenses. It's also raised questions which bring us into contact with two further, proprietary but universal libraries.

- How do we prevent two threads from using the same connection at the same time?
- We know now where persistent objects go and how to make that happen. Where do non-persistent objects end up and what makes them go there?

These questions will the answered in the next two chapters.

Exercise

What *does* happen to non-persistent objects?