# *AllegroCache*
## *Reference Manual*

Franz Inc.

version 3.0

# Table of Contents

# Introduction

AllegroCache is a persistent object store.  AllegroCache supports transactions and works either single-user with a file on the local machine or over the network in a client-server mode.

# Package

The package holding AllegroCache symbols is  "**db.allegrocache**" with the nickname "**db.ac**".    For those running lisp in ANSI mode the packages names are naturally in all upper case.

# Creating persistent objects

To create persistent objects first define a class with the metaclass  persistent-class:

```
(defclass foo ()
        ((next :initform nil)
         (val :initarg :val :accessor foo-val))
        (:metaclass persistent-class))
```

When make-instance is called a persistent object will be created.

```
  (make-instance 'foo :val 12345)
```

This make-instance call can only be done when a connection to a database is open (and the object representing that connection is stored in the global symbol **\*allegrocache\***. The  object just created will not be written to the database until the  **commit** function is called.

A persistent clos object operates  just like a normal clos object except that the slots of the object can only store lisp objects that can be stored in a database.  A list of what can be stored is given in Table 1.

| Data type | Notes |
|---|---|
| symbol | Symbols are stored along with their package. |
| number | Integers and floating point numbers. |
| string | Unicode characters are supported |
| character | Unicode characters are supported |
| cons | Proper and dotted lists of persistent values are also persistent |
| vector | Simple vectors of type t (containing objects of any type listed in this table) are persistent.   Simple vectors of (unsigned-byte 8) are also persistent. |
| object | A reference to a persistent clos object in the same database can be store.  A non-persistent clos object can be stored if an encode-object method is defined to tell AllegroCache how to store it. |
| map | A map object (either ac-map or ac-map-range) |
| set | A set object (of class ac-set) |
| structure | Objects defined by defstruct can be stored only if an encode-object method is defined to tell AllegroCache how to store the object. |

*Table 1 Persistent Values*


The normal clos operations are used for storing and retrieving persistent values.

storing persistent values:
```
(setf (slot-value x 'next) "none")
(setf (foo-val x) 1234)
```

retrieving persistent values:
```
(slot-value x 'next)
(foo-val  x)
```




    The metaclass **persistent-class** introduces a new slot allocation type **:persistent** and makes **:persistent** the default allocation type (instead of **:instance**).   If  you wish your persistent objects to have class or non-persistent instance allocated slots you can use the **:class** or **:instance** allocation types respectively:

```
(defclass foo ()
  ((next :initform nil)
   (val :initarg :val)
   (nonpersistent :allocation :instance)
   (classalloc    :allocation :class))
    (:metaclass persistent-class))
```

## Indexes

An index is a function associated with a slot S of a persistent class that if given a value X returns all persistent objects that have the value X in slot S.

You can specify that an index function can be created by the :index slot specifier:

```
(defclass foo ()
    ((val :initarg :val :index :any)
   (bar :initarg :bar :index :any-unique))
    (:metaclass persistent-class))
```

the value of the **:index** slot specifier can be one of:

| | |
|---|---|
| **:any** | this slot can take on any value and the same value can appear in this slot in more than one object of this class |
| **:any-unique** | this slot can take on any value and the same value will not appear in this slot in more than  one object of this class. |

If the **:index** slot specifier is added to the class after the class is used to create a database, then AllegroCache will create the index and populate it with information for existing instances in the database at the time of class redefinition.

Beginning in version 1.0.2 a violation of uniqueness causes an error to be signaled at commit time. Due to the overhead of  checking for uniqueness using an index type of  :any will give you  the fastest results.

## Maps

A map is a table that persistently stores key and value pairs.   A map may be named so that can be retrieved by name.

```
cl-user(3): (setq m (make-instance 'ac-map
                          :ac-map-name "mymap"))
#<ac-map oid: 13, ver 1, trans: nil,  modified @ #x71d7c952>
cl-user(4): (dotimes (i 10) (setf (map-value m i) (* i i)))
nil
cl-user(5): (map-value m 3)
9
t
cl-user(6):
```

## Sets

A set is a persistent collection of objects no two of which are equal (in the sense of the Lisp predicate **equal**).   You can add values to a set and remove them and you can iterate over the values.   If you're only storing a few objects in a set then you would be better off just using a Lisp list to store the values.  If however you may store many objects in the set then the special set data type is more efficient.

```
cl-user(4): (setq s (make-instance 'ac-set))
#<ac-set oid: 13, trans: nil,  modified @ #x71cd8422>
cl-user(5): (add-to-set s 10)
10
cl-user(6): (add-to-set s 'foo)
foo
cl-user(7): (doset (obj s) (print obj))

10
foo
nil
cl-user(8): (remove-from-set s 10)
10
cl-user(9): (doset (obj s) (print obj))

foo
nil
cl-user(10):
```

# Transactions

The AllegroCache transaction model is designed to be familiar to relational database users as well as permitting high performance object database operations.

AllegroCache can be run in *standalone* or *client-server* mode.   In standalone mode the transaction model is very simple to understand.  In client-server mode the model is harder to understand but this is to be expected as much more of the transaction features come into play when there are multiple clients operating on the same database.   It's important to understand though that the actual AllegroCache   transaction model is the same for standalone and client-server modes.

We'll first describe how transactions work in standalone mode.

## *Standalone Mode*

The key feature of transactions in standalone mode is *atomicity*.   This means that all changes made during a transaction are made to the database or none of them are.   If you call **commit** then all changes are stored in the database.  If you call **rollback** or just close the database with **close-database** then none of the changes in this transaction are made to the database.

The figure below shows what happens after two commits.  Initially the database is in state 1.   A commit brings it to state 2 and a second commit brings it to state 3.   The red arrow indicates changes made to the database but not yet committed.
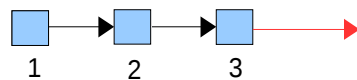


At this point the client program can **commit** and create a state 4 or **rollback** and remove all the changes indicated by the red arrow and return to an unmodified state 3 and then begin making a new set of changes.

When using standalone mode only one thread at a time can be accessing the database. Since simply CLOS operations like slot-value can invoke database functionality you have to be extremely careful if you have a multi-threaded program and you're using an AllegroCache database.   With SMP it's even more tempting to use multiple threads to solve problems.   You should seriously consider using client-server mode if you want your application to have multiple threads each communcating with an AllegroCache database.

## *Client-Server Mode*

A second key transaction feature that comes into play only in client-server mode is *isolation.* A client connecting to the database wants to see only its changes while it's operating on the database. At certain points when it's appropriate the client will want to save or flush its changes and then the client will want to see what other clients have been doing to the database.

Let's begin by assuming that one client has connected to the database server and has done one **commit**. The database is in state 2 and the client has made some local changes shown by the red arrow.
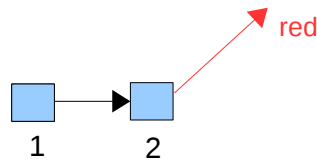
Now a second client connects to the database. It sees the database at the most recent committed state (state 2 in our example). This new client starts to make changes and we'll indicate those with a green arrow.
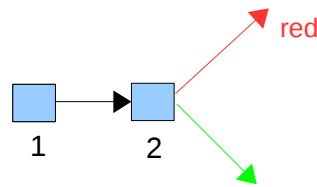
The changes that the Red client makes are not seen by the Green client and vice-versa.

Now suppose the Green client calls **commit** and that **commit** succeeds. The result is:
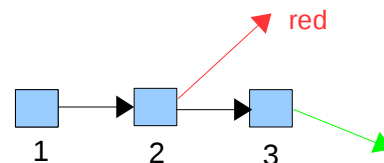
The database on disk has moved to state 3 and the Green client is now making changes based on state 3.
The Red client is still making changes based on state 2. It hasn't seen any of the changes made by the Green client. The Green client has not seen any of the changes made by the Red client either.

Let's now examine the two possible next steps for Red in our scenario.

Suppose the Red client decides that it doesn't want to keep the changes it has made  to the database and thus calls **rollback**.   All the changes Red has made to objects during the transaction are undone and Red's view of the database is advanced to the latest state of the database:



The reason that the Red advances to view the latest state of the database is that most of the time a rollback is done is because a commit failed and in order to fix a failing commit the best strategy is to advance to the most recent database state and redo the changes and try the the commit again.
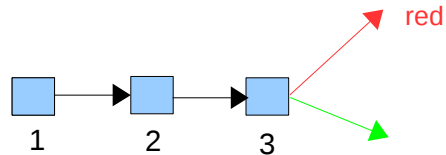[In the future we may allow a user to specify an option to rollback to prevent it from advancing to the latest state].

Suppose instead of a **rollback** Red had called **commit**.    AllegroCache would check to see if any of the objects Red had modified had been modified by some other client (e.g. Green) since state 2.  If so then the commit would fail.  If not then the Red changes are made to the database and the result is a new state 4 which only Red sees.   Green is still making changes to state 3.
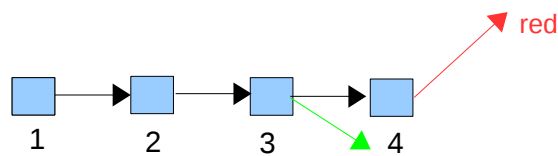


## *Rollback to go Forward*

In the client-server mode you may have a client connected to the database whose only job is to monitor the database for certain events being present and then act on those events. You've seen above that if a client connects to the database it will see the latest state of the database at the time the connection is made but it will never see the database change. This is not what the monitoring process wants.  It periodically wants to see the latest state of the database.   The only way to advance to the latest state is to abandon any changes it

may have made during this transaction and this is done using the **rollback** function. Thus the monitor program will be written this way:

```
(loop
  (rollback)
  (if* (work-for-us-to-do)
     then (do-work))
  (sleep 60) ; sleep for a minute
  )
```

If the **do-work** function referenced above makes changes to the database then it must call **commit** so that they aren't lost when **rollback** is called at the top of the loop.    When calling **commit** it's a good idea to wrap all database operations and the **commit** in a call to **with-transaction-restart.**  Then, if the **commit** fails a **rollback** will be done and the database changes will be done again followed by a **commit**.

```
(defun do-work ()
  (with-transaction-restart (:message "doing work")
    (do-the-actual-work)
    (commit)))
```

# Database

 Logically a database is a collection of persistent class definitions, instances of those classes, index tables for some of the slots of the persistent classes and finally named maps.

Physically a database is a directory containing a set of files, each file holding a single B-Tree.   A B-Tree is a data structure that maps keys to values and sorts the key-value pairs by the key.

# Object Identifiers

 In a database every object has a unique object identifier (oid).   This value can be

retrieved using **db-object-oid**. An oid is an integer.  There is no way to determine the class of an object given its oid.

 Usually a program need not concern itself with oids.  However in certain circumstances it may be convenient to work with oids.  One such case is when combining the results of multiple indexes over the same class.  You may want to ask for the set of objects whose X slot is greater than 10 and whose Y slot is greater than 20.  It consumes fewer resources to ask for the oids of objects whose X slot is greater than 10 than to ask for the objects themselves.  In the later case the objects retrieved have to be instantiated in the object cache and there's no point in doing that if you don't need all those objects.  In this case you don't need all those objects since you only need those objects  whose Y slot is also greater then 20.  Thus the optimal way to do this query is to find the intersection of the oids corresponding to "X > 10" and those oids with "Y > 20" and then from that intersection find the objects corresponding to the oids.

# Configuration: Standalone

 In the standalone configuration of AllegroCache, Lisp operates directly on the B-Tree files which should be stored on a directory on a local filesystem.   This gives you the maximum performance.   In standalone mode a commit can never fail.

# Configuration: client/server

 In this configuration you first start an AllegroCache server on one machine and have it open a B-Tree database on a local disk.   Next you can have any number of AllegroCache client programs connect to this server process.  Each client has an independent object cache.  The server also has a cache of the blocks in the B-Trees.

# The API

## *Standalone Configuration*

```
open-file-database (local-directory
                       &key if-exists
```

```
if-does-not-exist
(use :db)
(verify t)
class-cache-size
object-cache-size
log-size
read-only
max-fds
max-logs-open)
```

This function creates or opens an existing database on a local disk.   It will open
databases on remote mounted disks but you may run into performance problems when
using disks on another machine.

| | |
|---|---|
| local-directory | a string or pathname naming a directory (which may or may not yet exist) |
| if-exists | Specifies what to do if the database already exists:<br><br>• :open (the default value) - open the existing database<br>•  :supersede - destroy the existing database and create a new one<br>• :error - signal an error |
| if-does-not-exist | Specifies what to do if the database does not exist:<br><br>• :error (the default value) - signal an error<br>• :create  - create a new database |
| use | Specifies which class definition should take precedent if a class is defined in both the database and Lisp memory<br><br>• :db – the definition in the database will overwrite the definition in memory.  This is the default<br><br>• :memory – the definition in memory will be used |
| verify | • Specifies how extensively AllegroCache should verify the database on startup.   Valid arguments are<br><br>• nil – don't verify at all<br><br>• :quick – verify that the last few log entries are valid, and if they aren't do a full scan of the whole last log file<br><br>• t – do a full scan of the last log file truncating it at the point the log file is found to be corrupt.   This is the default value used a :verify argument is not given. |

| | |
|---|---|
| | • :full – do a full scan of the last log file and also scan all btrees.  If the btrees are invalid signal an error. |
| class-cache-size | The number of bytes in the page cache for each B-Tree used to store the data.   If you're building a huge database and your machine has a lot of physical memory then specifying a large cache will improve performance.  The default is to not put a limit on the size of the class cache. |
| object-cache-size | Specifies the maximum number of objects of each class to store in the local object cache.  There is one object cache per persistent class.   When the number of objects in the cache exceeds this size some objects are evicted but this eviction will only occur if there are no references to this object from the heap.  The default value for this parameter is 70,000. |
| log-size | the size to which a transaction log file can grow before AllegroCache will close it and create a new log file.   The default value is 200 MB. |
| read-only | The files in the database will be opened for reading only.  This will allow you to open a database for which you don't have write permission.  Commits are not permitted and will signal an error. |
| max-fds | The maximum number of files descriptors that AllegroCache is allowed to use for its index files.   If you are creating many indexes or ac-map objects then you may run out of file descriptors provided by the operating system (especially on Mac OS X which provides very few).   Specifying a value here will cause AllegroCache to avoid keeping too many index files open at the same time. |
| max-logs-open | The maximum number of log files that can be open at a time.  After this number is reached AllegroCache will close log files that weren't used recently and will open them again upon demand.   If this number is much smaller than the actual number of log files in the database and AllegroCache is forced to close and re-open the log files then there will be a performance degredation |

**open**-**file**-**database** returns a database object and sets **\*allegrocache\*** to that database object thus making this database the default database for many of the AllegroCache functions.

```
create-file-database (local-directory
                       &key (verify t)
                            class-cache-size
                            object-cache-size
                            log-size
                            max-fds)
```

This function calls **open-file-database** with the arguments:
  :if-exists :supersede  :if-does-not-exist :create
thus always causing a new database to be created.


## *Client Server Configuration*


```
start-server (local-directory port
                &key if-exists
                     if-does-not-exist
                     (verify t)
                     class-cache-size
                     authenticate
                     rsa-keys
                     log-size
                     address
                     max-fds
                     max-logs-open
                     read-only
                     )
```

**start-server** creates a network server for the given database on the given port.   Some arguments to start-server are the same as open-file-database.    Thus you can open an existing database or create a new one.

If the **port** argument is **nil** then the operating system will chose a free port.  You can determine which port it chose by calling **netdb-port** on the return value from **start-server**.

We may support unix domain sockets in the future.

If the **authenticate** argument is non-nil then clients connecting to the server (using open-network-database) must authenticate themselves before they can do database operations. The value of **authenticate** is either a list of names and passwords, such as

```
(("joe" "mypass")  ("john"  "beem") ("sally" "wolf"))
```

or is a function of one argument which will do the authentication.  The function will be passed a list whose first element is the given user name and whose second element is the password and the function should return non-nil if the given user name and password are correct.

If **authenticate** is given then AllegroCache will compute a pair of RSA keys to be used for secure transmission of the user name and password by the client.  This generation process can take 30 seconds on a fast computer.   This key generation will only happen once (not for every client connection).

To bypass the generation of the RSA keys you can pass the keys in yourself using the **rsa-keys** argument.  The value of the **rsa-keys** argument should be a list of two elements, a public key and a private key, such as returned by the **generate-rsa-keys** function in Allegro Common Lisp.

The **log-size** argument specifies the size to which the  transaction log can grow before the log is closed and a new log started.   The default is 20MB.

The **address** argument specifies the desired network interface on which to listen.  If not specified, then the server will listen on all interfaces.  A typical non-nil value for this keyword is "127.1" to limit connections from the local machine, or loopback interface.

The **max-fds** and **max-logs-open** arguments are described in the documentation for **open-file-database**.

If the **read-only** argument is true then the database is opened in read only mode and no commits allowed.   When a client connects to the database it will then be told that the database is read only and commits won't be allowed.   One important reason to open a server database read-only is to cause compressed indexes to be used if they are present.

**start-server** returns a netdb object which can be passed to **stop-server** in order to shut down this server

```
open-network-database (hostname port
                       &key (use :db)
                            user
                            password
                            object-cache-size)
```

opens a connection to a database server on the network on the host specified by **hostname** and given **port**.  The server must already be running.

The **use** argument specifies what to do if a class is defined in both the database and in Lisp memory.   By default the definition in the database takes precedence.  If **:memory** is specified then the definition in memory is used.

If the server was started with authentication enabled then you must pass values for **user** and **password** as well and these will be used in the authentication protocol.   Failure to authenticate will result in an error being signaled.

**object-cache-size** is the maximum number of objects that should be stored in the object cache (although the cache will always expand if has to to hold all modified and not committed objects).   The default value is 70,000.

**open-network-database** returns a database object and sets **\*allegrocache\*** to that database object, thus making it the default database for many of the database functions.

```
stop-server (netdb)
```

shuts down the server and closes the associated database.   The netdb object is what was returned by **start-server**.

```
netdb-port  (netdb)
```

returns the port number on which this network database server is listening for connections.   The netdb object is what was returned by **start-server**.

There are other functions for the client server configuration documented in the Miscellaneous section below.   The functions include  **client-connections**, **kill-client-connection** and **connection-alive-p**.

## *Both Configurations*

```
close-database (&key (db *allegrocache*) stop-server)
```

close the database connection given.  Note that a commit will **not** be done automatically before the close.  If you want your changes committed you must call commit before close-database.

If **stop-server** is non-nil then if  this is a client database connection then the server will be shut down after the database connection is closed.   The shutdown will be delayed until there are no commits in progress from other clients.

```
database-open-p (db)
```

returns true if the given database object has not been closed.

## *Summary of Operations*

objects
      make-instance
      slot-value
      (setf slot-value)
      database-of
      delete-instance
      deleted-instance-p
      db-object-oid
      oid-to-object
      oid-to-object*
      mark-instance-modified

classes
      doclass
      doclass*
      delete-persistent-class
      create-class-cursor
      next-class-cursor
      free-class-cursor

maps
      map-value
      (setf map-value)
      remove-from-map
      map-map
      create-map-cursor
      next-map-cursor
      previous-map-cursor
      free-map-cursor
      retrieve-from-map-range

sets
      add-to-set
      remove-from-set
      set-member
      set-count
      doset

indexes
      retrieve-from-index

        retrieve-from-index*
        retrieve-from-index-range
        index-count
        create-index-cursor
        create-expression-cursor
        next-index-cursor
        previous-index-cursor
        free-index-cursor


## *Object Deletion*


```
delete-instance (object)
```

deletes the object from the database.    Attempts to read or write the object will signal an error after the object is deleted.    If **rollback** is called before **commit** the object will no longer be deleted.

Even after an object is deleted it remains in Lisp's memory until there are no more references to it.   The object is marked as deleted and you can use **deleted-instance-p** to test if a given object has been deleted.   It's the program's responsibility to ensure that objects it encounters when  traversing persistent values are not deleted before accessing them.

Once the deleted object is garbage collected out of Lisp's memory any future persistent values that are read from the database that contain a reference to the deleted object will have that reference replaced by nil.


```
deleted-instance-p (object)
```
returns true if the given object is a deleted persistent instance.

```
;; create an instance, set a slot and verify that it was set.
cl-user(20): (setq x (make-instance 'tfoo))
#<tfoo oid: 11, trans: nil,  modified @ #x71fdb23a>
cl-user(21): (setf (slot-value x 'a) 3330)
3330
cl-user(22): (slot-value x 'a)
3330

;; commit that object
cl-user(23): (commit)
t

;; delete the object
cl-user(24): (delete-instance x)
t

;; with the object deleted the slot is no longer accessible
cl-user(25): (slot-value x 'a)
```

```
Error: attempt to access a deleted object:
       #<tfoo oid: 11, trans: 6,  deleted @ #x71fdb23a>


Restart actions (select using :continue):
 0: Return to Top Level (an "abort" restart).
 1: Abort entirely from this (lisp) process.
[1] cl-user(26): :reset

;; rollback the state to the last commit
cl-user(27): (rollback)
6

;; now the object is no longer deleted and the slot is visible
cl-user(28): (slot-value x 'a)
3330
cl-user(29):
```

## *Iteration*

```
(doclass (var class-expr &key (db *allegrocache*))
          &body body)
```
will evaluate class-expr to get a class object or class name and will then evaluate body with var bound to successive objects from the class.   The iteration is done in a **(block nil ...)** context so that with the body a **(return)** will leave the iteration and  return a value from doclass.

```
(doclass* (var class-expr &key (db *allegrocache*))
           &body body)
```
is just like doclass except that objects from the named class *and* all subclasses of that class are bound to var during the iteration.

## *Class Cursors*

A class cursor allow a program to iterate through the instances of a persistent class.   This is similar to **doclass**.

```
(create-class-cursor class)
```
returns a class cursor for the given class or **nil** if AllegroCache can determine immediately that there are no instances of the class.   **class** can be a symbol or a class object.   The program should eventually call **free-class-cursor** on all class cursors created.

```
(next-class-cursor cursor &key oid)
```
returns the next object of the given class.   If **oid** is true then it returns the object id rather than the object.  **next-class-cursor** returns **nil** if there are no more objects to return.

You should not call **commit** or **rollback** while using a class cursor since it may prevent thecursor from iterating over all objects of the class.

```
(free-class-cursor cursor)
```
frees the resources used by the cursor.   This should be called by the program when it is finished using the cursor.   After this is called no further operations should be done on this cursor.

## *Maps*

A map is a table that maps a key to a value.   It's like a persistent hash table.   As currently implemented each map is stored as a B-tree on disk.   This means that it occupies space on the disk and it requires operating system resources to maintain a

connection to the b-tree.  Therefore use maps sparingly (create no more than a few hundred of them).

In versions of AllegroCache before 0.9.0 the open-map call was used to create new maps and gain access to existing maps.   In 0.9.0 maps were changed to be just normal persistent objects of type **ac-map**.     Beginning in version 1.2.4 we introduced a new type of map **ac-map-range** and we suggest that users no longer use ac-map.   We do maintain the class ac-map for upward compatibility.

The difference between ac-map and ac-map-range is the way the map keys are encoded. In an ac-map-range the keys are encoded as they are in indexes so that keys are stored in numerical or lexicographic order (for numbers and strings).   This then allow access to the subsequences of the map using cursors.

The definition of the the map classes are is in part

```
(defclass ac-map ()
  ((ac-map-name
       :initform nil
          :accessor ac-map-name
          :initarg :ac-map-name
          :index :any
          )
    ....)
   (:metaclass persistent-class))

(defclass ac-map-range (ac-map)
  ()
  (:metaclass persistent-class))
```

From the definition you can see that you can name a map when you create it by using the :ac-map-name initarg and you can find a map by name by using retrieve-from-index

```
(retrieve-from-index 'ac-map-range 'ac-map-name  "mymap")
```

Unlike earlier versions of AllegroCache  more than one map can have the same name.

The **open-map** function is still present but its use is deprecated as you can now create maps with make-instance.  **open-map**  will only create a map of class **ac-map** and this is another reason to not use this function.

```
open-map (name &key (db *allegrocache*)
```

```
(if-does-not-exist :create)
(if-exists :open))
```
create or access the map named **name** in the database **db**

| if-exists | what to do if the map already exists<br>• :open - return a connection to an existing map<br>• :error - signal an error |
|---|---|
| if-does-not-exist | what to do if the map does not exist yet<br>• :create - create a new map<br>• :error - signal an error |

**open-map** returns an object which can be passed to **map-value** to access the map.

Changes to the map are only permanent if you do a **commit**..  On a **rollback** all changes to a map since the last **commit** are lost.

```
map-value (map key)
```

returns two values:  the value associated with the key and **t** to indicate that the value was found.   map-value returns  just nil if there is no value associated with **key**.

```
(setf (map-value map key) value)
```

stores the **value** in the **map** under the given **key.**   After this call is made neither the **key** object nor the **value** object should be destructively modified until after a **commit** or **rollback**.

```
remove-from-map (map key)
```

remove the given key from the map.   This is like the remhash function used with hash tables.

```
map-map (function map)
```

The given **function** of two arguments is applied to each key and value pair  found in the **map**.   A map consists of committed and uncommitted data.   In a map of type ac-map-

range the committed data will be passed the the function in ascending order of keys.  The uncommitted data will be presented in any order.

```
;; create an ac-map-range
cl-user(3): (setq m (make-instance 'ac-map-range
                          :ac-map-name "mymaprange"))
#<ac-map-range oid: 13, ver 5, trans: nil,  modified @ #x724606da>

;; store three values in the map
cl-user(6): (setf (map-value m 10) 300)
300
cl-user(7): (setf (map-value m 20) "twenty")
"twenty"
cl-user(8): (setf (map-value m 30) '(a b c d e))
(a b c d e)

;; show the values in the map.   The data is uncommitted so
;; it will be presented in an unspecified order
cl-user(9): (map-map #'(lambda (k v) (print (list k v))) m)

(30 (a b c d e))
(10 300)
(20 "twenty")
nil
cl-user(10): (commit)
t

;; after comitting we see that the data is now in sorted order
cl-user(11): (map-map #'(lambda (k v) (print (list k v))) m)

(10 300)
(20 "twenty")
(30 (a b c d e))
nil
cl-user(12):
```

```
retrieve-from-map-range  ((map ac-map-range)
                           initial-value end-value
                           &key (key t) (value t))
```
returns a list of keys and value from the map.  The keys are greater than equal to **initial-value** and less than **end-value**.  The returned lists consists of  conses, where the car of the cons is the key and the cdr is the value.   The conses are ordered by key value from smallest to largest (according to the ordering specified for index values) .   You can chose to not compute and return the key and value values by specifying **nil**  for the **:key** and **:value** keyword arguments.  If both **:key** and **:value** are given nil values then the return values is a list of **nil**'s, one for each key/value in the range in the table.

**This function only returns committed values.**

```
cl-user(5): (setq mm (make-instance 'ac-map-range))
```

```
#<ac-map-range oid: 13, ver 5, trans: nil,  modified @ #x10018cf782>
cl-user(6): (dotimes (i 20) (setf (map-value mm i) (* i i)))
nil
cl-user(7): (commit)
t
cl-user(8): (retrieve-from-map-range mm 5 10)
((5 . 25) (6 . 36) (7 . 49) (8 . 64) (9 . 81))
cl-user(9): (retrieve-from-map-range mm 5 10 :value nil)
((5) (6) (7) (8) (9))
cl-user(10): (retrieve-from-map-range mm 5 10 :key nil :value nil)
(nil nil nil nil nil)
cl-user(11):
```

```
map-count ((map ac-map-range)
           &key initial-value end-value transient
                (committed t) max)
```

returns the number of values in the map.    Specifying **initial-value**  and/or **end-value**
will limit the count to those key values greater than or equal to **initial-value** and less than
**end-value**.   If **max** is given then the counting will cease when it reaches the value **max**
and **max**  will be returned.

This function only works for committed values.   Thus **transient** must have a value nil.
If **committed**  has a value nil then zero will always be returned.

```
cl-user(5): (setq mm (make-instance 'ac-map-range))
#<ac-map-range oid: 13, ver 5, trans: nil,  modified @ #x10018cf782>
cl-user(6): (dotimes (i 20) (setf (map-value mm i) (* i i)))
nil
cl-user(7): (commit)
t
cl-user(11): (map-count mm)
20
cl-user(12): (map-count mm :initial-value 10)
10
cl-user(13): (map-count mm :initial-value 10 :end-value 15)
5
cl-user(14): (map-count mm :initial-value 10 :end-value 15 :max 3)
3
cl-user(15):
```

## *Map Cursors*

A map cursor is an object that points into a map.   You can move the cursor forward and
backwards and retrieve the key and value to which it points.

*The map cursor only maps over committed objects in the map.*

Map cursors can only point into ac-map-range maps.

```
create-map-cursor ((map ac-map-range)
                    &key initial-value limit-value position)
```
returns a map-cursor object that will scan over the map **map**. The map cursor is positioned at the beginning of the map unless **initial-value** or **position** is specified.

If **initial-value** is given the cursor it set to point to the key whose value is **initial-value**. If such a key doesn't exist then the cursor is set to point to the value just after where the key initial-value be place if it had been in the map.

If i**nitial-value** is nil and **position** is :last then the cursor is placed on the last value in the map.

If **limit-value** is given then it specifies that the cursor will stop scanning the index forward when it reaches the point at which **limit-value** is or would be found as a key in the map. The **limit-value** is only used by **next-map-cursor** and not by **previous-map-cursor**.
   If there are no committed objects in the map, **create-map-cursor** may return nil.

```
next-map-cursor ((cursor map-cursor) &key (key t) (value t))
```
advances the cursor to the next key and value and returns three values: key, value and t.

The keyword arguments **key** and **value** can be specified as nil to indicate that the caller doesn't care about the key or value being returned. This can save time in decoding the values from the map.

The first time **next-map-cursor** is called after the cursor is created, **next-map-cursor** will *not* advance the cursor before returning the key and value.

**next-map-cursor** will return just nil when there are no more values to return. At that point the cursor object is disconnected from the associated map and cannot be used again.

```
previous-map-cursor ((cursor map-cursor)
                     &key (key t) (value t))
```
backs up the cursor to the previous key and value and returns three values: key, value and t.

The keyword arguments **key** and **value** can be specified as nil to indicate that the caller doesn't care about the key or value being returned. This can save time in decoding the values from the map.

The first time **previous-map-cursor** is called after the cursor is created, **previous-map-cursor** will *not* back up the cursor before returning the key and value.

**previous**-**map**-**cursor** will return nil when there are no more values to return.   At that point the cursor object is disconnected from the associated map and cannot be used again.


```
free-map-cursor ((cursor map-cursor))
```
disconnects the map cursor from the associated map.   After this call is made no further cursor operations can be done.   It's important to do this disassociation for every cursor you create to allow the backend database to free the cursor it uses to match the map-cursor.   You needn't call this function if a call to **next**-**map**-**cursor** or **previous**-**map**-**cursor** returned nil, as when this occurs the disassociation has been done.

## Sets

A set is constructed just like any other persistent object using **make-instance**. The name of the set class is **ac-set**.

```
(make-instance 'ac-set)
```

You can add objects to a set and remove objects from a set. The only objects that can be stored in a set are those objects that can be stored in the database (see the previous table that lists such objects).

```
add-to-set ((set ac-set) value)
```
add **value** to the **set**. If value is already in the set then nothing is done.

```
remove-from-set ((set ac-set) value)
```
remove **value** from **set**.

```
(doset (var set) &rest body)
```
A macro which iterates over all the elements in the **set** binding **var** to each element and evaluating the forms in the **body**.

```
set-member (value (set ac-set))
```
return true if **value** is a member of **set**. The membership test is **equal**.

```
set-count ((set ac-set) &key transient (commited t) max)
```
Return the number of elements in the set. If **transient** is true then the count will reflect items added and removed during the current transaction. If **committed** is true then the count will reflect data previously committed. The default value for **transient** is false and **committed** is true.
If **max** is given then it should be an integer and it is the maximum value that set-count will return. By specifying a value for **max** you allow set-count to return without having to read all of the set data.
The set-count function can run much faster (especially in client/server mode) if it doesn't have to consider transient data. Thus if possible don't pass a true value for **transient**.

**set-count** used to take a **db** argument but now the database to use is found by **(database-of set)**.

## Indexes

Index values are stored in the following order:

1. all other lisp values
2. negative infinity
3. integers and floating point numbers, ordered from least to greatest value
4. positive infinity
5. floating point NaN
6. strings sorted lexicographically by the unicode value of their characters

You can use this ordering to retrieve values within numeric or string ranges.   You'll find the **retrieve-from-index-range** function described in this section and then a whole section on Cursors follows this section on Indexes.

```
retrieve-from-index ((class persistent-class)
                      slot-name value
                      &key (db *allegrocache*)
                           all
                           oid)
```

retrieves the object or objects of class **class** that have **value** in the slot **slot-name.**
In order to use this function **slot-name** must have been declared to have an index when the class **class**  was defined.

| all | return all objects or oids instead of the first one.  It is intended that **nil** will be used when you know there are zero or one values.  In other words, for **:any-unique** index. |
| --- | --- |
| oid | if true return the object id instead of the object. |

```
retrieve-from-index ((class-name symbol)
                      slot-name value
                      &key (db *allegrocache*)
                           all
                           oid)
```

This method is identical to the previous method except that it takes a class name symbol as the first argument.   The **class-name** must name a persistent-class.

```
retrieve-from-index* ((class persistent-class)
                       slot-name value
                       &key (db *allegrocache*)
```

```
                          all
                          oid)
```

This method is just like **retrieve-from-index** except the given **class** and all subclasses are examined to find objects to return.  If **all** is **nil** then the first object found is returned.  The order that the given **class** and its subclasses are searched is unspecified.

```
retrieve-from-index* ((class-name symbol)
                       slot-name value
                       &key (db *allegrocache*)
                             all
                             oid)
```

This method is just like the above method except that a class name is specified.   The **class-name** must name a persistent-class.

```
oid-to-object ((class persistent-class) oid
               &key (db *allegrocache*))
```

return the object of the given class with the given oid.   You must specify the exact class of the object with the given oid or nil will be returned.   See oid-to-object* below for a related function.

```
oid-to-object  ((class-name symbol) oid
                 &key (db *allegrocache*))
```

this method is identical to the previous method except a class name is specified instead of a class object.

```
oid-to-object*  ((class class) oid &key (db *allegrocache*))
```
This returns the object with the given oid, checking all classes that a subclass of the given class.   The given class can be any class, persistent or not.

```
oid-to-object* ((class-name symbol) oid
                 &key (db *allegrocache*))
```
This method is identical to the previous method except that the class-name is specified instead of the class.   Given an oid whose class is unknown you can find the object with

```
(oid-to-object*  t oid)
```

since the class t is the superclass of all classes.

```
db-object-oid  (persistent-object)
```

Return the object id (oid) of the given persistent object.

```
retrieve-from-index-range (class slot-name
                                initial-value end-value
                           &key (db *allegrocache*) oid)
```

returns all objects of the given class (a persistent-class object or a symbol) whose slot **slot-name** has a value in the range beginning with **initial-value** up to but not including **end-value**.   If **oid** is true then the object id values are returned instead of the objects.

If **initial-value** is not specified then objects returned begin with the first object in the index.  If **end-value** is not specified then all objects from the **initial-value** up to the end of the index are returned.

Sorting is only done on integers, floating point numbers and strings.  Passing any other type of value as **initial-value** or **end-value** will signal an error (although passing **nil** is the same as not specifying a value).

**Very important note**: this function only returns values that have been committed to the database because only the database on disk keeps the index values sorted.   Thus before calling this function on newly created data you should commit the data.

For example, suppose the Employee object is defined as
```
(defclass Employee
   ((first-name :index :any)
    (last-name  :index :any)
    (salary))
  (:metaclass persistent-class))
```

We could find all the employees whose name begins with F by evaluating
```
(retrieve-from-index-range 'Employee 'last-name "F" "G")
```

We could find the oids of all the employees whose first name begins with "Jo" and whose last name begins with "F" using

```
(intersection (retrieve-from-index-range
                     'Employee 'first-name "Jo" "Jp" :oid t)
```

```
            (retrieve-from-index-range
                   `Employee `last-name "F" "G"   :oid t))
```

```
index-count (class slot-name &key (db *allegrocache*)
                                        initial-value
                                        end-value
                                        transient
                                        (committed t)
                                        max)
```

returns the number of values that would be returned by retrieve-from-index given the same values for **class**, **slot-name**, **db**, i**nitial-value** and **end-value**.   If **max** is specified then index-count will return a value no larger than max.   Specifying **max** will allow index-count to return without reading all values in the index.    At this time index-count can only count values committed to the database.   As a result the **transient** argument must remain **nil** and the **committed** argument must remain **t** (these are the default values for these arguments).

## Index Cursors

A cursor in a pointer into an index.   You can use a cursor to retrieve objects one at a time rather than all at once in a list.

A cursor can only reference an index in the database on disk, it cannot reference uncommitted index values (this is not the same as the **retrieve-from-index** function which looks both in the database and in uncommitted data).

A cursor is created with **create-index-cursor** and then passed to **next-index-cursor** until that function returns nil.   At that point the cursor is exhausted and you can just drop the reference to the cursor and it will be garbage collected.

```
create-index-cursor (class slot-name
                            &key (db *allegrocache*)
                                 initial-value
                                 limit-value
                                 position
                                 unique
                                 preread)
```

creates and returns an index cursor for slot **slot-name** of class **class** (which can be a symbol or a class object).    The **initial-value** argument, if given, specifies that the cursor should begin at location in the index where **initial-value** would be found, if it was in fact it was in the index.   If **limit-value** is given then it specifies that the cursor will stop scanning the index forward when it reaches the point at which **limit-value** is or would be found in the index.    The **limit-value** is only used by **next-index-cursor** and not by

**previous**-**index**-**cursor**.   If **position** is **:last** and initial-value isn't given then the cursor will be positioned on the last item in the index.  This is useful if you wish to scan the index backwards.

 If **preread** is true then the contents of the whole index will be read in order to put the blocks of the index in the operating system's file cache.   If you plan to do a operation that will scan the entire index then do this preread will speed up a large index scan by a tremendous amount

 If **unique** is true then this is a very special kind of cursor that only returns the first item found for any index value.   This is designed for one purpose: to be able to find the set of unique values in the index, as shown here

```
(defun find-unique-index-values (class slot)
  (let ((cur (create-index-cursor class slot :unique t))
        (res))
    (loop
      (let ((value (next-index-cursor cur :oid-value :value)))
        (if* (null value) then (return))
        (push value res)))
    res))
```

An index cursor can only be created for slots on which an index was requested in the **defclass** expression that defined the class.

```
next-index-cursor ((cursor index-cursor) &key oid oid-value)
```

advances the cursor and returns the next object (or oid if the keyword **:oid** is specified as true**)**.  The first time that **next**-**index**-**cursor** is called after the cursor is created the cursor is not advanced and instead **next**-**index**-**cursor** returns the object to which the cursor points.

If **:oid**-**value** is specified as true then two values will be returned: the oid and the index value, however if **:oid**-**value** is specified as **:value** then only the index value is returned.

**next**-**index**-**cursor** returns **nil** when the cursor has reached the end of the values in the index or the **limit**-**value** specified in **create**-**index**-**cursor.**

```
previous-index-cursor ((cursor index-cursor)
                               &key oid oid-value)
```

This is just like **next**-**index**-**cursor** except that the cursor moves in the opposite direction towards the beginning of the index

```
free-index-cursor ((cursor index-cursor))
```

Once **next**-**index**-**cursor** returns **nil** the resources used by the cursor are freed.

If you wish to free an index cursor before **next-index-cursor** returns nil then call **free-index-cursor**.

As an example of the ors, this is the definition of **retrieve-from-index-range** described above:

```
(defun retrieve-from-index-range (class slot-name
                                  initial-value end-value
                                    &key (db *allegrocache*)
                                         oid)
  (let ((cursor (create-index-cursor class slot-name
                                     :db db
                                     :initial-value initial-value
                                     :limit-value end-value)))
    (do ((obj (next-index-cursor cursor :oid oid)
              (next-index-cursor cursor :oid oid))
         (res))
        ((null obj) (nreverse res))
      (push obj res))))
```

You can take advantage of the fact that indexes are sorted to retrieve the values in an index from lowest to highest.   In this example we create 100 objects with a random integer in the one slot named **a**.   Then we use a cursor to retrieve in order the 10 smallest random numbers computed and stored in slot **a**.

```
(defclass* sample () (a :index :any))

(defun testit ()
  (create-file-database "foo.db")

   ; store 100 objects in the database with
   ; random values for the a slot
  (dotimes (i 100)
    (make-instance 'sample :a (random 1000)))

  (commit)

  ; print out the values of the a slot for the lowest 10 values
  (let ((cur (create-index-cursor 'sample 'a)))
    (dotimes (i 10)
      (let ((obj (next-index-cursor cur)))
           (if* obj then (format t "a is ~s~%" (a obj)))))
    (free-index-cursor cur))
  (close-database))
```

```
Here it is in operation:

cl-user(3): (testit)
a is 0
a is 32
a is 34
a is 47
a is 47
a is 49
a is 63
a is 75
a is 91
a is 112
#<AllegroCache db "/home/jkf/acl8/src/cl/src/acache/foo.db"
-closed- @
  #x7215b3b2>
cl-user(4):
```

## *Expression Cursors*

Expression cursors allow a program to select objects of a class based on the value of  one or more slots.    This is a new facility and we will be extending it in future versions of AllegroCache.  Expression cursors are a somewhat low level facility and we will create a high level query system that will make use of expression cursors.

```
create-expression-cursor (class expression)
```
creates a cursor that returns all items of the given class that satisfies the given expression. **nil** may be returned instead of a cursor if the expression obviously denotes no objects of the given class.   **class** can be a symbol naming a class or a class object.

An expression cursor is similar to an index cursor except instead of specifying one slot and possibly a range of values to scan over, with an expression cursor you specify an expression over one or more slots.    In both an index cursor and an expression cursor, once the cursor is created you call **next-index-cursor** to retrieve the objects matching the cursor.    When you're finished with the cursor you call **free-index-cursor**.

One difference is that you cannot call **previous-index-cursor** on an expression cursor.

An expression has this syntax:

```
<expression>  := <slot-expr>  | (:limit <number> <subexpression>)

<subexpression>  := <slot-expr> | <logical-expr>

<slot-expr> := (= <slot-name> <value>) |
               (:range <slot-name> <min-value> [<limit-value>]) |
               (:regex <slot-name> <regular-expression>)

<logical-expr> := (or <subexpression>*) |  (and <subexpression>*)

<slot-name> :=  symbol naming a slot in the class over which the
cursor is being created

<number> := is a positive integer like 1000 or 200006

<regular-expression> is an ACL regexp2 style regular expression
```

In the above pseudo-BNF the square brackets [] denote an optional element, the vertical bar | denotes a choice of elements, and the asterisk * denotes a sequence of zero or more of the preceding element.

In an expression

- **=** means that it satisfies the **equal** predicate in Lisp
- **:range** means that the value is greater than or equal to the minimum value and less than the limit value.   A value of nil for the minimum value means start at the beginning of the index and a nil for the limit value means continue until the end of the index.
- **:regex** means that the value of the slot matches the regular expression (more on this below)
- **:limit** is used to control how much work the expression cursor computation code will do to compute the set of results (more on this below)

Here are some examples are expressions

| | |
|---|---|
| (= foo 4) | all instances whose foo slot is equal to 4 (this includes instances who foo slot has value 4.0) |
| (= foo bar) | all instances whose foo slot has the symbol bar as a value.  Note that we're not evaluating bar. |
| (= foo "fred") | all instances whose foo slot has the string "fred" as the value.   The test is case |

| (= foo 4) | all instances whose foo slot is equal to 4 (this includes instances who foo slot has value 4.0) |
| --- | --- |
| | sensitive |
| (= foo (a b c)) | all instances whose foo slot is equal to the list (a b c) |
| (:range foo 10 30) | all instances whose foo slot is greater than or equal to 10 and less than 30. |
| (:range foo 10) | all instances whose foo slot is greater than or equal to 10 |
| (:range foo 10 nil) | all instances whose foo slot is greater than or equal to 10 |
| (:range foo nil 100) | all instances whose foo slot is less than 100 |
| (:range foo "abc" "def") | all instances whose foo slot is string>= "abc" and string< "def" |
| (:regex foo "^abc") | all instances who's foo value begins with "abc"  (e.g. "abcdef are letters") |
| (:regex foo "[0-9]+") | all instances whose foo slot contains a string which contains an integer in it somewhere (e.g. "I like 31 flavors") |
| (:regex foo "\\.jpg$") | all instances whose foo slot is a string ending in  ".jpg" (e.g. "rambo.jpg") |

*Some* of the the above expressions work whether the slot foo is indexed or not.   Even those that do work over un-indexed slots will work much faster if the slot is indexed. Regular expression tests only work over indexed slots.

The objects found by the **:**range expressions where the minimum or maximum value is not given can be non-intuitive.   You have to consider the ordering of values in an index given in the table on page 28.    That table notes that strings are stored later in the table than integers.   Thus the expression (:range foo 10 nil) will find all objects whose foo slot contains 10 or greater as well as all objects whose foo slot contains a string.

However, **at the present time**, if the foo slot is **not** indexed the only objects that will be returned are those whose foo slot contains a number that is greater than or equal to 10. This difference is due to the fact that the computation of the range predicate is done differently if there is no index.   We may choose to change this behavior in a future version therefore it's best to avoid using range expressions that don't specify both minimum and limit when scanning over non-indexed slots.


Using **and** and **or** you can create compound expressions:

| (and (= foo 3) (= bar 4)) | all instances whose foo slot is equal to 3 and whose bar slot is equal to 4 |
|---|---|
| (or (= foo 10) (= foo 20)) | all instances whose foo slot is equal to 10 or 20 |
| (and (= first "joe") (= last "smith")) | all instances whose first slot is "joe" and whose last slot is "smith" |
| (or (and (= first "joe") (= last "smith")) (and (= first "jane") (= last "doe"))) | all instances whose first slot is "joe" and last "smith" or whose first is "jane" and last "doe" |
| (and (= foo  24) (:range bar 10 30)) | all instances whose foo slot is equal to 24 and whose bar slot is greater than or equal to 10 and less than 30 |

While the answer doesn't depend on the order of the arguments to an **and** or **or** expression, the time taken to compute the answer can vary based on the ordering.  It's best to put expressions over slots with indexes before expressions over  slots without indexes.

Testing predicates over non-indexed slots must be done in the client (if you're running a client/server version).  This requires passing messages between the client and server so be sure to take this into account when you are considering using expression cursors over non-indexed slots.

Expression cursors are designed to be used when you are testing more than one expression over one or more slots.   If you are just testing a single expression then you are better off using functions like retrieve-from-index and retrieve-from-index-range.


In this example of using expression cursors we'll create an object for each person and store their gender and age.

```
(defclass* person (:defprinter t)
         (gender :index :any)
         (age :index :any))

(defun populate ()
  (create-file-database "people.db")
  (dotimes (i 1000)
    (make-instance 'person
      :gender (aref '#(:m :f) (random 2))
      :age (random 100)))
  (commit)
  (close-database))
```

Calling (populate) will build a sample population.

Our query function is this.  It prints all members of the person class of the given gender and with an age in the given range.

```
(defun find-people (gender min-age limit-age)
  (open-file-database "people.db")
  (let ((cur (create-expression-cursor
                  'person
                  `(and (= gender ,gender)
                        (:range age ,min-age ,limit-age)))))
    (if* cur
        then (loop (let ((obj (next-index-cursor cur)))
                     (if* (null obj)
                        then (free-index-cursor cur)
                             (return))
                     (print obj))))
    (terpri)
    (close-database)
    :done))
```

We test out our query function:

```
cl-user(12): (find-people :m 30 35)

#<person [29] :m 34>
#<person [94] :m 31>
#<person [120] :m 31>
#<person [199] :m 32>
#<person [290] :m 31>
#<person [298] :m 30>
#<person [308] :m 30>
#<person [335] :m 33>
#<person [363] :m 34>
#<person [420] :m 32>
#<person [496] :m 31>
#<person [500] :m 34>
#<person [590] :m 34>
#<person [698] :m 31>
#<person [754] :m 30>
#<person [826] :m 34>
#<person [880] :m 32>
#<person [898] :m 33>
#<person [920] :m 32>
:done
cl-user(13):
```

**:regex**

Regular expression cursors work best if the regular expression specifies the first few characters the matching string must contain.   A regular expression such as "^SN" specifies that the first two characters must be S and N and thus allows the search for matches to zero in on that part of index where it might find such values.  If you wrote the expression as "SN" then you may get the same results however it will require every object be examined to see if its slot contains a string with "SN" anywhere in it.

The regular expression given need not matching the whole value of the slot.  The regular expression "^SN" will match "SN" but it will also match "SN1123-12-12123".   If you want your regular expression to match the whole value then use the caret and dollar sign characters as in  "^SN[-0-9]+$"  which will match "SN123-123-42334" but not "SN23 is good".

**:limit**

 When evaluating an expression cursor there are times when acache must build a table of intermediate results (such as when evaluating an **and** clause).   Once acache starts computing these intermediate results it can take a long time and use a lot of storage.  This clause allows you to state that if the number of intermediate results is larger than the first argument to :limit then the function that causes the computation (usually  next-index-cursor) will return **:precompute-limit-exceeded** right away instead of continue to compute the result set.

   One must keep in mind that a huge number of intermediate results doesn't mean that the number of final results will be large as well.

   Here is an example of an expression cursor expression where the :limit clause can be useful.

```
    (:limit 1000000
         (and  (:regex date "^200") (:regex itemnum "^300")))
```

## *Compressed Indexes*

Index files can grow very large because they contain a portion of the history of the index as well as the current values. Also they are designed for speed of access not space efficiency and for the ability to be modified while the application is running.

If a database is to be accessed read-only then one can convert the index files to a different format, one that is more space efficient and which eliminates the history of the index.  We call this format the trie format.  Trie is commonly pronounced "try".

```
(compress-indexes &key db remove-compressed

                             remove-original)
```

compress-indexes can only be called in standalone mode.  When you want to create a smaller size version of your database suitable only for read only operation you'll open it in standalone mode and call compress-indexes.  By default this function will look for any index file that has not been compressed since it was last modified and it will compress it.  The result will be a file ending in trie in the database directory.  After compress-indexes is done and the database is closed, if you open the database in read-only mode (either standalone or client-server) then acache will use the trie indexes instead of the normal indexes.  If you open the database in non read-only mode then acache will use the normal indexes.

Since your goal is to reduce disk space for your database it doesn't make much sense to keep the original and compressed indexes in the database dircectory.  Therefore you would typically create your compressed indexes this way:

```
  (compress-indexes :remove-original t)
```

which will remove the original index after it has compressed it.   The resulting database will take up less disk space but it can only be opened in read-only mode.   If you open it in non read-only mode then acache will not be able to find the index files it needs.

If you're experimenting with compressing indexes then you might find

```
  (compress-indexes :remove-compressed t)
```

to be useful.  It will only remove the compressed indexes and will do no compression.  It's useful for cleaning out your database directory if you decide that you're not longer interested in storing the compressed indexes.  If you do this you had better have the original indexes still in the database directory or you'll end up with no indexes at all and thus a database that is non-functional.

A compressed index is designed to be a drop in replacement for a regular index.  There are a few differences:

1.  the index is read-only.
2.  you can only scan the index in the forward direction (next-index-cursor) and not the reverse direction (previous-index-cursor).
3.  the ordering of index values is similar to the regular index for strings and integers. For all other values (such as floating point and infinities) the ordering is non-deterministic.

## *Transactions*

```
commit  (&key (db *allegrocache*))
```

The **commit** function attempt to save all changes made to persistent objects to the database. The commit will fail if an an object to be saved was modified by another database client since the time that this client got a copy of the object from the database. If AllegroCache is used in standalone mode then the commit cannot fail since there is only one client to the database.

If the commit succeeds the client is now viewing the latest state of the database (that is the client will now see changes all changes committed by all clients).

```
rollback  (&key (db *allegrocache*))
```

Abandon all changes made to the database since the last call to commit or rollback. Then advance the view of the database so that the client is now viewing the latest committed state of the database.

Instances of persistent classes created since the last commit or rollback will continue to exist in Lisp memory (because there's no way for AllegroCache to eliminate them) but the application should not attempt to access the slots of these objects. The application should eliminate all pointers to them so that Lisp can garbage collect them.

The following macro is useful when writing code to be run in the client/server mode where more than one client may be accessing the database.

```
(with-transaction-restart (&key (verbose t) message count)
                          &rest body)
```

The **body** forms are executed in a context where a commit failure will cause **rollback** to be called and then the **body** forms to be automatically re-executed. The arguments are

| verbose | If true then should a commit fail inside the body a message will be printed to *standard-output* . The default value for verbose is true. |
|---------|------------------------------------------------------------------------------------------------------------------------------------------|
| message | If a commit failure causes a message to be printed then the value of message (which should be a string) is printed as well as a way of personalizing the message to this particular use of with-transaction-restart |

| count | The number of times that the restart should be tried before giving up and signaling an error.  The default is 10.  Setting count to nil will allow restarts to be tried forever. |
|-------|------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|

Note that with-transaction-restart does not call commit itself.  It is up to your application to call commit.

Here's a typical use of this function.  We want to change a slot value from 34 to 52 and we just wrap that in a with-transaction-restart to ensure that it works well in a multi-user situation

```
(defun fixup ()
  (with-transaction-restart nil
    (let ((val (retrieve-from-index 'foo 'i 34)))
      (if* val
            then (setf (slot-value val 'i) 52)
                 (commit)))))
```

Let's use the example above to make another point about AllegroCache programming style.  Suppose you really did care to periodically check for a foo object with an i slot whose value is 34 and wanted to change it to 52.   Thus you arranged to call the **fixup** function above every few seconds.  If this is all the database operations you were doing then you would find that **fixup** would soon run out of work to do.   That's because  if fixup doesn't commit then fixup will never see any changes made by other processes to the foo objects.

Thus you're likely to want to write this as:

```
(defun fixup ()
  (rollback)   ;;; add this line
  (with-transaction-restart nil
    (let ((val (retrieve-from-index 'foo 'i 34)))
      (if* val
            then (setf (slot-value val 'i) 52)
                 (commit)))))
```

Now when fixup starts it first does a **rollback** which puts it up-to-date with the latest state of the database and then it begins its transaction.   Of course you don't want to call rollback unless you're sure that no other function has made a change to the persistent store that it wants to save.

```
last-transaction-number (&key (db *allegrocache*))
```

This returns the number of the last transaction that successfully committed to the given

database as far as this client is aware.   Remember that a client only synchronizes with the database when the client calls commit or rollback.   A client watching for the database to change by virtue of the work of other clients can just call this function after calling rollback.  If the value returned by last-transaction-number is different then some other client has successfully committed to the database.


## Saving and Restoring Databases


A database can be saved as an xml file and that xml file can be read and the database reconstructed.   The main purpose of saving and restoring a database is that it will allow you to migrate your data from one version of AllegroCache to the next.   Another reason is to compact the tables used in the database.   Also the fact that the save format is xml means that you can write tools outside of AllegroCache to analyze the database.

As noted above an AllegroCache database holds a number of different database states. When a database is saved only the most recent state is saved.   A Standalone AllegroCache only has one state so the saving of only one state is natural.   For the client-server AllegroCache the client initiates the saving of the database but it's the server that does the saving of the database (and stores it in a file on the server).   Regardless of the current transaction number of the client, the server stores the state associated with the most recent transaction number.


```
save-database (filename &key (db *allegrocache*)
                             file
                             (verbose t))
```

Store the latest state of the database in the file **filename**.
If **file** is given then it names a database (as in the first argument to open-file-database) and that database is opened, then saved as xml in **filename**, and then closed.
If **file** is not given then **db** should be an open database connection and that database is saved.
If **verbose** is true then comments will be written to the resulting xml file describing what the encoded data represents as lisp objects.   This can add a significant amount of time to the saving process so you'll likely want to disable verbose dumping for huge databases. The verbose switch has no effect in client/server mode since the server does not have enough information to do the annotation.

In client/server mode all other clients will be blocked from committing while one client does a save-database.


```
restore-database  (filename db-file)
```

Reads the xml file **filename** and creates the database in directory **db-file**.    If **db-file** already exists it will be overwritten.

In client/sever mode the client can call save-database.  However to restore the database you run inside a standalone AllegroCache.  Once the database is built you can then call start-server to begin a server process serving this database.

## *Transaction Logs*

Beginning in version 1.1.0 AllegroCache stores all data in files known as *transaction logs*. B-trees are still used as indexes to locate data in the transaction logs but the b-trees can be reconstructed using only the data in the transaction logs.   Thus the database can be said to be stored entirely in the transaction logs.

Each database has a set of transaction logs numbered from zero to N.   A transaction log is written to only at the end.   Once the log file reaches a certain size AllegroCache closes that log file and creates a new one.   Log files other than the newest are never modified by AllegroCache.  Older log files may need to be read by AllegroCache and if  they are then they are opened in read-only mode.

The purpose of this design is to allow the database to handle situations where it's abruptly terminated, such as loss of power or disk failure.   The only places that an AllegroCache database is vulnerable are the b-trees and last block of the newest log file.    The b-trees can be recreated from the log files and AllegroCache can locate the last valid record in the last log file, ignoring any illegal values place at the end of the log.

Another advantage of this design is that you can back up the database by just copying the log files to your backup media.  Once a log file is not the newest log file it will never change again, so there's no need to repeatedly back it up.

One problem with this design is that the log files contain the entire history of your use of the database and this can take a lot of disk space.   AllegroCache contains a function to compress log files, eliminating that data that will never be referenced again.   The old log files are not modified, they are simply renamed and new smaller log files are created in their place.  The database administrator will delete the old log files or move them to a backup device in order to complete the process of freeing up space in the database.

First we'll describe how to recover a database from its log files and then how to compress a database.

## Database Recovery from Log Files

The log files for a database have names of the form acNNNNNN.dat where NNNNNN is a number beginning with 000000 followed by  000001 etc.   The oldest log file is ac000000.dat.

Log files contain a record of the operations you performed on the database, and in particular they record all calls to commit.   You can recover the database to the state it was in following any successful commit (i.e. a commit that didn't signal an error).

If you want to recover the database to the latest state recorded in the log files then you can

skip right to call to **recover-from-log**.   If you want to recover to some intermediate point
then you call this function:

```
recovery-times (directory)
```

This function reads all the log files in the directory and returns a list of all the successful
commits.  Each item in the list is a transaction number followed by the time of the
commit in universal time and in human readable format.

For example

```
cl-user(8): (pprint (recovery-times "bigac.db"))

((16 3352128789 "3/23/2006 10:53:09") (15 3352128789 "3/23/2006 10:53:09")
 (14 3352128788 "3/23/2006 10:53:08") (13 3352128787 "3/23/2006 10:53:07")
 (12 3352128787 "3/23/2006 10:53:07") (11 3352128786 "3/23/2006 10:53:06")
 (10 3352128785 "3/23/2006 10:53:05") (9 3352128785 "3/23/2006 10:53:05")
 (8 3352128785 "3/23/2006 10:53:05") (7 3352128785 "3/23/2006 10:53:05")
 (6 3352128785 "3/23/2006 10:53:05") (1 3352128785 "3/23/2006 10:53:05"))
cl-user(9):
```

To recover the database from its logs you call this function

```
recover-from-log (log-directory db-directory
                         &key log-size transaction)
```

A fresh database is created in db-directory and the logs from log-directory are read and
used to fill the database.   If the :transaction argument is given then the database will be
filled with all data up to and including that transaction number (see **recovery-times** for
how to determine the valid transaction numbers in a set of log files).   The :log-size
argument can be given to specify the log size for the newly created database.   After the
database is recovered it is closed.

## Compressing Log Files

When log files accumulate you'll want to remove old unreachable data from them.
Compression is a multi-stage process.   First the log files are scanned to find all the
success commits (since knowing this means that data for unsuccessful commits needn't be
saved).  Next each log file is scanned and still valid data is copied to a new version of the
log file.   If  the new log file is identical to the old log file (i.e no compression could be
done) then the new log file is removed as there is no need to keep a duplicate copy.  Once
all log files have been scanned the database is locked and the old log files are renamed
(and given a suffix .bNNNN) and the new log files take the place of the old log files.
Then the b-tree's are modified to point into the correct positions in the new log files.
Finally the database is unlocked.

There are two important points to note.  The first is that the old log files are not deleted.  Thus if something goes wrong in the compression you can still get back to your old database.  Second,  in a client-server database the compression can be happening while database accesses from other clients continue.  It's only in the final phases that the other database clients are locked out.

```
compress-log-files (&key (db *allegrocache*)
                            (start-log 0) (verbose t))
```

Compress the log files for the given database.    Start the compression with log file number **start-log** and compress all log files up to the next to newest log file.  If **verbose** is true print out informative messages about the progress.

Once the compression is done you'll have to remove the old versions of the log files manually.  They will end in .bNNNN where NNNN is a number indicating that this file is a back up of a log file found during the NNNNth call to compress-log-files.

## Miscellaneous

In the standalone version and in the client side of the client/server version every persistent class in the database has a cache of objects of that class.    The cache must be at least as large as all the objects currently pointed to by objects in the lisp heap.   It is useful to make it even larger though so that objects that might be referenced at some future time are found in the cache rather than having to be created by reading them from the database.  A cache that is too large will be a burden to Lisp's storage management system.    The desired size of the cache can vary based on what the program is doing at the moment.  If the program is simply creating a large group of objects, committing them to the database and then not referring to the objects again the cache can be small.   If the program is looking at lots of objects during a computation then a large cache can be very helpful.

The programmer cannot control the exact size of the cache but the programmer can give hints to AllegroCache to tell it how the cache size should change.

The **object-cache-size** generic function returns AllegroCache's idea of the desired cache size for a class.   There are three methods which all do the same thing but take different arguments:

```
object-cache-size  (class-name &key (db *allegrocache*))

object-cache-size  ((class persistent-class)
                      &key (db *allegrocache*))


object-cache-size  ((class ac-class)
                      &key (db *allegrocache*))
```

These methods return the maximum number of objects AllegroCache will attempt to store in the cache for the given class.

You can use (setf object-cache-size) to change the maximum cache size.   Note that this is an advisory number and the cache size will not change immediately.  However during subsequent automatic cache object evictions AllegroCache will bring the cache size closer to the number you specified.    After calling (setf object-cache-size) the value returned by object-cache-size may not be exactly what you specified.   Object eviction is a process that runs concurrent with regular database operations  and the value of object-cache-size value us used by AllegroCache itself to track the eviction process.

```
cl-user(7): (object-cache-size 'foob)
70000
cl-user(8): (setf (object-cache-size 'foob) 5000)
5000
cl-user(9)
```

### flush-object-cache  &key all class (db *allegrocache*))

Start the process of removing from the cache all objects for which there are no references in the lisp heap.   The flush-object-cache function causes all objects in the cache to now be referenced inside the cache by a weak vector.   When the last (non-weak) reference in the heap to an object goes away and then a garbage collection occurs, that object will be removed from the object cache.  If an object was referenced from old space then a full global gc will be required to detect when that old-space reference no longer exists.

Thus flush-object-cache does not have an immediate effect on the size of the cache but it starts a process by which  unreferenced objects will be eliminated from the cache.

If **:all t** is specified then the object caches for all classes will be flushed.  If a value for **:class** is specified then it should be a class name or class object and that class's cache will be flushed.

### client-connections (&key (db *allegrocache*))

This function returns a list of connections to the database server in client/server mode. In standalone mode it always returns nil.

The first value in the returned list is the current connection, i.e. the connection used to make the call to client-connections.

Each connection is described by a cons, the car of which is the string holding the dotted representation of the IP address of the machine connecting to the database and the cdr is the port number of the socket on the remote machine.

```
cl-user(8): (client-connections)
(("127.0.0.1" . 50223) ("192.132.95.84" . 54981))
cl-user(9):
```

## kill-client-connection (connection &key (db *allegrocache*))

Disconnect the given client from the database and return true if the disconnection was done.   It is not permitted to disconnect the client calling kill-client-connection and attempts to do so will be ignored and kill-client-connection will return nil.   In order to disconnect the current client call close-database.

A client connection is denoted by a cons of the client ip address (in string form) and the port on the client.   This is the same form as returned by client-connections.

For example to close of all client connections you could do

```
cl-user(5): (mapc #'kill-client-connection (client-connections))
(("127.0.0.1" . 56775) ("127.0.0.1" . 44975))
cl-user(6): (close-database)
#<AllegroCache db "port 56775 to localhost:3333" -closed- @ #x10016a1492>
cl-user(7):
```

## connection-alive-p (&key (db *allegrocache*))

Return true if the database object has not been closed and is connected to a live server. Note that even if this function returns true the server could die seconds later.   This is just the nature of networked applications.    It's best to handle server failure with every client operation rather than calling this function once and, it having returned true, expecting the server to remain up for a certain duration of time afterwards.

If the server is found to be down the database connection will be closed with close-database.

connection-alive-p will work on standalone database as well in which case it works just like database-open-p.

It's possible to store objects of non-persistent classes in the database but the user must help AllegroCache with the process.   The user must convert the object to be stored into something that AllegroCache already knows how to store and when the object is to be reconstructed the user must write code to do the reconstruction.    The user must define methods on the generic functions  encode-object and decode-object for the class of object they wish to store.   You should recall that the defstruct macro defines a class as well so this protocol is used to store structures built by defstruct as well.

User code must define these methods

## encode-object ((obj my-class))

## decode-object ((obj my-class) object-values)

The encode-object function takes an object to be stored and returns a value that AllegroCache knows how to store.   For example given this definition of a non-persistent class:

```
(defclass frob ()
  ((frob-a :accessor frob-a :initarg :frob-a)
   (frob-b :initarg :frob-b)))
```

You could write this encoding method:

```
(defmethod encode-object ((object frob))
  (flet ((do-slot (slot)
           (if* (slot-boundp object slot)
               then (cons slot (slot-value object slot)))))
    (mapcar #'do-slot '(frob-a frob-b))))
```

Which returns a list of conses holding the slot names and values for the object.

The next step is writing a decode-object method to reconstruct the object.   This method must create the object and fill in the slots appropriately.   The object passed into decode-object can not be used – a new object must be created.   Here is the method for our sample frob class:

```
(defmethod decode-object ((object frob) slot-vals)
  (let ((frob (make-instance 'frob)))
    (dolist  (sv slot-vals)
      (setf (slot-value frob (car sv)) (cdr sv)))
    frob))
```

```
mark-instance-modified (persistent-instance)
```

The given persistent-instance is marked as having been modified during the current transaction.   This means that the value of this object will be written to the database on the next commit.  Normally it isn't necessary to call this function as instances are automatically marked modified when values are stored in their slots.   However if the value of a slot is a lisp object that itself can be modified (such as an array or a cons) then modifications to that lisp object will not cause the persistent object pointing to it to be considered modified.

For example here we set an element in the array that's stored in a slot in a persistent object and then we call mark-instance-modified so that AllegroCache knows that this persistent object has been modified in this transaction.

```
(setf (aref (slot-value obj 'a) 23) 'foo)
(mark-instance-modified obj)
```

```
delete-persistent-class ((class persistent-class)
                            &key (db *allegrocache*))
```

Delete the information about the **class** from the database **db**.   The class definition will remain in the Lisp's heap.   When the database is opened again in a fresh lisp the definition of **class** will not be loaded into Lisp's heap.

Like other operations on the database, the delete will not take effect until a **commit** is done.

delete-persistent-class will **not** delete objects of this class from the database.   It is an error to reference a persistent object of  a deleted class.   It is an error if some other class still references this deleted class.

This function is rarely needed.  The cost of storing a class in the database is minimal so there's no need to delete them for that reason.

Example:

```
cl-user(1): (delete-persistent-class (find-class 'myclass))
cl-user(2): (commit)
```

```
(database-of instance)
```

returns the database connection of the given persistent instance.   Normally this would be the same value as *allegrocache* unless the application is using multiple database connections simultaneously.

```
(close-compressed-index-streams &key (db *allegrocache*))
```

Close any open file streams to compressed index files.  When doing index operations on a compressed index database a file stream is opened for the compressed index file and that stream will remain open until the database is closed.   This makes index operations run more quickly.   If there are a huge number of indexes referenced while the database is open then this may use up too many open file handles which are a limited system resource.  This function will cause all of those compressed index files to be closed and then they will be reopened again as needed.    It is very unlikely that this function will be needed as most systems allow thousands of files to be open at once, however if you see messages about a failure to open a file due to a exceeding the number of file descriptors and you're using a database with compressed indexes (which means also that you're opening it read-only) then this function will prove useful.

## *Where the Illusion Breaks Down*

AllegroCache tries to present the illusion of normal CLOS programming but with persistence.   This is done using the CLOS metaobject protocol which was designed for purposes just like this.  There are however certain operations that are not supported yet and certain operations that will never be supported.  We'll describe these operations here

- **change-class** not supported.   You cannot use **change-class** if the old or new class is a persistent class.    We may make this work in a future version.

- Destructive modifications to non-CLOS objects are not noticed by AllegroCache. If you setf an array element or the car or cdr of a cons AllegroCache cannot tell that that modification was done.  Thus AllegroCache's will not automatically save that change when a commit is done nor will it undo that change if a rollback is done.   This will never work in AllegroCache.   See the documentation for mark-instance-modified for a way to force a CLOS object and all its slots to be written to the database upon a commit.

## *Upgrading Databases*

Whenever possible when we introduce a new version of AllegroCache we use the same database format.  This allows you to use the new features on your existing databases. If the database format changes then we include functions that allow you migrate your database from the old version ot the new version.

In this section we'll note those version of AllegroCache where the database format changed and we'll describe how to migrate your old databases.  Our goal is to allow you to use older databases in newer versions of AllegroCache.  We are not concerned about using newer databases in older versions of AllegroCache although that often works.

## AllegroCache versions 1.1.0 to 2.0.1

All of these AllegroCache versions have the same database format.   We'll assume that there are no databases in use from earlier versions (if there are please contact us and we'll tell you how to upgrade).

We denote these databases as **version 8** databases.  If you want to see which version your database is use the save-database command and examine the beginning of the xml file created for the value of **dbver** in the **admin** element.  Here you can see its value is 8:

```
<admin
   dbver='8'
   next-oid='163239010'
   next-classid='94'
   next-ivnum='1'
   last-trans='885838'
  />
```

## AllegroCache version 2.1.0

In this version we changed the format of index btrees so that expression cursors would work..  This is a **version 9** database.

This is an unusual database upgrade in that the upgrade is optional.  Here are the rules:

•    databases created with AllegroCache 2.1.0 will be version 9 databases

•    version 8 databases can be opened and used with AllegroCache 2.1.0 and those databases will remain version 8 databases.

•    If you use **save-database** to save a version 8 database and then in AllegroCache 2.1.0 use **restore-database** to reconstruct that database then that database will continue to be a version 8 database.

•    to turn a version 8 database into a version 9 database run the `(upgrade-index-type)` function while the database is open in standalone mode.   This function will do nothing unless it's run in a version 8 database.

•    The expression cursor functions will signal an error unless the database version is at least 9


# Lisp Multiprocessing

In this section we'll describe how to use AllegroCache with an application that takes advantage of Lisp's own threading system (which is officially called the Lisp Multiprocessing System).

The most important rule is that **only one thread at a time should be using a given database connection**.  *Using* a database connection includes
   •    reading or writing the slot of a persistent object
   •    locating  a persistent object from an index
   •    commit or rollback
   •    creating a persistent object
   •    deleting a persistent object

You can be sure that threads don't overlap their use of a connection through the use of locks.  If this is too inconvenient you can create a pool of database connections and have each thread pick a connection out of the pool for its exclusive use, returning the connection to the pool when the thread is finished.

When running an AllegroCache program the variable **\*allegrocache\*** should be bound to the database connection.   This allows you to call **make-instance** for example without specifying a database connection to use should make-instance have to create a persistent object.   Thus if you want a thread to use AllegroCache be sure to bind \*allegrocache\* when you start the thread.

When a persistent object is created it *belongs* to the database connection which was used to create it.   If you just have one database connection and it's open, then you can treat persistent objects  just like transient objects.   However if  you have multiple connections open then you have to be very careful to not confuse persistent objects that belong to different connections.   It's even possible to have two different copies of the same persistent object in the Lisp heap, each copy belonging to a different connection.

# Bulk Loading

 AllegroCache is tuned for the typical workload which is a mixture of reading objects from the database, changing values of slots and creating new objects.   A different and important workload is that of adding a massive number of objects to the database.  We call this workload: bulk loading.

 Bulk loading usually occurs when you first create a database.  It can also occur if your application periodically adds a large amount of data to the database.

 What characterizes our special bulk loading mode is the addition of new objects with indexed slots and the lack of querying about information in those objects until the bulk loading is complete.    Adding objects without indexed slots does not require the special bulk loading mode to run efficiently.

 What makes bulk loading slow under the normal AllegroCache settings is the building of indexes for the new objects.  If the index values added are randomly distributed then the action of adding values to the index btrees will tend to touch every btree page.  If the size of the btree is much larger than the size of the memory cache (specified by :class-cache-size when the database is opened) then AllegroCache will spend most if it's time reading and writing from the disk when adding objects and indexing them.  When AllegroCache is in bulk loading mode the addition of the index values to the btrees is delayed until bulk mode finishes with the result being that far fewer disk reads and writes are necessary to update the index.

 **Beginning with version 2.1.3 bulk loading is supported in client/server mode as well as standalone mode.**

 Bulk loading begins when the program calls `(commit :bulk-load :start)`. It continues until the program calls `(commit :bulk-load :end)`. Between the start and end of bulk loading there will usually be many commits (perhaps a commit after each 10,000 objects are added). In the intermediate commits the :bulk-load argument is either not given, or is given with the value :start, both having the same effect of continuing the bulk load operation.

 Efficient bulk load requires more than just putting AllegroCache into bulk loading mode. Below we show an example of creating and bulk loading records of class crecord which

has three indexed slots.  The indexed slots are filled with random values thus making
normal indexing operations slow.

 We've used this code to build a billion object database.  What's particularly noteworthy
about this function is that the cost per object added is the same from the first object added
to the billionth object added.  This is a very desirable property as it says that the size of
your database is only limited by the amount of disk space you have available. After the
code we'll describe in detail why this function was written the way it was.

```lisp
(defpackage :user (:use :db.ac :db.ac.utils))

(defclass* crecord ()
  ((cell-id)
   (mobile-id      :index :any)
   (called-party-no :index :any)
   (calling-party-no :index :any)))


(defun bulk-build (count)
  ;;
  ;; create a new database and add count crecord objects to that
  ;; database
  ;;
  (sys:resize-areas :old (* 100 1024 1024))
  (setf (sys:gsgc-parameter :generation-spread) 20)
  (create-file-database "testa.db"
                          :class-cache-size (* 50 1024 1024)
                          :object-cache-size 11000)


  (commit :bulk-load :start)

  (let ((last-time (get-universal-time)))

    (dotimes (i   count)
      (make-instance 'crecord :cell-id (random 1000000)
                      :mobile-id (random 1000000)
                      :called-party-no (random 1000000)
                      :calling-party-no (random 1000000))
      (if* (zerop (mod (1+ i) 10000))
          then (commit  :sync nil)
               (flush-object-cache :all t)
               (if* (zerop (mod (1+ i) 100000))
                   then
                        (let ((this-time (get-universal-time)))
```

```
                              (format t "~14d  ~d secs~%" (1+ i)
                                      (- this-time last-time))
                              (setq last-time this-time))))))
  (commit :bulk-load :end)
  (close-database)
  )
```

The call to
```
  (sys:resize-areas :old (* 100 1024 1024))
```
ensures that there is at least 100MB free in old space.  When btree buffers are allocated they are allocated in old space immediately thus we know that this application will soon be allocating space in old space.   If old space is grown as a result of objects being allocated then the program will end up with many small old spaces. The more old spaces the slower the garbage collection runs.  By preallocating a large old space we reduce the number of old spaces that will be created.

 If you're not sure how much old space to allocate you can run your program once and use (room t) to see how much old space was required.

The call to
```
  (setf (sys:gsgc-parameter :generation-spread) 20)
```
increases the time objects stay in new space before they are moved to old space.  The default generation spread is 4. We do this because in this function we will be creating 10,000 objects and then committing after which point we don't need those 10,000 objects any longer.   If some of those objects are moved to old space then getting rid of them would require a global gc.  Thus by setting the generation spread high we can reduce the likelihood that any of these 10,000 objects get moved to old space.

We create the database here
```
  (create-file-database "testa.db"
                              :class-cache-size (* 50 1024 1024)
                              :object-cache-size 11000)
```
During the building phase it's a good idea to specify as large a class cache size as you can. We ran this on a 64-bit machine with 2gb of memory so 50Mb of cache per btree is not going to be a problem. In this particular example all of the index values are fixnums and that triggers special code in AllegroCache which bypasses using btrees to store index values until the last step.  Thus in this test case we don't need a very large btree cache and even 50Mb is excessive.

We want to keep the object cache small as well. We'll be creating many objects and then never looking at them again.  Thus we don't need a large object cache.  We specify a size of 11000 in the create-file-database since we'll be committing after every 10,000 objects are created.

The call to
                `(commit :bulk-load :start)`
does a commit (but we've done nothing so no changes are saved) and then it puts
AllegroCache into bulk loading mode.

After every 10,000 objects have been created we call
   `(commit  :sync nil)`
This commit causes the 10,000 newly created objects to be written to the btree buffers in
memory.  Since we specified :sync nil all of the btree buffers won't necessarily be written
to the disk.   This speeds up the commit operation but if the application or machine
should crash the database files would likely be inconsistent.  Thus we only specify :sync
nil here because we know that in this case if the application doesn't finish running we'll
just delete the database and start over again.

The next function called is
     `(flush-object-cache :all t)`
this is optional and is only done to increase performance.  We know that our program isn't
going to reference (any time soon) any of the 10,000 objects in the cache we just
committed.  So we tell AllegroCache to try to remove them all from the cache.
AllegroCache can only remove an object from the cache if it can prove that there are no
references to the object from any place in the lisp heap and this will be determined after
the next garbage collection takes place. Without the call to flush-object-cache the process
of removing objects from the cache would still take place (because on database open we
specified that we wanted the object cache to hold only 11,000 objects).  However the
process of evicting objects from the table would be much more gradual and it would
involve many scans of the object table.  Therefore calling flush-object-cache speeds up
the eviction process.


Every 100,000 objects we print out the total object count and the time taken to build and
commit those 100,000 objects.    What we look for is the time to commit to remain
constant or to grow very very gradually.   This is necessary in order to be able to create a
large number of objects in a reasonable amount of time.

In our example above this is the information printed for the first 4 million objects created:

```
cl-user(6): (bulk-build 50000000)
        100000  5 secs
        200000  4 secs
        300000  4 secs
        400000  4 secs
        500000  5 secs
        600000  4 secs
```

```
         700000   4 secs
         800000   5 secs
         900000   4 secs
        1000000   5 secs
        1100000   4 secs
        1200000   4 secs
        1300000   4 secs
        1400000   4 secs
        1500000   5 secs
        1600000   4 secs
        1700000   5 secs
        1800000   5 secs
        1900000   4 secs
        2000000   4 secs
        2100000   5 secs
        2200000   4 secs
        2300000   4 secs
        2400000   4 secs
        2500000   5 secs
        2600000   4 secs
        2700000   4 secs
        2800000   4 secs
        2900000   5 secs
        3000000   4 secs
        3100000   4 secs
        3200000   4 secs
        3300000   5 secs
        3400000   5 secs
        3500000   4 secs
        3600000   4 secs
        3700000   5 secs
        3800000   4 secs
        3900000   4 secs
        4000000   5 secs
```

We've let that build process run up to a billion objects and the time per 100,000 object still remains constant.

To see what the numbers look like when the building process is not going well remove the call to `(commit :bulk-load :start)` from our example function.  Now the build will be done in the normal AllegroCache mode:

```
cl-user(4): (bulk-build 50000000)
         100000   9 secs
         200000   9 secs
         300000   8 secs
         400000   9 secs
         500000   9 secs
```

```
 600000   9 secs
 700000   9 secs
 800000   9 secs
 900000   10 secs
1000000   8 secs
1100000   10 secs
1200000   9 secs
1300000   10 secs
1400000   9 secs
1500000   10 secs
1600000   10 secs
1700000   9 secs
1800000   9 secs
1900000   10 secs
2000000   9 secs
2100000   11 secs
2200000   9 secs
2300000   16 secs
2400000   13 secs
2500000   19 secs
2600000   19 secs
2700000   23 secs
2800000   23 secs
2900000   29 secs
3000000   33 secs
3100000   46 secs
3200000   36 secs
3300000   38 secs
3400000   43 secs
3500000   49 secs
3600000   51 secs
3700000   56 secs
3800000   53 secs
3900000   57 secs
4000000   57 secs
```

You'll note that it starts running slower than in bulk load mode. That is because AllegroCache is filling in the index btrees right away rather than waiting until the end of bulk load mode.   Thus this slowdown is expected.   What is bad is that the cost per 100,000 objects starts to grow when the total count reaches 2.3 million objects.  This is the point at which the index btrees start outgrowing their caches in memory.   As the btrees continue to grow and the cache size remains constant the odds of having to read a block from a disk grows and this means that the insertion process will continue to slow down.

If you can run the application on a machine with a large amount of physical memory then you can specify a huge cache size and delay the point at which the size of the index btree

exceeds the cache in memory. For example on a 64-bit machine with 16gb of memory you could allocate 2gb per class cache and that would allow you to bulk load with all the btree pages in the cache.

The call to
```
  (commit :bulk-load :end)
```
ends bulk loading and causes all index information stored in temporary files to be copied to the actual database index files.   This will be a time consuming operations if there are many millions of index entries to copy.

In summary AllegroCache, like other databases, works fastest if all the data is stored in memory.  This is accomplished in AllegroCache by specifying a class-cache-size that's larger than the largest btree in the database.

Too often you don't have the luxury of keeping the whole database in memory. When this happens you have to make optimal use of the memory that you can allocate to holding database pages.   If you know you would be inserting a lot of random data into an index then you should make use of AllegroCache's bulk loading mode to speed up the insertion of that index information.

# AllegroCache Utilities

In this section we'll describe functions that while strictly not part of AllegroCache are nevertheless useful for AllegroCache users.    We've put these functions and macros in a separate package:  "**db.allegrocache.utils**" or "**db.ac.utils**".

## defclass*/defprinter

The syntax for defclass is rather verbose.    Creating a class where all slots have initargs, initforms and accessors requires typing a large expression.   If you're doing a demo or just exploratory programming this amount of typing can be a burden.

**defclass\*** is a macro that allows you to define persistent classes in much the same way that defstruct allows you to define structures.

For example

```
(defclass* foo () a b c)
```

is equivalent to

```
(defclass foo nil
          ((a :accessor a :initform nil :initarg :a)
           (b :accessor b :initform nil :initarg :b)
           (c :accessor c :initform nil :initarg :c))
          (:metaclass persistent-class))
```

**defprinter** is a macro which allows you to easily create print-object methods for persistent classes.

Using our **foo** class defined above we see the effect of using defprinter to define a print-object method.

```
cl-user(4): (setq x (make-instance 'foo :a 1 :b 2 :c 3))
#<foo oid: 12, ver 5, trans: nil,  modified @ #x724aa26a>
cl-user(5): (defprinter foo a c)
#<standard-method print-object (foo t)>
cl-user(6): x
#<foo [12]* 1 3>
cl-user(7):
```

You can see that defprinter allow you to specify which slots should have their values printed as part of the printed representation of the object.

It's also possible to tell defclass* to expand into a call to defprinter.

Here are the details on these two macros:

```
(defclass* classname (&rest supers-and-flags) &rest slots)
```

**classname** is the class being defined.  It is a symbol.

**supers-and-flags** is a combination of superclasses for the class being defined and flags to be passed to the defclass* macro.  The superclasses are always first.  The flags are always keywords so that is how they are distinguished from superclasses.  For example you may have

```
(defclass* foo (bar baz :init nil :defprinter t) a b)
```
The flags are:

| Flag | Default | Meaning |
| --- | --- | --- |
| :conc-name | nil | If nil then accessors are given the same name as the slot name.  If true then accessors are given the name *classname-slotname.*   The accessors can be overridden on a slot by slot basis by specifying an :accessor in the slot definition. |
| :defprinter | value of  *default-defprinter* | If true then defprinter is invoked and passed all the slots given in the defclass* |

| Flag | Default | Meaning |
|------|---------|---------|
| :init | t | If true then slots are initialized to nil.   The initform can be overridden on a slot by slot basis by specifying an initform in the slot definition. |
| :make | nil | If true then a make-*classname* macro is created which just turns into (make-instance '*classname* ...) |
| :print | value of<br><br>*default-print-defclass*-expansion* | If true then the resulting defclass is printed to standard output. |

The **slots** are a list of slot definitions using the same syntax as found in **defclass**.   We allow two ways to specifying the slot definitions.

```
(defclass* foo () a b c)
(defclass* foo () (a b c))
```
The former is like defstruct and the latter is like defclass.    The defclass* macro is able to distinguish which form you're using.

We've just be using symbols as slot definitions but you can use the full defclass syntax.

```
(defclass* foo () (a :index :any) (b :accessor bee :initform 3)
c)
```

```
(defprinter classname &rest slots-names)
```
Defines a print-object method on **classname** which causes the values of the given slots to be printed when the object is printed.   The printed form looks like

```
cl-user(5): (defprinter foo a c)
#<standard-method print-object (foo t)>
cl-user(6): (make-instance 'foo :a 1 :b 2 :c 3)
#<foo [12]* 1 3>
cl-user(7):
```

Where foo is the class name, the 12 in brackets is the object identifier (oid), and the asterisk after the brackets indicates that this object has modified in the current transaction. Following all that are the values of the slots whose name were given in the defprinter call.

# Index