AllegroCache Tutorial
v 1.1.10

Franz Inc

draft: 09/28/06 @ 03:19:26 PM

Introduction

AllegroCache is an object database built on top of the Common Lisp Object System. In this tutorial we will demonstrate how to use AllegroCache to build, retrieve and search for persistent objects.

Loading AllegroCache

You may have received AllegroCache as part of your ACL distribution or installed a newer version from our web site. In both cases, you will use **require** to load AllegroCache, however to load newer versions you will need to supply **require** with a second argument. For example, on Windows:

```
cl-user(1): (require :acache)
; Fast loading C:\Program Files\acl80\code\acache.001
;;; Installing acache patch, version 1
Warning: You have loaded AllegroCache version 1.0.0 and newer versions
         have already been downloaded.  (If you would like to disable
         this warning, then create a file named
         C:\Program Files\acl80\code\acache_inhibit_version_warning.txt.
         The contents of this file do not matter, so it can be of zero
         length.)  To use one of the newer versions of AllegroCache,
         change the (require :acache) to one of the following, or, if
         you used `load' the specific the appropriate version directly:

  (require :acache "acache-1.0.3.fasl")
  (require :acache "acache-1.1.11.fasl")
  (require :acache "acache-1.1.2.fasl")

AllegroCache version 1.0.0
;   Fast loading C:\Program Files\acl80\code\sax.001
;;; Installing sax patch, version 1
;   Fast loading C:\Program Files\acl80\code\regexp2.fasl
;     Fast loading C:\Program Files\acl80\code\yacc.fasl
t
cl-user(2):
```

In this case, version 1.0.0 is the default, and versions 1.0.3, 1.1.2 and 1.1.11 have already been installed and are ready to use. To load one of the specific versions listed above, exit Allegro CL and load it:

```
cl-user(1): (require :acache "acache-1.1.11.fasl")
; Fast loading C:\Program Files\acl80\code\acache-1.1.11.fasl
AllegroCache version 1.1.11
;   Fast loading C:\Program Files\acl80\code\sax.001
;;; Installing sax patch, version 1
t
cl-user(2):
```

Newer versions of AllegroCache are not automatically loaded in preference to older versions. Newer versions of AllegroCache may not be compatible, so the decision to upgrade to newer versions must be a conscious one.

Package

AllegroCache symbols are in the **db.allegrocache** package which has the abbreviation **db.ac**. In the examples below we'll assume that the current package *uses* the db.allegrocache package so we will not precede AllegroCache symbols with package qualifiers.

Sample Program

The sample program we'll use in this tutorial is one that creates a tree. A tree consists of a set of nodes. Each node has zero or more child nodes. One node is not the child of any other node and we call that the *root* node. Nodes with zero children are called *leaf* nodes.

We are interested in computing the maximum depth of each node. The maximum depth of a node is the maximum number of nodes from the given node along a path to a leaf node. The maximum depth of a leaf node is one.

Make AllegroCache symbols accessible

We'll be working in the **user** package but we want to be able to access AllegroCache symbols without package qualifiers. In order to do so we do

```
cl-user(5): (defpackage :user (:use :db.allegrocache))
#<The common-lisp-user package>
cl-user(6):
```

If you're using the IDE on Windows or Linux then the default package is *cg-user*, not *user*, so you'll want to do

```
cl-user(5): (defpackage :cg-user (:use :db.allegrocache))
#<The common-graphics-user package>
cl-user(6):
```

Defining a Persistent Class

In Common Lisp classes are defined with **defclass**. When defining a class you specify a *metaclass* which then determines how objects of the defined class will be built. If you don't specify a metaclass the metaclass **standard-class** is assumed. In order to define a class whose members will be persistent you specify the metaclass **persistent-class**.

When you define a persistent class you can also specify that a slot is *indexed*. An index is a table that maps values to objects. You use an index on slot X to answer the question

"which objects have this value in slot X".

Below is the **defclass** we'll use to define the objects we'll use to build our graph in persistent space. The parts of the **defclass** that are persistent-class specific are highlighted in bold.


```
(defclass node ()
  ((name :initarg :name :reader name :index :any-unique)
   (children :initarg :children :reader children)
   (max-depth :initform nil :accessor max-depth :index :any)
   )
  (:metaclass persistent-class))
```

Each node object has a unique name, a list of children (which are node objects) and a max-depth which is the number of nodes in the longest path from this node to a leaf. We'll want to locate an object given its name thus we've declared an index for the name slot.   We'll want to ask which nodes have a certain max-depth and thus we've added an index for the max-depth.   The max-depth value isn't necessarily unique thus when we declare the index we don't include 'unique' in the index type.

We'll also define a print method for the node class so we can see at a glance the name and max-depth of each node

```
(defmethod print-object ((node node) stream)
  (format stream "#<node ~s  max-depth: ~s>"
                  (name node) (max-depth node)))
```

Creating a Tree

We begin by creating a database.   In AllegroCache a database is a directory. The directory need not exist when the call to open-file-database is done; AllegroCache will create it.

```
cl-user(5): (open-file-database "testit"
               :if-does-not-exist :create
               :if-exists :supersede)
#<db.allegrocache::database @ #x20fd48a2>
```

The above call removes the database if it exists and then creates the database in the directory **testit** in the current directory. **open-file-database** returns a database object that can be passed to many of the functions in AllegroCache. open-file-database also sets the ***allegrocache*** variable to this database object.   Functions that take a database object as an argument use the value of ***allegrocache*** as the default value, thus we didn't bother to capture the value returned by **open-file-database**.

4

Next we'll create a persistent node object and use **retrieve-from-index** to recover it from the database using its name.

```
cl-user(6): (make-instance 'node :name "foo")
#<node "foo"  max-depth: nil>
cl-user(7): (retrieve-from-index 'node 'name "foo")
#<node "foo"  max-depth: nil>
```

AllegroCache is a transactional database which means that nothing is permanent until a commit is done.   A rollback undoes all changes to the database since the last commit.

Since we really don't want to keep the object we just created we'll **rollback** our changes (which consisted of creating a node named "foo") and we'll then use **retrieve-from-index** again to verify that the object is no longer present:

```
cl-user(8): (rollback)
6
cl-user(9): (retrieve-from-index 'node 'name "foo")
nil
cl-user(10):
```

In order to make an interesting tree we'll add a random number of children to each node. However in order to make this example reproducible we'll write our own random number generator which supplies the same values each time.

The **buildtree** function below first builds the child nodes and then builds the node itself. If the **max-children** argument is zero then this is to be a leaf node.

```
(defparameter *my-random-state* nil)

(defun my-random (max)
  (mod (or (pop *my-random-state*)
        (progn (setq *my-random-state* '(4 3 2 1 0 0 1 2 3 4))
           11))
      max))

(defun buildtree (name max-children)
  (let (children)
    (if* (> max-children 0)
       then (dotimes (i (1+ (my-random max-children)))
           (push (buildtree (format nil "~a-~a"
                          name i)
                     (- max-children (my-random 3)))
               children)))
    (make-instance 'node :name name
                       :children (nreverse children)))))
```

We initialize our personal random number generator and build a tree:

```
cl-user(11): (setq *my-random-state* nil)
nil
cl-user(12): (buildtree "root" 5)
#<node "root"  max-depth: nil>
cl-user(13):
```

The root node of the tree will be named "root" and its children named "root-0", "root-1" and so on.

At this point the tree is in the private memory of our Lisp process and not in the database, but we can access and modify it just as if it was in the persistent store.

One feature of persistent classes not found in normal clos classes is that you iterate over all instances of a persistent class using the **doclass** macro.   (*When you run this example your results may be in a different order than shown below).*

```
cl-user(14): (doclass (obj 'node) (print obj))

#<node "root"  max-depth: nil>
#<node "root-1"  max-depth: nil>
#<node "root-1-0"  max-depth: nil>
#<node "root-1-0-1"  max-depth: nil>
#<node "root-1-0-1-0"  max-depth: nil>
#<node "root-1-0-1-0-0"  max-depth: nil>
#<node "root-1-0-0"  max-depth: nil>
#<node "root-1-0-0-0"  max-depth: nil>
#<node "root-0"  max-depth: nil>
#<node "root-0-3"  max-depth: nil>
#<node "root-0-3-0"  max-depth: nil>
#<node "root-0-3-0-0"  max-depth: nil>
#<node "root-0-3-0-0-0"  max-depth: nil>
#<node "root-0-2"  max-depth: nil>
#<node "root-0-2-0"  max-depth: nil>
#<node "root-0-2-0-2"  max-depth: nil>
#<node "root-0-2-0-2-1"  max-depth: nil>
#<node "root-0-2-0-2-1-0"  max-depth: nil>
#<node "root-0-2-0-2-1-0-0"  max-depth: nil>
#<node "root-0-2-0-2-1-0-0-0"  max-depth: nil>
#<node "root-0-2-0-2-0"  max-depth: nil>
#<node "root-0-2-0-2-0-0"  max-depth: nil>
#<node "root-0-2-0-2-0-0-0"  max-depth: nil>
#<node "root-0-2-0-1"  max-depth: nil>
#<node "root-0-2-0-1-1"  max-depth: nil>
```

```
#<node "root-0-2-0-1-1-0"  max-depth: nil>
#<node "root-0-2-0-1-1-0-0"  max-depth: nil>
#<node "root-0-2-0-1-0"  max-depth: nil>
#<node "root-0-2-0-0"  max-depth: nil>
#<node "root-0-2-0-0-1"  max-depth: nil>
#<node "root-0-2-0-0-1-0"  max-depth: nil>
#<node "root-0-2-0-0-1-0-0"  max-depth: nil>
#<node "root-0-2-0-0-1-0-0-0"  max-depth: nil>
#<node "root-0-2-0-0-0"  max-depth: nil>
#<node "root-0-2-0-0-0-0"  max-depth: nil>
#<node "root-0-2-0-0-0-0-0"  max-depth: nil>
#<node "root-0-1"  max-depth: nil>
#<node "root-0-1-0"  max-depth: nil>
#<node "root-0-1-0-1"  max-depth: nil>
#<node "root-0-1-0-1-0"  max-depth: nil>
#<node "root-0-1-0-1-0-0"  max-depth: nil>
#<node "root-0-1-0-0"  max-depth: nil>
#<node "root-0-1-0-0-0"  max-depth: nil>
#<node "root-0-0"  max-depth: nil>
#<node "root-0-0-1"  max-depth: nil>
#<node "root-0-0-1-0"  max-depth: nil>
#<node "root-0-0-0"  max-depth: nil>
#<node "root-0-0-0-0"  max-depth: nil>
#<node "root-0-0-0-0-0"  max-depth: nil>
#<node "root-0-0-0-0-0-0"  max-depth: nil>
nil
cl-user(15):
```

Next we'll write a function to compute the max-depth value for each node.

```
(defun compute-max-depth (node)
  (if* (max-depth node)
     thenret
     else (let ((max 0))
          (dolist (child (children node))
            (setq max (max max (compute-max-depth child))))
          (setf (max-depth node) (1+ max)))))
```

You'll note that setting values in persistent objects is done in the same way as in normal clos objects: using **setf** of a slot-writer function.

We run this function beginning at the root node.   We use **retrieve-from-index** to find the node named "root".

```
cl-user(15): (compute-max-depth
                (retrieve-from-index 'node 'name "root"))
9
cl-user(16):
```

Now we can verify that all the nodes have been assigned a max-depth value:

```
cl-user(16): (doclass (obj 'node) (print obj))

#<node "root"  max-depth: 9>
#<node "root-1"  max-depth: 5>
#<node "root-1-0"  max-depth: 4>
#<node "root-1-0-1"  max-depth: 3>
#<node "root-1-0-1-0"  max-depth: 2>
#<node "root-1-0-1-0-0"  max-depth: 1>
#<node "root-1-0-0"  max-depth: 2>
#<node "root-1-0-0-0"  max-depth: 1>
#<node "root-0"  max-depth: 8>
#<node "root-0-3"  max-depth: 4>
#<node "root-0-3-0"  max-depth: 3>
#<node "root-0-3-0-0"  max-depth: 2>
#<node "root-0-3-0-0-0"  max-depth: 1>
#<node "root-0-2"  max-depth: 7>
#<node "root-0-2-0"  max-depth: 6>
#<node "root-0-2-0-2"  max-depth: 5>
#<node "root-0-2-0-2-1"  max-depth: 4>
#<node "root-0-2-0-2-1-0"  max-depth: 3>
#<node "root-0-2-0-2-1-0-0"  max-depth: 2>
#<node "root-0-2-0-2-1-0-0-0"  max-depth: 1>
#<node "root-0-2-0-2-0"  max-depth: 3>
#<node "root-0-2-0-2-0-0"  max-depth: 2>
#<node "root-0-2-0-2-0-0-0"  max-depth: 1>
#<node "root-0-2-0-1"  max-depth: 4>
#<node "root-0-2-0-1-1"  max-depth: 3>
#<node "root-0-2-0-1-1-0"  max-depth: 2>
#<node "root-0-2-0-1-1-0-0"  max-depth: 1>
#<node "root-0-2-0-1-0"  max-depth: 1>
#<node "root-0-2-0-0"  max-depth: 5>
#<node "root-0-2-0-0-1"  max-depth: 4>
#<node "root-0-2-0-0-1-0"  max-depth: 3>
#<node "root-0-2-0-0-1-0-0"  max-depth: 2>
#<node "root-0-2-0-0-1-0-0-0"  max-depth: 1>
#<node "root-0-2-0-0-0"  max-depth: 3>
#<node "root-0-2-0-0-0-0"  max-depth: 2>
#<node "root-0-2-0-0-0-0-0"  max-depth: 1>
#<node "root-0-1"  max-depth: 5>
#<node "root-0-1-0"  max-depth: 4>
#<node "root-0-1-0-1"  max-depth: 3>
#<node "root-0-1-0-1-0"  max-depth: 2>
#<node "root-0-1-0-1-0-0"  max-depth: 1>
#<node "root-0-1-0-0"  max-depth: 2>
#<node "root-0-1-0-0-0"  max-depth: 1>
#<node "root-0-0"  max-depth: 5>
```

```
#<node "root-0-0-1"  max-depth: 2>
#<node "root-0-0-1-0"  max-depth: 1>
#<node "root-0-0-0"  max-depth: 4>
#<node "root-0-0-0-0"  max-depth: 3>
#<node "root-0-0-0-0-0"  max-depth: 2>
#<node "root-0-0-0-0-0-0"  max-depth: 1>
nil
cl-user(17):
```

We can see that the max-depth of the root node is 9 but it's not so clear from the above printing of the nodes which nodes are on the maximum length path from the root to a leaf.   So we'll write a function to find and print all maximum paths:

```
(defun print-longest-paths ()
  ;; print the nodes in the longest path from root to leaf
  ;; there may be more than one solution, we print all
  ;; of them

  (let* ((root-node
            (retrieve-from-index 'node 'name "root")))
    (search-longest-paths nil root-node)))

(defun search-longest-paths (sofar node)
  (let ((depth (max-depth node)))
    (if* (children node)
        then
            (dolist (child (children node))
               (if* (eql (max-depth child) (1- depth))
                   then (search-longest-paths
                           (cons node sofar) child)))
        else ; hit a leaf, print solution
            (let ((ind 0))
               (dolist (node (reverse (cons node sofar)))
                  (dotimes (i ind) (write-char #\space))
                  (format t "~s~%" node)
                  (incf ind))))))
```

and we'll run that function

```
cl-user(18): (print-longest-paths)
#<node "root"  max-depth: 9>
 #<node "root-0"  max-depth: 8>
  #<node "root-0-2"  max-depth: 7>
   #<node "root-0-2-0"  max-depth: 6>
    #<node "root-0-2-0-0"  max-depth: 5>
     #<node "root-0-2-0-0-1"  max-depth: 4>
      #<node "root-0-2-0-0-1-0"  max-depth: 3>
       #<node "root-0-2-0-0-1-0-0"  max-depth: 2>
```

9

```
        #<node "root-0-2-0-0-1-0-0-0"  max-depth: 1>
#<node "root"  max-depth: 9>
 #<node "root-0"  max-depth: 8>
  #<node "root-0-2"  max-depth: 7>
   #<node "root-0-2-0"  max-depth: 6>
    #<node "root-0-2-0-2"  max-depth: 5>
     #<node "root-0-2-0-2-1"  max-depth: 4>
      #<node "root-0-2-0-2-1-0"  max-depth: 3>
       #<node "root-0-2-0-2-1-0-0"  max-depth: 2>
        #<node "root-0-2-0-2-1-0-0-0"  max-depth: 1>
nil
cl-user(19):
```

You can see that there are two paths of length 9.

To save this tree in the database we must commit the transaction.

```
cl-user(20) (commit)
t
cl-user(21):
```

and finally when we're done with the database we should close it:

```
cl-user(22): (close-database)
#<AllegroCache db "testit" -closed- @ #x2072a5e2>
cl-user(23):
```


Composite Indexes

Object databases such as AllegroCache derive their power from the fact that related objects often point to one another and thus you can navigate between such objects without doing queries over the whole database.

There are times however when you want to extract information based on characteristics of objects where there are no explicit relations. In this case we use indexes that map from values to the objects that hold those values in a given slot. An important feature of indexes are that integer and string values in an index are sorted.   This allows one to perform range queries using a cursor over the index.

For some queries one index is not enough and that is the subject of the following example. A *composite* index is an index over the values of more than one slot. AllegroCache does not create composite indexes automatically (yet).   That means that you must create them yourself and we'll show that this isn't very hard.

In our example we're keeping track of chess games. We have a set of players and a set of games between those players. In each game there is either a winner and a loser or the game was a draw.

We'll define a **player** class whose instances represent the players. Each player will have a unique **player number** and then information about the player.

We'll define a **game** class whose instances represent the games played. The information we'll save is a unique **game number**, the two players involved, and the result of the game.

To complete our schema design we have to consider the kinds of queries we're likely to make on the database. We'll want to know *how many* games a player won, lost and drew. We could just store that information in the player object so that that would be a trivial question to answer. We'll want to to know *which* games a player won, lost and drew. This is much harder since no fixed and reasonably sized container can hold all the games now and into the future. We could store a three set objects in each player object, one each for win, lose and draw. This would work at some expense in space (now we have four persistent objects for each player). Now what if we wanted a list of all the games where Player A defeated Player B, or all the games where Player A and Player B drew? A matrix seems like the obvious solution but it doesn't scale well. If we have 10,000 players in our database we need a 100,000,000 element matrix to represent the results of two players meeting.

One solution that can answer all the queries mentioned in the preceding paragraph is a composite index. We'll describe each game using a triple of these values of the game:

- the result
- the winner
- the loser

We'll denote this by a triple: (R,W,L). So if Player 10 defeats Player 20 we'll denote this way: (Victory,10,20). If players 39 and 22 draw we'll denote it (Draw,22,39), making an arbitrary rule that in the case of a draw the lower numbered player is listed first.

The next question is "how should the triple be represented". The "Lispy" way would be to use a list but in an index there is no ordering between values that are lists and we'll see that using ordering is important. A better way to represent this triple is to encode the triple as an integer. We'll use this formula to create the integer:

```
(+ (ash result 40) (ash winner 20) loser)
```

where result is 1 if the game is a draw and 0 if the game resulted in a victory. **winner** and **loser** are the player's unique numbers.

Here we allow 20 bits for the **winner** number and **loser** number. That means we can hold roughly a million players in our database. If you think that's too few you can replace the number 20 with a larger number.

Now that we've encoded the triple as an integer we can take advantage of the ordering of integers in an index. For example all of the games where player 67 wins will have index values greater than or equal to

```
(+ (ash 0 40)  (ash 67 20) 0)
```
and less than

```
(+ (ash 0 40)  (ash 68 20) 0)
```
We'll see that it's a simple matter in AllegroCache to enumerate all the objects with a range of integer values in an indexed slot.

What if you want find all the index values where player 67 lost. The first possible one is

```
(+ (ash 0 40) (ash 0 20) 67)
```
and the next possible on is

```
(+ (ash 0 40) (ash 1 20) 67)
```
These numbers aren't consecutive so the ordering won't help.

What you to solve this problem is to store a reverse encoded value in another slot and index on that.

Thus (R,W,L) is reverse encoded

```
(+ (ash result 40) (ash loser 20) winner)
```
so that we can now pick a loser number and find all the associated winners in the following index numbers.

Let's examine the code for this example. If you're running this under the IDE on Windows replace references to the **:user** package with the **:cg-user** package.

```
(eval-when (compile load eval) (require :acache))

(defpackage :user (:use :db.allegrocache))
(in-package :user)

(defclass game ()
  ((number :index :any-unique
        :initarg :number
        :reader game-number)

   ; :victory, :draw
   (result  :initarg :result
         :reader game-result)

   ; if :draw then winner slot is filled with player
   ; with the lower player-number
   (winner :initarg :winner
        :reader game-winner)
   (loser  :initarg  :loser
```

```
            :reader game-loser)
   (encoded :initarg :encoded
            :index :any
            :reader game-encoded)
   (encoded-rev :initarg :encoded-rev
            :index :any
            :reader game-encoded-rev))
  (:metaclass persistent-class))



(defclass player ()
  ((number :index :any-unique
         :initarg :number
         :reader player-number)

   (name  :initarg name
        :reader player-name))
  (:metaclass persistent-class))
```

In the game object we store the encoding we discussed above in the **encoding** slot and the reverse encoding in the **encoding-rev** slot.

In the real chess database both the player and game objects would contain a lot more information about the players and the game played.

Next we'll show the code to build up a test database on which to run our queries.   We create 50 players and pretend that 10,000 games were played between them.   That's a lot of chess.

```
(defparameter *number-of-players* 50)

(defun build-test-case ()
  (create-file-database "chess.db")

  ; create a set of players
  ; we should name them but that's not important
  ; for the demo
  (dotimes (i *number-of-players*)
    (make-instance 'player :number i))

  ; make up a set of games

  (dotimes (i 10000)
    (multiple-value-bind (result winner loser)
      (choose-random-result)

        ; in the case of a draw we put lower numbered player
```

```
       ; in the winner slot
       (if* (and (eq result :draw)
            (> winner loser))
       then (rotatef winner loser))

       (make-instance 'game
            :number i
            :result result
            :winner (retrieve-from-index 'player 'number winner)
            :loser  (retrieve-from-index 'player 'number loser)
            :encoded     (encode-game result winner loser)
            :encoded-rev (encode-game result loser  winner)
     )))
   (commit)
   (close-database))


(defun choose-random-result ()
  (let* ((result (case (random 2)
                   (0 :victory)
                   (1 :draw)))
         (winner (random *number-of-players*))
         (loser (loop
                  (let ((player (random *number-of-players*)))
                     (if* (not (eql player winner))
                        then (return player))))))
    (values result winner loser)))
```

This is the encoding function we mentioned earlier:

```
(defun encode-game (result playera-number playerb-number)
  (+ (ash (case result
            (:victory 0)
            (:draw 1))
          40)
     (ash playera-number 20)
     playerb-number))
```

With just this code loaded we can run (build-test-case) to create and populate our database:

```
cl-user(16): (build-test-case)
#<AllegroCache db "/home/jkf/chess/chess.db" -closed-
    @ #x723cffca>
cl-user(17):
```

Our first query will be to chose a possible result (e.g. player 23 defeated player 12) and then find in which games (if any) that result occurred.   One way to execute this query is to inspect all the game objects `(doclass (game 'game) ...)` and check for the desired result. This will take time proportional to the number of games played as well as causing memory bloat as the game objects are brought into memory (although they will be evicted from memory once the cache fills up).   A better way is to use the index on the **encoded** slot.   This index will bring us right to the games where a certain result occurred.

```
(defun test-query-a ()
  (open-file-database "chess.db")
  (dotimes (i 10)
    (multiple-value-bind (result winner loser)
        (choose-random-result)

      (if* (eq result :draw)
        then (format t "In which games did player ~s and ~s~
draw?~%"
                 winner loser)
           (if* (> winner loser)
             then ; put in canonical order
                  (rotatef winner loser))
        else (format t "In which games did player ~s defeat ~
player ~s~%"
                 winner loser))

      (let ((enc (encode-game result winner loser)))
        (let ((games (retrieve-from-index 'game 'encoded enc
                                          :all t)))
          (if* (null games)
            then (format t "  no games~%")
            else (dolist (game games)
                   (format t " game ~s~%" (game-number game))))))
      (terpri)))
  (close-database)
  )
```

The above function runs that type of query over 10 sample results.   Most of the function is code to setup the test. The actual query is shown above in bold. It's just a call to retrieve-from-index.   The output looks like

```
cl-user(17): (test-query-a)
In which games did player 13 and 24 draw?
 game 5437
 game 3089
```

In which games did player 17 defeat player 44
 game 1157
 game 513
 game 9720
 game 2040

In which games did player 18 and 46 draw?
 game 1340
 game 1692
 game 9468
 game 9672

In which games did player 43 and 35 draw?
 game 9641
 game 512
 game 4206
 game 616

In which games did player 9 defeat player 31
 game 5056

In which games did player 26 and 44 draw?
 game 8179
 game 8946
 game 3805
 game 9678

In which games did player 40 defeat player 21
 game 1312
 game 7245

In which games did player 0 and 47 draw?
 game 8630
 game 693
 game 654
 game 3981
 game 4106
 game 3658

In which games did player 13 defeat player 28
 game 4625

In which games did player 41 defeat player 31
 game 2106
 game 393
 game 7303
 game 1154
 game 4686

16

```
  game 3137

#<AllegroCache db "/home/jkf/chess/chess.db" -closed-
   @ #x729c9732>
cl-user(18):
```

Suppose that you didn't want to know the exact games that a certain result occurred but were just interested in the number of such games. Here's a test that shows how that query is done.

```
(defun test-query-b ()
  (open-file-database "chess.db")
  (dotimes (i 10)
    (multiple-value-bind (result winner loser)
        (choose-random-result)

      (if* (eq result :draw)
         then (format t "In how many games did player ~s and ~s
draw?~%"
                       winner loser)
              (if* (> winner loser)
                 then (rotatef winner loser))
         else (format t "In how many games did player ~s defeat
player ~s~%"
                       winner loser))

      (let ((enc (encode-game result winner loser)))
        (let ((count (length (retrieve-from-index 'game 'encoded
                                                  enc
                                                  :all t
                                                  :oid t))))
          (format t "  ~s game~p~%" count count)))

      (terpri)))
  (close-database))
```

What is different in this case is that we specify :oid t since we don't want AllegroCache to bother creating the game objects since we're just going to count them.

```
cl-user(18): (test-query-b)
In how many games did player 16 defeat player 2
  3 games

In how many games did player 3 defeat player 10
  0 games

In how many games did player 20 and 45 draw?
```

```
   5 games

In how many games did player 46 and 16 draw?
  5 games

In how many games did player 22 and 20 draw?
  5 games

In how many games did player 29 defeat player 27
  1 game

In how many games did player 1 defeat player 36
  0 games

In how many games did player 37 defeat player 41
  1 game

In how many games did player 20 defeat player 29
  2 games

In how many games did player 25 defeat player 43
  3 games

#<AllegroCache db "/home/jkf/chess/chess.db" -closed-
  @ #x72c763d2>
cl-user(19):
```

Now you want to find a list of the games a player won. We'll use the index on the encoded slot and we'll make use of cursors since we want to examine a segment of the index. We can create a cursor that scans over the precise part of the index range that denotes a game won by a player.

We test this on three random players:

```
(defun test-query-win ()
  (open-file-database "chess.db")
  (dotimes (i 3)
    (let ((player (random *number-of-players*)))

      (format  t "Player ~s won these games~%" player)
      (let ((cur (create-index-cursor
                            'game 'encoded
                            :initial-value (encode-game :victory
player  0)
                            :limit-value    (encode-game :victory
(1+ player)  0)))))
        (unwind-protect
```

18

```
          (do ((game (next-index-cursor cur) (next-index-cursor
cur)))
              ((null game))
            (format  t "  In game ~s against player ~s~%"
                     (game-number game)
                     (player-number (game-loser game))))
        (free-index-cursor cur))))
   (terpri))
  (close-database))
```

Running this function produces a large result so we'll omit most of the results below:

```
cl-user(19): (test-query-win)
Player 42 won these games
  In game 8024 against player 0
  In game 1412 against player 0
  In game 5678 against player 0
  In game 1985 against player 1
  In game 4164 against player 1
    . . .
  In game 5016 against player 47
  In game 6000 against player 49
  In game 6813 against player 49

Player 48 won these games
  In game 786 against player 0
  In game 7202 against player 0
  In game 7998 against player 0
  . . .
  In game 2087 against player 20
  In game 177 against player 20
  In game 444 against player 49

Player 22 won these games
  In game 379 against player 0
  In game 9002 against player 0
  . . .
  In game 5872 against player 49
  In game 1652 against player 49

#<AllegroCache db
"/home/jkf/acl8/src/cl/src/acache/doc/examples/chess/chess.db"
-closed-
  @ #x723dc57a>
cl-user(20):
```

Testing to see which games a player lost is the same as above but we check the **encoded-rev** slot's index.

19

```
(defun test-query-lose ()
  (open-file-database "chess.db")
  (dotimes (i 3)
    (let ((player (random *number-of-players*)))

      (format  t "Player ~s lost these games~%" player)
      (let ((cur (create-index-cursor
                  'game 'encoded-rev
                  :initial-value (encode-game :victory player 0)
                  :limit-value (encode-game :victory (1+ player)
                                                         0)))))
        (unwind-protect
            (do ((game (next-index-cursor cur)
                       (next-index-cursor cur)))
                ((null game))
              (format  t "  In game ~s against player ~s~%"
                       (game-number game)
                       (player-number (game-winner game))))
          (free-index-cursor cur))))
    (terpri))
  (close-database))
```

And finally check to see which games a player drew requires that you check the encoded
and the encoded-rev index. The encoded index will show draws with players of higher
numbers and the encoded-rev index will show draws with players of lower numbers

```
(defun test-query-draw ()
  (open-file-database "chess.db")
  (dotimes (i 3)
    (let ((player (random *number-of-players*)))

      (format  t "Player ~s drew these games~%" player)
      (let ((cur (create-index-cursor
                  'game 'encoded
                  :initial-value (encode-game :draw player 0)
                  :limit-value (encode-game :draw (1+ player)
0))))
        (unwind-protect
            (do ((game (next-index-cursor cur)
                       (next-index-cursor cur)))
                ((null game))
              (format  t "  In game ~s against player ~s~%"
                       (game-number game)
                       (player-number (game-loser game))))
          (free-index-cursor cur)))
      (format t " .. and ..~%")
```

20

```
      (let ((cur (create-index-cursor
                   'game 'encoded-rev
                   :initial-value (encode-game :draw player 0)
                   :limit-value (encode-game :draw (1+ player)
0)))))
        (unwind-protect
            (do ((game (next-index-cursor cur)
                       (next-index-cursor cur)))
                ((null game))
              (format  t "  In game ~s against player ~s~%"
                       (game-number game)
                       (player-number (game-winner game))))
          (free-index-cursor cur))))
    (terpri))
  (close-database))
```

Running this function also produces a large result so we'll omit most of the results below:

```
cl-user(45): (test-query-draw)
Player 48 drew these games
  In game 2204 against player 49
 .. and ..
  In game 3549 against player 0
  In game 2401 against player 0
  . . .
  In game 1028 against player 46
  In game 2582 against player 46

Player 4 drew these games
  In game 5200 against player 5
  . . .
  In game 5927 against player 28
  In game 8885 against player 28

#<AllegroCache db
"/home/jkf/acl8/src/cl/src/acache/doc/examples/chess/chess.db"
-closed- @
  #x71c9afd2>
cl-user(46):
```