



# AllegroCache: an Introduction

---

David Margolies ([dm@franz.com](mailto:dm@franz.com))

Questions and comments to  
[support@franz.com](mailto:support@franz.com)



## Topics

- What is AllegroCache?
- Getting started
- Persistent classes and objects
- Transactions, committing and rollback
- Indexes and cursors
- Multiple users
- Transactions, committing and rollback again
- Other features
- Object editor
- Documentation



## What is AllegroCache?

- A high-performance, dynamic object caching database system.
- Programmers to work directly with objects as if they were in memory while in fact the object data is always stored persistently.
- It meets the classic ACID (atomicity, consistency, isolation, durability) requirements for a reliable and robust database.



## Talk assumptions

- You have some familiarity with Lisp and CLOS programming.
- You have some familiarity with databases and issues of consistency, multiple users making changes, data loss and recovery, and so on.
- You have access to Allegro CL.



## Our background example

We will be using an example of a theater box office, with performance events, tickets and sales

- This is a simple example but one that is (we hope) easy to understand.
- We will show how we can define persistent classes and then change them when the requirements evolve.



## AllegroCache paradigm

- You create classes and instances just as with regular CLOS programming.
- When the class is persistent, the definition and instances are stored in the database.
- You do have to say when you want your changes stored: that is not done automatically (to avoid inconsistency).



## Getting started

- Load AllegroCache  
(require :acache) will signal an error which will show what versions are available.  
(require :acache "acache-2.1.12.fasl") – or whatever version you have and want - loads AllegroCache.
- AllegroCache package is :db.allegrocache, nicknamed :db.ac.



## Opening a database

(open-file-database "theater" ;; names local directory  
:if-does-not-exist :create  
:if-exists :supersede) ;; or :open (the default) if you  
;; already have a database

**create-file-database** calls **open-file-database** with those arguments.

Additional keyword arguments allow specifying sizes and other features for better performance (see reference manual).

Variable `*allegrocache*` set to opened database.





## Persistent classes and objects

- You create persistent objects by creating persistent classes and then instances of those classes

```
(defclass performance ()  
  ((date :initarg :date :index :any  
        :accessor perf-date)  
   (time :initarg :time :accessor perf-time)  
   (unsold :initarg :available :initform 100  
          :accessor perf-unsold)  
   (sold :initarg :sold :initform 0  
        :accessor perf-sold)  
   (event-title :initarg :event-title :index :any  
               :accessor perf-event-title))  
  (:metaclass persistent-class))
```

This class is now defined and instances can be stored in the database.



## Creating persistent objects

```
(make-instance 'performance  
  :event-title "Rolling Stones"  
  :date "20101010" :time "2000")
```

```
(make-instance 'performance  
  :event-title "Joan Rivers"  
  :date "20101101" :time "1900")
```

These two events are created, but not yet stored.  
You store them by evaluating (commit).



## What can be stored?

- Symbols
- Numbers (integers and floats)
- String or character
- Vector (type t) or cons, where values are from this list
- Persistent objects
- Map or set (discussed later)
- Structure (when encode-object method is defined)



## Transactions, committing and rollback

- A transaction is an action which syncs the local status with the stored status (if possible).
- The database is always in a consistent state.
- When you make changes in Lisp, they are not stored until you say they should be (by calling (commit) or an equivalent).
- They are only stored if all changes can be made.
- If any change fails, no change is made.
- (rollback) undoes any local changes.
- In single user mode, commits never fail.



## Single-user database connection

- When only one person is using the database, the transaction model is simple because you make the changes. Commits will not fail.
- In our example, we define a **sell** function to sell tickets to a performance:

```
(defun sell (perf-title perf-date perf-time number)
  (let* ((perf-title-set
         (retrieve-from-index 'performance
                              'event-title perf-title :all t))
        (perf avail sold))
    (setq perf (dolist (i perf-title-set)
                      (if (and (string-equal (perf-date i) perf-date)
                              (string-equal (perf-time i) perf-time))
                          (return i))))
    (if (null perf)
        (error "no such performance with that title, date, and time"))
    (setq avail (perf-unsold perf))
    (setq sold (perf-sold perf))
    (if (<= number avail)
        (progn (setf (perf-avail perf) (- avail number))
               (setf (perf-sold perf) (+ sold number)))
        (error "only ~d available, cannot sell ~d" avail number))))
```



## Digression: we need an perf-id slot

- The only way to get an instance is to use event-title, date, and time.
- If we had an perf-id slot, we could use that.

```
(defclass performance ()  
  ((date :initarg :date :index :any :accessor perf-date)  
   (time :initarg :time :accessor perf-time)  
   (unsold :initarg :available :initform 100  
            :accessor perf-unsold)  
   (sold :initarg :sold :initform 0 :accessor perf-sold)  
   (event-title :initarg :title :index :any  
                 :accessor perf-event-title))  
  (perf-id :initarg :perf-id :index :any-unique  
   :accessor perf-id)  
  (:metaclass persistent-class))
```



## When we commit, changed definition will be seen

- We must set the perf-id's for existing objects, doing something like:

```
(doclass (i 'performance)
  (format t "title ~s, date ~s, time ~s ~%"
    (perf-event-title i) (perf-date i) (perf-time i))
  (setf (perf-id i) (read))))
```

- **doclass** iterates over objects in a class. New **sell** function:

```
(defun sell (perf-id number)
  (let* ((perf (retrieve-from-index 'performance 'perf-id))
        (avail (perf-unsold perf))
        (sold (perf-sold perf)))
    (if (<= number avail)
      (progn (setf (perf-unsold perf) (- avail number))
            (setf (perf-sold perf) (+ sold number)))
      (error
        "only ~d available, cannot sell ~d" avail number))))
```



## Indexes

- When a slot is defined with **:index :any**, an index is created which goes from a slot value to a set of instances.
- When a slot is defined with **:index :any-unique**, an index is created which goes from a slot value to a single instance.
- **retrieve-from-index** get the list of instances (for :any) or single instance (for :any-unique).
- Indexes order objects as follows: lisp values (except below), numbers, NaNs, strings.





## Cursors

- A cursor is a pointer that iterates through the instances of a class based on the indexed values of a slot.
- An index cursor is created with **create-index-cursor**

```
(setq ic (create-index-cursor  
          'performance 'event-title))
```

- **next-index-cursor** and **previous-index-cursor** move forward and backward
- You can create a cursor with restrictions (where to start, where to end, etc.)
- **free-index-cursor** free cursor storage when no longer needed.



## More on indexes

- Indexes use resources so only index slots which you will use to access objects.
- We indexed date, event-title, perf-id, but not time, sold, or unsold.
- When retrieving from an index, you can specify specific values (as we do with perf-id) or ranges:

```
(retrieve-from-index-range 'performance 'date "20101001" "20101101")
```

retrieves all events in October, 2010



## Multiple users: what is supported

- Multiple users can access a database server running on the machine where the database is stored.
- Clients can be on any machine that can connect to the server.
- Now, commits may fail because of simultaneous changes!
- **with-transaction-restart** catches the commit failed error.



## Transactions, commit, and rollback again

- You have made changes and call (commit) to store them.
- If the changes do not conflict with the current state of the database, the commit works.
- If there is a conflict, the commit fails with an error.
- Now (rollback) undoes all your changes **and** syncs your local copy of the data with the stored copy.



## Transactions, commit, and rollback again 2

- Suppose two people are selling performance tickets. Seller one calls (sell “jr1101” 5) and commits.
- Now Seller two, without updating, does (sell “jr1101” 6) and tries to commit. That will fail because the local and stored sold and unsold slot values are different.
- (rollback) deletes changes and syncs local and stored.
- Now (sell “jr1101” 6) (commit) *may* work if seller one has not again sold tickets and committed.



## Transactions, commit, and rollback again 3

- Here is the standard way:

```
(with-transaction-restart ()  
  (sell "jr1001" 6)  
  (commit))
```

- This tries to commit and, if necessary, syncs (with (rollback), done automatically) and tries to commit again until success.
- Won't work with very many users and transactions: you need locks or other mechanisms.
- **with-transaction-restart** returns body return value and t (two values) if successful and nil (one value) if not.



## Transactions, commit, and rollback again 4: new sell function

```
(defun sell (perf-id number)
  (let* ((perf (retrieve-from-index
                    'performance 'perf-id perf-id)))
    (with-transaction-restart ()
      (let* ((avail (perf-unsold perf))
             (sold (perf-sold perf)))
        (if (<= number avail)
            (progn
              (setf (perf-unsold perf) (- avail number))
              (setf (perf-sold perf) (+ sold number))
              (commit))
            (error
             "only ~d tickets available, cannot sell ~d"
             avail number))))))
```



## But you really need to design for multiple users

With the ticket selling model, these problems:

- If a lot of sellers, your transactions may never commit.
- As designed, you cannot guarantee you will not oversell tickets, or that you will not promise tickets are available and find them sold out from under you by another sales person,
- Solution: locks might work, but likely better is a ticket-hold slot for the performance class:

```
(defclass performance ()  
  ((date :initarg :date :index :any :accessor perf-date)  
   (time :initarg :time :accessor perf-time)  
   (unsold :initarg :available :initform 100  
           :accessor perf-unsold)  
   (tickets-hold :initform 0 :accessor perf-hold)  
   (sold :initarg :sold :initform 0 :accessor perf-sold)  
   (event-title :initarg :title :index :any  
                :accessor perf-event-title))  
  (:metaclass persistent-class))
```





## commit and rollback review

- Commit stores any current changes to the database. If the local data on your machine conflicts with stored data (because another user committed changes) commit fails.
- Rollback deletes all changes you made since your last commit **and** updates your local data to latest database values.
- **Changes to values in vectors, lists, or structures are not noticed automatically and so you have to mark objects as changed.**



## Some network details

- On server use start-server (database-directory and port are required arguments, password etc. supported).
- On client, open-network-database (server and port are required arguments).



## Other features and details

There are more features than we will mention!

- Sets and maps: a set is (for a large number of objects) better than lists. Maps (of class `ac-map-range`) are a kind of persistent hash table, but should be used sparingly because of resource demands.
- Objects have internal but accessible ids (called *oids*). You can program using them or not as you wish.



## More features and details

- Delete instances (objects) with delete-instance.
- When an instance is deleted locally, accessing it causes an error. (rollback) undeletes. (commit) deletes in stored database.
- **deleted-instance-p** tests whether an object is deleted.



## More features and details 2

- Loading a large amount of data at once can be slow because index btrees can overflow the cache. Using **commit**'s :bulk-load option delays index creation and can be much faster.
- Bulk loading should be used when a lot of data is loaded and other accesses can wait until the bulk load is finished (such as an initial load).



## More features and details 2

- When accessing the database from multiple processes within the same Lisp, only one process can use a database connection at a time, but there can be multiple connections. See the reference manual for details.



## More features and details 3

- There are many ways performance can be optimized, such as, for example, specifying appropriate file and cache sizes.
- Modifying gc parameters is sometimes useful.
- This is discussed in the reference manual.



## More features and details 4

- **close-database** closes the connection to the database.
- There is functionality for closing client connections and other connection functionality detailed in the reference manual.
- 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. See the reference manual.





## Where Acache differs from standard CLOS programming

AllegroCache is designed to make database operations similar to ordinary CLOS programming, so, for example, class changes are easy and standard. But there are some important differences:

- **change-class** does not work. You cannot change the class of a persistent object.
- Modifications of non-CLOS objects (values in vectors or lists, e.g.) are not tracked automatically. See **mark-instance-modified**.



## Finding objects and iteration options

- You can always (of course) iterate through the whole database using **docclass**.
- You can use indexed slots and find objects, specifying ranges to restrict the search.
- You can use expression cursors (made with **create-expression-cursor**) to search for values in multiple slots

(and (:range date "20101101" "20101201")  
(:range time "1800"))

Will find all events in November, 2010 starting at 6:00 pm or later and **next-index-cursor** will loop through what is found.



## Data recovery

- **save-database** stores current state in a file.  
**restore-database** uses that file to open a database (the file may be different from the current state of the database).
- AllegroCache builds a log file recording all transactions. The database can be restored from these files.



## Object editor

The IDE's Object Editor can be used to display the AllegroCache data:

The screenshot shows the Franz IDE's Object Editor window titled "Test". The window contains a form with the following fields and buttons:

- Buttons: First, Previous, Next, Last, Select, Search
- Fields: Title (Joan Rivers), Date (20101101), Time (1900), Available (73), Sold (27)
- Buttons: Save, Revert, Commit, Rollback

Below the form is a table titled "All Instances" with the following data:

	Title	Date	Time	Avail
1	Rolling Stones	20101010	2000	
2	Joan Rivers	20101101	1900	



## Documentation

- <http://www.franz.com/products/allegrocache/docs/>
- Reference manual  
<http://www.franz.com/products/allegrocache/docs/acache.pdf>
- Web page  
<http://www.franz.com/products/allegrocache/>



## AllegroCache: an Introduction

- David Margolies ([dm@franz.com](mailto:dm@franz.com))
- Questions and comments to  
[support@franz.com](mailto:support@franz.com)