AllegroStore manual

version 2.1.8, release with Allegro CL 6.2

June, 2002

Copyright and other notices:

This manual has Franz Inc. document number D-U-00-AST-02-020529-8-F. This is revision 8 of this document.

Contents

Preface

This manual contains definitions of AllegroStore concepts and functionality and information on installing and maintaining an AllegroStore system.

It is in PDF format, intended to be read online using an Adobe Acrobat (r) Reader.

We would appreciate comments, suggestions, and criticisms of this manual. See the onlie HTML files *doc/introduction.htm* for information on contacting Franz Inc.

[This page intentionally left blank.]

Chapter 1 Installation

Installation instructions can be found in the document *doc/installation.htm* provided with the distribution. That document describes the unified installation procedure for Allegro CL and all associated products including AllegroStore.

[This page intentionally left blank.]

Chapter 2 Release notes

2.1 How to use this section

Release Notes for AllegroStore on Allegro CL 6.2 can be found in the document *doc/release-notes.htm* provided with the distribution.

[This page intentionally left blank.]

Chapter 3 Introduction

3.1 Format of the manual

This manual contains definitions of Lisp functions, variables, named constants, special forms, macros, etc. The name of the object being defined appears on the left while the type of object appears in brackets on the right. Arguments (if any) appear on the next line, which is labeled Arguments. Definitions use a larger type font to highlight them. What follows is a template followed by specific examples:

name **[Object type]**

Arguments: *parameters*

- [Description goes here.]

Three specific examples follow:

digit-char-p **[Function]**

Arguments: *char* &optional *radix*

default-pathname-defaults **[Variable]**

setf **[Macro]**

Arguments: *{place newvalue}**

Arguments lists may spill over onto additional lines. If the arguments list is blank, it means that there are no arguments. (Objects such as variables do not have an arguments line, of course.) The line naming the object and the arguments line, if present, are followed by explanatory text and examples. Many optional and keyword arguments have default values which are specified in the explanatory text.

Type faces are used to distinguish between operators (functions, macros, etc.), symbols, constants, printed forms, and examples. Operator names are printed in **bold Courier**. Arguments (and other placeholders) are in *slant Courier*. Other symbols are printed in plain Courier, as are constants (such as #\A and nil) and special symbols (such as *package*) and keywords and lambda-list keywords (such as :test and &optional, respectively). Printed forms and examples are printed in Courier. User input is typically in **bold** while what the system prints is typically in plain.

Examples may not exactly match what is on the screen

Sometimes the printed example has a line break which does not actually appear on the screen. More important, there are slight differences between different versions of Allegro CL, so the printed examples showing Lisp sessions may differ from what you will actually see, especially as regards what the system prints. Suggested code will work with all versions, however, unless otherwise indicated.

On occasion, a page break is placed above the usual bottom of the page so that text and, more commonly, examples will be on a single page. We have tried to ensure that symbol names and other literals are only broken where a hyphen actually appears in the name. Thus **`digit-char-p`** could be broken between **`digit`** and **`char`**, or between **`char`** and **`p`**, but not elsewhere. Because we justify left and right, this restriction can result in odd type spacing in some lines.

A note on the prompt

The prompt for Allegro CL is `USER(N):`, where *N* is the expression number (so USER(1) is the first prompt, USER(2) the second, etc.) Most transcripts in this manual use the simplified prompt `cl:`. This (we hope) avoids confusion about the expression numbers.

3.2 An outline of the manual

This manual contains the following chapters:

1. **Installation**: Complete installation instructions for your platform.
2. **Release notes and technical memoranda**: The newest technical information and clarifications will appear here in between software releases. Keep all incoming memoranda here.
3. **Introduction**: The chapter you are reading.
4. **Technical overview**: A general discussion of object-oriented databases. A technical overview of AllegroStore is included.
5. **Tutorial**: A brief introduction to the workings of AllegroStore. Everyone who is going to use the program should do the tutorial.
6. **Programmer's guide**: This chapter contains specialized programming information and discussion on all parts of the database structure.
7. **Reference guide:** Every AllegroStore macro, function, variable, and other object is discussed in detail here.
8. **Database maintenance and administration**: This is the first of three chapters included as a reference from the Object Design documentation.
9. **Administration utilities**: Describes the utilities used in tuning ObjectStore databases and their directories. Directory Manager database utilities are included.
10. **User utilities**: Describes user-level utilities for creating and manipulating ObjectStore databases and their directories.

There is an index after the last chapter. The last page is an information sheet describing how to contact Franz Inc.

3.3 Comments and suggestions

We are pleased to hear from our users in order to improve AllegroStore. We invite your comments and suggestions. The address to which to write, either by post or by electronic mail, is on the Information sheet at the very end of this manual.

3.4 Reporting bugs

We are committed to the highest standards of software engineering. This release of AllegroStore was tested both internally and in the field. Nevertheless, as with all computer programs, it is possible that you will find bugs or encounter behavior that you do not expect. In that event, we will do our utmost to resolve the problem. But, resolving bugs is a cooperative venture, and we need your help.

What follows here is essentially identical to the bug-reporting instructions for Allegro CL. If you have read section 1.7 of the *Allegro CL User Guide*, also called **Reporting bugs**, you know exactly what to do.

Before reporting a bug or a suspected bug, please study this document, the *Allegro CL User Guide* and *Common Lisp: the Language* (2nd edition!) to be sure that what you experienced is indeed a bug. If the documentation is not clear, this is a bug in the documentation: AllegroStore may not have done what you expected, but it may have done what it was supposed to do.

A report that such and such happened is generally of limited value in determining the cause of a problem. It is very important for us to know what happened before the error occurred: what you typed in, what AllegroStore printed out. A verbatim log may be needed. If you are able to localize the bug and reliably duplicate it with a minimal amount of code, it will greatly expedite repairs.

It is much easier to find a bug that is generated when a single isolated function is applied than a bug that is generated somewhere when an enormous application is loaded. Although we are intimately familiar with AllegroStore, you are familiar with your application and the context in which the bug was observed. Context is also important in determining whether the bug is really in AllegroStore or in something that it depends on, such as the operating system.

Please include the following information in bug and suspected-bug reports to us. Incomplete information is likely to delay or complicate our response.

- **Lisp implementation details**. All the necessary details are provided by evaluating the following forms (which produce an output file *bug-rep-file* -- but you can use any filename, of course) and send us the contents of the file:

```
(excl:dribble-bug "bug-rep-file")
allegrostore:*allegrostore-version*
(dribble)
```

- **Information about you**. Tell us who you are, where you are and how you can be reached (an electronic mail address, a postal address, and your telephone number), your AllegroStore license number, and in whose name the license is held.

- **A description of the bug**. Describe clearly and concisely the behavior that you observe.
- **Exhibits**. Provide us with the *smallest, self-contained* Lisp source fragment that will duplicate the problem, and a log (e.g. produced with **`dribble`** or **`dribble-bug`**) of a *complete* session with AllegroStore that illustrates the bug.

A convenient way of generating at least part of a bug report is to use the **`excl:dribble-bug`** function mentioned above. Typing

```
(excl:dribble-bug "filename")
```

causes implementation and version information to be written to the file specified by *`filename`*, and then records the Lisp session in the same file. Typing

```
(dribble)
```

excl:dribble-bug

will close the file after the bug has been exhibited. **`excl:dribble-bug`** is defined in the online manual page *doc/pages/operators/excl/dribble-bug.htm*.

Note that if whatever you type to duplicate the bug loads in files of yours either directly or indirectly, attach a complete listing of the source version of these files to your session log. The following dialogue provides a rudimentary template for a bug report.

```
USER(5) (dribble-bug "bug.dribble")
USER(6) allegrostore:*allegrostore-version*
;; Now duplicate your bug . . .
USER(7) (dribble)
```

Send bug reports to either the electronic mail or postal address given on the information sheet at the end of this manual. We will investigate the report and inform you of its resolution.

We will meet you more than half way to get your project moving again when a bug stalls you. We only ask that you take a few steps in our direction.

Patches

All complex software products contain bugs and errors. Some bugs can be fixed with a patch, which is a short file containing a correction to a specific problem. Patches have several advantages as a way to fix bugs:

- Because they are short, they can be sent by electronic mail. Electronic mail is preferable to sending a tape (or other physical media) since it is fast, goes directly to the right person, and avoids problems with shippers and receiving departments, to say nothing of customs agents.
- Because they fix a specific problem and do not otherwise change the Lisp, they do not constitute a new release. This allows us to keep track of exactly what version of AllegroStore you have.
- If the patch itself causes a problem (it is rare, but it can happen), it is very easy for you to back it out. We do not have to send you anything, you just rebuild AllegroStore without the patch and you are back where you started.

See the information on getting and installing patches in the HTML file *[Allegro directory]/doc/introduction.htm*.

3.5 Upcoming technical memoranda

We may issue Technical Memoranda between releases of AllegroStore. These are numbered articles documenting changes in the software.

[This page intentionally left blank.]

Chapter 4 Technical overview

This chapter is an overview of the AllegroStore object-oriented database management system (ODBMS) and how it provides a solution to building CLOS based groupware applications.

4.1 Product description - What does AllegroStore do?

AllegroStore provides full-fledged database functionality for CLOS programmers. AllegroStore is a high-performance object-oriented database management system which offers Allegro CL users the power of persistent object storage with fast retrieval and update of object data. AllegroStore provides query processing, transaction-based operations, and permits concurrent access to objects in a client/server environment. AllegroStore also offers standard database features such as deadlock detection, exception handling (integrated into the Lisp condition system), referential integrity, and inverse functions.

AllegroStore combines proven database engine technology from industry leader Object Design with CLOS and a rich set of development tools. The result is a productive development environment for the design and delivery of commercial applications. AllegroStore can handle large-scale database applications without sacrificing performance.

AllegroStore is seamlessly integrated with Allegro CL and CLOS, the Common Lisp Object System. CLOS provides the incremental compilation, automatic memory management, method discrimination and specialization, multiple inheritance and dynamic redefinition.

Note that you must be licensed to use AllegroStore. It is not part of the standard Allegro CL package. You must install ObjectStore (which is provided to AllegroStore customers) for AllegroStore to work.

4.1.1 AllegroStore features

Persistent object access

AllegroStore uses ObjectStore as its database manager. Persistent Lisp objects are described to ObjectStore in a C-structure format which enables ObjectStore to query, access and modify persistent Lisp objects. This allows AllegroStore to use fewer machine resources than would be expected in a standard large Lisp system (where all of the data is present in virtual memory).

ObjectStore itself supports high-performance data manipulation by mapping the parts of the database that contain the objects directly into the Lisp system's virtual address space. Only the objects being directly referenced are mapped in. Other objects (including the objects that are the values of slots of directly referenced objects) are not mapped in until

they too are directly referenced, so even large objects with many slots incur minimal database access overhead.

Using the ObjectStore Virtual Memory Mapping Architecture (VMMA), AllegroStore can make the speed of dereferencing pointers to persistent objects the same as that for transient objects, namely the speed of a single “load” instruction.

Minimal locking overhead

A problem in databases which are simultaneously accessed by many users who are able to modify the data is efficient locking. When one user wishes to make a change to data, other users cannot access that data until the change is completed (for they might otherwise see partially modified and thus inconsistent data). AllegroStore uses an internal algorithm for locking objects, typically locking the page or pages on which data resides. Only a single lock action is needed for an entire page, which provides significant performance gains over other locking mechanisms.

Locking granularity

Why is page locking optimal in most cases? Early designs for object-oriented databases focused on trying to lock each object separately. Although early architects felt that per-object locking would give the user the highest throughput in terms of concurrent access, the reality turned out to be quite the opposite. While object-level locking can in some cases reduce the potential for concurrency conflicts, it does so at the cost of introducing locking overhead on every object. Since object-oriented applications use composite objects built with objects often several levels deep, accessing, locking, and transporting data on a per-object basis turns out to have disastrous consequences for performance. Furthermore, by increasing the length of each transaction, object-level locking actually increases the potential for concurrency conflict in many cases.

Even a simple transaction may touch several thousand such objects and will create unacceptable overhead just doing the lock acquisition. Further, the system must track the status of every locked object in it. This leads to an overhead rate that escalates in a non-linear fashion as the system maintains the “waits for” graph needed for deadlock detection. In addition, object-level locks negatively affect coding productivity, because the developer must explicitly manage all locks, which is difficult and error-prone.

Minimal data caching overhead

Even when many users access a shared database, very often the next user to use a data item will be the same as the previous user. In other words, while concurrent access must be allowed and must work correctly, many data items will be used mainly by one user over a short span of time. AllegroStore uses a caching mechanism that allows a user to re-access previously retrieved objects without experiencing the overhead for refetching or relocking those objects. This *data caching* means that when a sequence of transactions accesses the same objects, there is a high probability that the data accessed in the next transaction will already be cached in workstation memory.

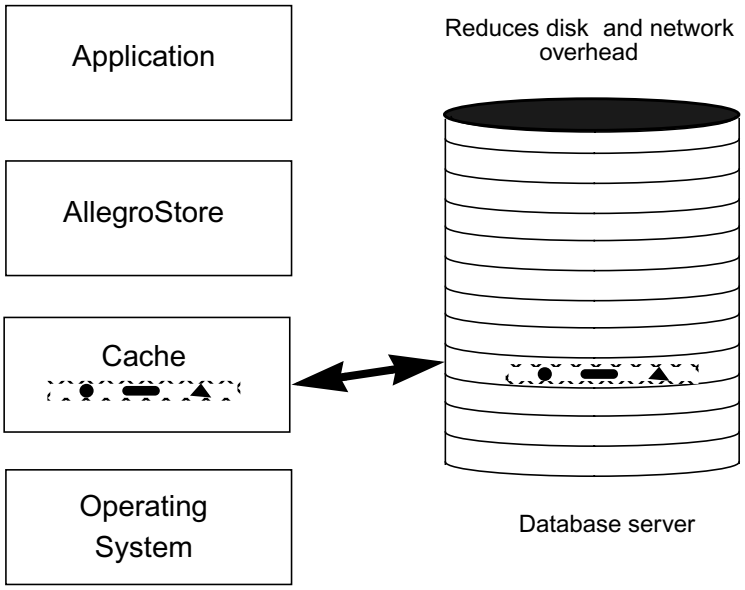


Figure 1. When a client requests a group of objects, clustering and data caching minimize disk and network overhead.

Minimal per-object network overhead

Since network accesses are expensive operations, reducing the number of data transfers is critical to the success of distributed object-oriented database management systems. AllegroStore batches objects to be transmitted between client and server, and typically only one network request is used to send a group of referenced objects into the client cache. This strategy minimizes network traffic and allows the client to proceed through a transaction without contacting the server until a transaction is committed.

Minimal disk access overhead

Often an application will use only a portion of a database, and that portion will be (or can be arranged to be) stored contiguously in a small section of the database. With AllegroStore's automated clustering facility, related objects are clustered together in the database, enhancing locality of reference.

Clustering increases performance (or throughput), since disk access time for contiguous data is faster than random access, because disks can typically read or write many sequential blocks in the time it takes to move the disk head to a random location. AllegroStore's object-level clustering feature allows programmers to increase performance (as well as minimize concurrency conflicts) by grouping only those objects that belong together on a disk page. In future versions, AllegroStore programmers will be able to tune the clustering process.

Transparent lock management

AllegroStore manages the locking of database objects in a way that is both automatic and transparent. Persistent data is just like ordinary heap-allocated (transient) data; once there is a pointer to it, the object can be used in the ordinary way.

AllegroStore automatically takes care of locking, and keeps track of what has been modified. AllegroStore's locking model insures integrity by automatically managing a transaction's read and write sets. Any object that has been updated by a transaction will be written out to stable storage when the transaction commits.

In comparison other object database systems require the programmer to lock each object explicitly, adding a great deal of complexity to the task of application coding. Forgetting to call a lock subroutine can result in loss of data, because the changes will not be flushed to disk. This can corrupt the database, because concurrent accesses will not be serialized. Forgetting to release a lock will cause other programs to wait forever. AllegroStore's automatic and transparent locking protects the integrity of the database against many kinds of programmer error.

4.2 Relationships

What are relationships?

Relationships are useful in modeling complex objects such as designs, parts hierarchies, documents, and multimedia information. Each relationship is composed of two or more objects that are constrained to be consistent with one another in a particular fashion. The constraints on the data members composing the relationship are declared by the user.

Example

In the example below, we illustrate a parts hierarchy in a car. A set of tires are parts of cars. We make a database of tires and then constrain a set of tires to be part of an automobile. We can query over the automobiles and see the tires for each automobile as well. We define the `tire` class with the code below. First two notes.

- **For readers unfamiliar with CLOS**. When evaluated, the following code defines a class of objects named `tire`. It also defines one slot: the `brand`. '`:allocation :persistent`' means the brand will be stored in the permanent database. `:initarg :brand` means the brand can be specified when an instance of a tire is created by specifying an argument labeled `:brand`. '`:metaclass persistent-standard-class`' says that data about tires will be stored in the permanent database.
- **For users familiar with tires**. The brand is of course only one of several necessary pieces of information about a tire (width, rim size, etc. are all also necessary). However, this is a simple example.

```
(defclass tire ()
   ((brand :allocation :persistent :initarg :brand))
   (:metaclass persistent-standard-class))
```

Now we receive two new tires, so we create instances of tires and store them in the database in the file *tires*.

```
(with-database (db "tires")
  (with-transaction ()
    (make-instance 'tire :brand 'michelin)
    (make-instance 'tire :brand 'pirelli)))
```

A query will print these out:

```
cl: (with-database (db "tires")
     (with-transaction ()
      (for-each ((obj tire))
         (format t "Tire brand is ~s~%" (slot-value obj 'brand)))))
Tire brand is michelin
Tire brand is pirelli
nil
cl:
```

Now we define the `auto` class. It contains a `tires` slot where data on the tires on the auto is stored. Thus we have a relationship between the persistent tire class and the persistent auto class.

```
(defclass auto ()
   ((tires :type (set-of tire)
          :allocation :persistent :accessor tires :initarg :tires))
   (:metaclass persistent-standard-class))
```

We create an instance of an auto. We also create five tire instances and associate them with the new auto (the cheap tire is the spare):

```
(with-database (db "cars")
  (with-transaction ()
    (make-instance 'auto :tires (list
                                 (make-instance 'tire :brand 'goodyear)
                                 (make-instance 'tire :brand 'goodyear)
                                 (make-instance 'tire :brand 'goodyear)
                                 (make-instance 'tire :brand 'goodyear)
                                 (make-instance 'tire :brand 'cheap)))))
```

We can execute a query to print all tires currently on all automobiles in the database. (Of course, so far we only have one auto in the database, and five tires: The michelin and pirelli tires defined above are in another database.)

```
cl: (with-database (db "cars")
     (with-transaction ()
       (for-each ((c auto) (w (tires c)))
         (print w))))


#<tire in /db/mjm/cars p: #x72f0004  @ #xad479a>
#<tire in /db/mjm/cars p: #x72f001c  @ #xad536a>
#<tire in /db/mjm/cars p: #x72f0034  @ #xad57f2>
#<tire in /db/mjm/cars p: #x72f004c  @ #xad5a2a>
#<tire in /db/mjm/cars p: #x72f0064  @ #xad5c62>
nil
```

And, just as above, we can list all tires in the database, even though they were defined as slots of an instance of an auto:

```
cl: (with-database (db "cars")
     (with-transaction ()
       (for-each ((obj tire))
         (format t "Tire brand is ~s~%" (slot-value obj 'brand)))))
Tire brand is goodyear
Tire brand is goodyear
Tire brand is goodyear
Tire brand is goodyear
```

```
Tire brand is cheap
nil
cl:
```

Referential integrity

AllegroStore provides *referential integrity* which is automatically enforced as an update dependency. When an object is deleted, all objects which have slots that point to the deleted objects are updated.

To see this, let us expand the definition of a tire to include a barcode slot. Note we use the `:inverse` option to define the inverse function **`code-to-tire`**, which tells what tire has a specific barcode. See chapter 5 **Tutorial** and section 6.9 **Inverse functions** for more information on inverse functions. Also, the redefinition is in a **`with-database`**, ensuring the definition of the tire class in the database is the same as the one in memory:

```
(with-database (db "cars")
  (defclass tire ()
    ((brand :allocation :persistent :initarg :brand)
     (barcode :allocation :persistent :initarg :barcode
              :inverse code-to-tire))
    (:metaclass persistent-standard-class)))
```

Now we assign a barcode to each tire in the database:

```
cl: (with-database (db "cars")
      (with-transaction ()
       (let ((bc 1))
         (for-each ((obj tire))
           (setf (slot-value obj 'barcode) bc)
           (setf bc (1+ bc))
           (format t "Tire with barcode ~d has brand ~s~%"
                     (slot-value obj 'barcode)
                     (slot-value obj 'brand))))))
Tire with barcode 1 has brand goodyear
Tire with barcode 2 has brand goodyear
Tire with barcode 3 has brand goodyear
Tire with barcode 4 has brand goodyear
Tire with barcode 5 has brand cheap
nil
```

Now, assume a driver of the single auto in our database has a puncture. While putting on the cheap spare, the bad tire rolls down the hill, into the river, and floats away. The driver tells us it is the tire with barcode 3. We first make sure there is such a tire by using the inverse function **`code-to-tire`**:

```
cl: (with-database (db "cars")
      (with-transaction ()
           (code-to-tire 3)))
(#<tire in /db/mjm/cars (closed-database) @ #xcedfba>)
```

We now delete the instance of that tire, and then look at the tires associated with our auto (since **`code-to-tire`** returns a list of one tire, we use **`car`** to extract the tire from the list):

```
cl: (with-database (db "cars")
      (with-transaction ()
         (delete-instance (car (code-to-tire 3)))))
#<tire in /db/mjm/cars (closed-database) @ #xd0400a>
```

Now we look at the tires associated with our auto, and find there are only four. The deleted one is gone:

```
cl: (with-database (db "cars")
      (with-transaction ()
        (for-each ((c auto) (w (tires c)))
          (print w))))
#<tire in /db/mjm/cars p: #x72f0004  @ #xd6c342>
#<tire in /db/mjm/cars p: #x72f001c  @ #xd6d852>
#<tire in /db/mjm/cars p: #x72f004c  @ #xd6da8a>
#<tire in /db/mjm/cars p: #x72f0064  @ #xd6dcc2>
nil
cl:
```

Garbage collection

Persistent objects are of two types: persistent CLOS instances and other Lisp objects.

AllegroStore uses a simple reference counting garbage collector, which imposes some limitations on how objects are stored in the persistent database.

In typical CLOS programming, circular references between CLOS instances (either directly through a chain of CLOS instances, or indirectly through Lisp objects such as hash tables) are common. AllegroStore permits such circular references since persistent CLOS instances are always reachable by querying over the classes of the instances.

If a persistent class is explicitly deleted, all instances of that class are also automatically deleted and immediately garbage collected. Any other persistent objects pointing to the deleted CLOS instances are automatically updated using referential integrity.

When the slot of a persistent CLOS instance is set to a regular Lisp object, a copy of that object is stored in the database. Thus, every regular Lisp object in the database has a reference count of exactly 1. If the slot of the persistent CLOS instance is set to another object, then this Lisp object is deleted.

Currently we enforce the limitation that the Lisp object must contain no circularities. This limitation is present because the code to delete objects or to scan objects to keep referential integrity assumes no circularities. This limitation may be removed in a later release. AllegroStore automatically detects any attempt to store circular objects and signals an error.

4.3 Distributed object management

Client/server architecture

AllegroStore supports cooperative access through its flexible client/server architecture which gives the best use of the computational power of the client workstation. AllegroStore's client/server implementation means that:

- servers can support many client workstations,
- workstations can simultaneously access multiple databases on many servers, and
- a server and client can be co-resident on the same machine.

The server and the client communicate via a network when they are running on different hosts, and by operating system facilities such as shared memory, local sockets, etc., when they are running on the same host.

4.3.1 The Server

What is a server?

In multi-user configurations, computers dedicated to running server processes are referred to as servers. In the AllegroStore environment, the server process manages physical data on disk and arbitrates among client processes making requests for the data. The server process maintains locking tables, provides deadlock detection, checkpointing, buffer management, and license management, and handles logging.

Restart/recovery

Recovery is based on a log, using a write-ahead log protocol. Transactions involving more than one server are coordinated using the two-phase commit protocol. The server also provides backup to long-term storage media such as tapes, allowing full backups as well as continuous archive logging. When there is contention for an object, the server overrides the default client behavior of encaching locks for future use, and calls back the lock so that the competing transactions can proceed.

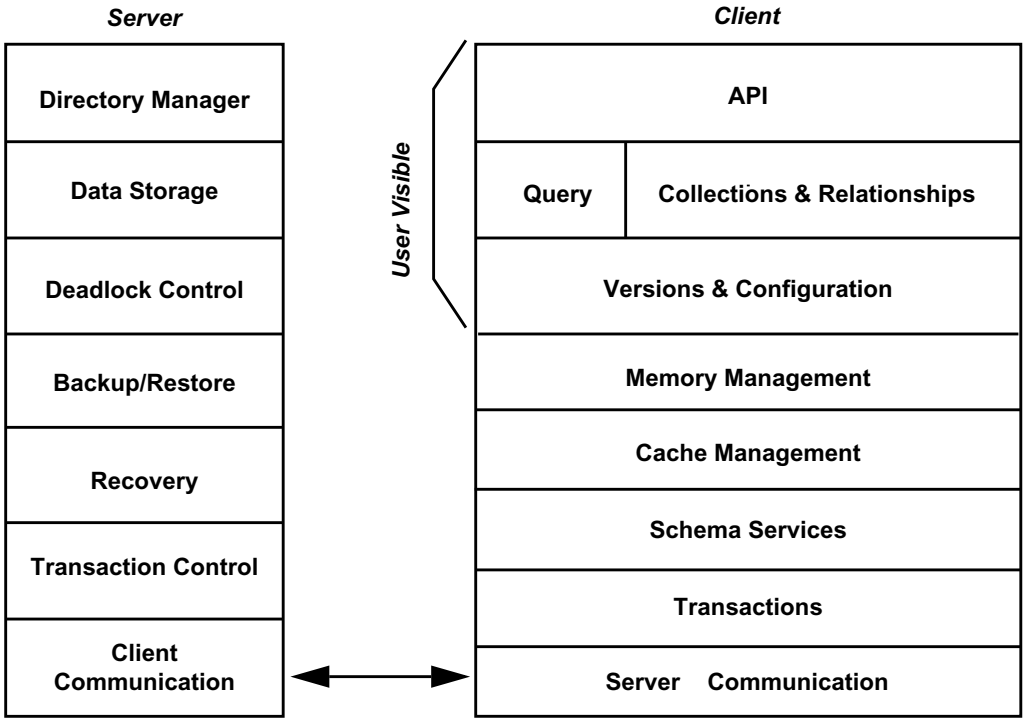


Figure 2. AllegroStore provides a powerful distributed client/server architecture for distributed object computing.

4.3.2 The client

What is a client?

Systems that run application processes to access the servers over the network are referred to as clients. AllegroStore uses the ObjectStore client libraries to provide the interface between the user's application and the database server. These automatically manage the logical view of the data including sets, queries, versions, transaction management, memory management, and relationships among objects.

Much of the query and DBMS processing occurs on the client side of the network. This contrasts sharply with traditional relational DBMS systems in which the server is largely responsible for handling all query processing, optimization, and formatting. When each client does its own work, the aggregate computing power of the network is used, and the server process can be on the same machine as the client processes. This allows more flexibility in configuration: any potential client machine is also a potential server.

The client environment includes the AllegroStore Cache Manager, which maintains a cache of objects accessed by all the client processes on a machine. In single-system configurations both the client processes and the server process exist on the same computer.

4.4 Concurrency control in a client/server environment

What is concurrency control?

Concurrent access by multiple clients enabled by AllegroStore's client/server architecture creates the potential for one set of updates to interfere with another. The prevention of this interference is called *concurrency control*. AllegroStore handles simultaneous access to objects with transactions. A transaction in this case means a unit of work that is handled by the application and determined by the user. The act of reading or writing to the database must be within a transaction.

The concurrency control scheme employed for conventional transactions is based on a serializable transaction management approach.

Client/server locking model

AllegroStore's locking model ensures integrity by automatically managing a transaction's write set. Any object that has been updated by a transaction will be written out to stable storage when the transaction commits.

This contrasts with more primitive approaches that require the program to mark which objects have changed by calling a subroutine for each such object. Neglecting to call this subroutine can result in loss of data since the changes will not be flushed to disk. AllegroStore locking, by contrast, is both automatic and transparent.

Lock management

Locking information is cached on both the client and server, to minimize the need for network communication when the same process performs consecutive transactions on the same data. A copy of an object in a client cache is marked as either shared mode or exclusive mode. The server keeps track of which objects are in the caches of which clients, and with which modes. The client holding the data in a cache is said to hold a read lock if the data is in shared mode and a write lock if the data is in exclusive mode. (‘Read’ and ‘write’ refer to what the client holding the lock is doing.)

When a client requests an object from the server and the server notices that the object is in the cache of some other client, the server will check to see if the modes conflict. If they do, the server sends a message to the client holding the lock, asking it to remove the object from its cache. This is called a “callback” message, since it goes in the opposite direction from the usual request.

When the client holding the lock receives the callback, it checks to see if the lock is currently in use, and if not, relinquishes the object immediately, and removes the copy of the object from its cache. If the object is in use, the client replies negatively to the server, and the server forces the requesting client to wait until the holder is finished with the transaction. When the holding client commits or rolls back, it then removes the copy of the object from its cache, and the server can allow the waiting client to proceed.

Example

In Figure 3 below, objects are locked for read and write on client demand. At the end of the transaction, objects are unlocked and modified objects are written to the server log. Another client's attempt to write to a locked object is blocked until the lock is released. In the exam-

ple, if Client 1 has a object is required by Client 2's transaction, the server sends a callback notice to Client 1 using the shared object. Client 2 will receive the object as soon as Client 1's transaction is through.

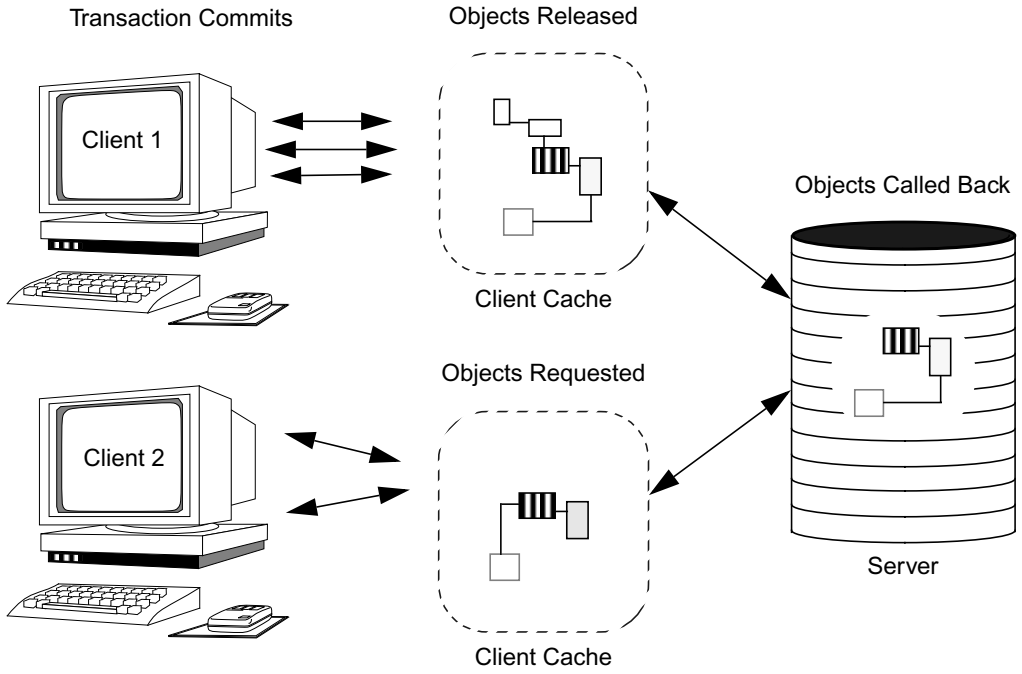


Figure 3. Data integrity is guaranteed in distributed environments

Deadlock detection

On rare occasions, the system may roll back a transaction either because of a network failure or because it has determined that the transaction is involved in a deadlock. By default, the transaction is restarted at the first statement following the start of the transaction. The transaction will continue to be retried until the transaction is successful or the maximum number of tries has occurred.

4.5 Heterogeneous operation

What is heterogeneity?

Heterogeneous operation means that

- data are stored in the database using a portable format that does not depend on any particular architecture's data format, alignment requirements, byte ordering, or floating-point representation;
- the client or server do not need to run on platforms of the same type;
- different clients can access a database simultaneously;
- a database created on one platform can be moved to a different server on a different heterogeneous platform.

AllegroStore is for the most part heterogeneous. All 32-bit platforms perform heterogeneous operation with other 32-bit platforms. However, heterogeneous operation is not supported between 32-bit and 64-bit platforms because doing so would have compromised the integrity of certain Lisp datatypes.

4.6 An administrator's view of AllegroStore

AllegroStore uses the basic set of ObjectStore database management facilities.

What is a database?

When an application allocates a persistent object, it specifies a database to contain that storage. An AllegroStore “database” is simply a named area on the disk that serves as a container for objects. These named areas may be simply files, or for certain high speed applications, areas on the disk managed by the ObjectStore server. An application designer may choose to store all the objects for an application in a single database or in multiple small databases. Unlike many other database systems, there is no requirement that the user predetermine the size of the database.

Choice of native or ObjectStore file system

ObjectStore provides the user with two ways to store databases physically. The fastest performance comes when the ObjectStore server manages a raw disk partition. Alternatively, databases may be stored as host operating system files. This support for native file systems lets users treat databases as any other files so that users may employ standard operating system utilities for organizing and managing AllegroStore databases. There are many situations where this is convenient. For instance, file databases can be accessed from machines that can access the file using normal file protocols. Many users will prefer to store AllegroStore databases in working directories along with normal files.

Database management utilities

ObjectStore database management utilities are designed to look very much like the system utilities of the host operating system. Object Store utilities on Unix looks much like Unix system utilities. The ObjectStore database utilities manage:

- Directories (initialize the file system, lists, directory contents, create new directories, move directories)
- Databases (report size, report content, delete, copy, verify the validity of the contents, backup and restore)

Performance monitoring

ObjectStore provides performance statistics that system administrators can use to tune applications and systems. Available utilities provide information on both client and server processes such as:

CPU usage,

page traffic,

block I/O operations,

message traffic,

logging activity,

transaction activity, and

buffer activity.

In addition, a utility is provided to monitor each client process connected to a particular server. Information is reported on the active transactions and the number of blocks locked on behalf of the client process.

Restart/recovery

ObjectStore takes responsibility for restart/recovery and guarantees data integrity. A transaction is an atomic unit of work that is applied to the database. One of the advantages of transaction-based data management is that you can decide not to apply changes made during your transaction. You can undo your changes back to the beginning of the current transaction by throwing out of the **`with-transaction`** body. When this is done, persistent data is rolled back to its initial state at beginning of the transaction, all locks are released and the other processes are allowed to access the pages that the aborted transaction had locked.

In the case of a disruption to the system, ObjectStore insures recovery via a logging mechanism. All after-images of data (redo records) updated by the client are held in a redo log on the server. The redo log is used as a staging area on the server for changes to the database. It is structured so that writes to the log always write contiguous sections of the disk. This is important because any transaction that writes data must wait until the server has insured that the changes are represented persistently on the server disk. Writing contiguous extents to the log provides the fastest possible response from the server during a transaction commit.

Access control

ObjectStore provides the developer with tools to specify the appropriate level of access control for directories and for each database within a directory. Access control within ObjectStore is provided in a manner which is analogous to the operating system file access control system. Each database is automatically accessible to its creator.

ObjectStore provides utilities to assign users to groups that are assigned permissions. The permissions assigned are either read only, write, or execute. User identification and groups are protected by the operating system's password protection mechanism.

On operating systems that provide advanced security feature, ObjectStore uses the feature to provide greater security for databases. Because ObjectStore has a client/server architecture, each ObjectStore server handles requests from many different users and must take responsibility for enforcing access control to files containing databases.

For instance, on Sun systems the ObjectStore server has four optional modes to control read/write access for databases:

1. Authentication required: NONE
2. Authentication required: SYS
3. Authentication required: DES
4. Authentication required: UNIX login

In future releases, enhancements will include support for the Kerberos protocol which is expected to be widely available on all major operating systems.

4.7 Continuous operation

Support for mission-critical applications requires a set of features that guarantee high availability of object stored data. These enhancements make it possible to run ObjectStore applications in a non-stop fashion, so a continuous stream of transactions can be issued against a database twenty-four hours a day.

These enhancements include non-stop database backup facilities, and non-stop archive logging facilities.

Non-stop database backup

Non-stop database backup, also know as “fuzzy backup”, is used for backing up databases or file systems. (Note the term “data item” as used below, will mean the data being backed up, whether it is a database, or a complete database system). There are three main components in the fuzzy backup:

1. The temporary log
2. The backup medium
3. The parallel log

The temporary log is the place where redo records are written as part of recovery processing so that in the case of a system failure it is only necessary to replay this log against the material database. The temporary log is periodically flushed (known as checkpointing) by propagating redo records to the material database. The backup medium is the output device onto which the backup is written. It may be a random access device, e.g. a disk, or a sequential device, e.g. a tape. The parallel log is a second copy of the temporary log that is only written to during backup processing and, unlike the temporary log, is not garbage collected.

Fuzzy backup is conceptually simple. At the start of the backup, the parallel log is enabled so that all subsequent writes to the data item are written to the temporary log (as always) as well as the parallel log. Once enabled, the data item is copied to the backup medium, creating the fuzzy copy of the data. It is called fuzzy because processes are allowed to update the database while the backup is taking place. This is possible because the parallel log captures the results of all the update transactions, which are appended to the backup medium. To restore a fuzzy backup, the data item is restored and then the redo records that were in the parallel log are applied to the restored data item.

Archive logging protects against media failure

Although the fuzzy backup system provides increased protection against media failure, the recovered data is only as recent as the most recent backup copy. Archive logging increases reliability by writing a duplexed copy of the log on media with independent failure mode. Should there be a system failure, the temporary log is used for recovery purposes. For media failure the archive log is used for recovery.

The archive is also conceptually simple: all redo records are written to the temporary log and the (duplexed) archive log. In the event of media failure, the latest backup copy of the database is restored (by restored all full backups and subsequent incremental backups), and the archive log is replayed against the data to make it transaction-consistent.

[This page intentionally left blank.]

Chapter 5 Tutorial

This chapter provides an introduction to using AllegroStore. Many features are demonstrated, but often without a great deal of detail. This chapter is worth scanning by any user, since it provides some programming tips as well as examples.

5.1 Getting started

Put the databases on the server machine

Although it is possible to have a database file on a machine different from the machine running the ObjectStore server, doing so is somewhat involved (see Technical Memorandum # AS-2). Use database files on the same machine as the server runs for purposes of this tutorial.

Use the allegrostore package

Functions, macros, and classes and so on in AllegroStore are named by symbols in the `allegrostore` package. Programs using AllegroStore should work in a package which uses the `allegrostore` package. If you are running AllegroStore interactively, evaluate

```
(use-package :allegrostore)
```

In this manual we have assumed that form has been evaluated, so we do not qualify `allegrostore` symbols in examples. (However, we often use the qualifier when a symbol is first mentioned in the text.)

The nickname for the `allegrostore` package is `astore`.

Assumed background

AllegroStore is derived from the Common Lisp Object System (CLOS). CLOS is an object-oriented programming language and an extension of Common Lisp. We assume that you already understand how to use CLOS. In section 6.2, several introductory books on CLOS are listed.

The code samples shown here are provided in a separate file included in the distribution, in case you want to use them to follow along. The file is called *tutorial.cl*.

Further things before you start the tutorial

Before you begin running the tutorial you should verify that your shell environment variables are set up correctly. There are three steps to the process.

- **The environment variable `OS_ROOTDIR` should point to the directory where you installed Objectstore.** Here is how to verify that the variable is set from the Unix shell:

```
% echo $OS_ROOTDIR
```

To verify that the variable is set from the DOS shell:

```
c:\> echo %OS_ROOTDIR%
```

- **Your** AS_CONFIG_PATH **variable should point to the directory where the Allegrostore configuration database is stored** (this directory is usually the Allegro CL installation directory).

  To test whether the AS_CONFIG_PATH file has been set from a Unix shell

  ```
  % echo $AS_CONFIG_PATH
  ```

  To test it from a DOS shell:

  ```
  c:\> echo %AS_CONFIG_PATH%
  ```

- **Start a Lisp that includes the AllegroStore code**. We assume you are running in a package that uses the `allegrostore` package during the tutorial. To verify, evaluate `(use-package :allegrostore)` in the Lisp that you have started (it is not an error to evaluate this form when the `allegrostore` package is already used but an error will be signaled if the AllegroStore code has not been loaded and, therefore, the `allegrostore` package does not exist).

If you haven't been able to verify any one of these steps: see chapter 1 **Installation guide**. If you have verified all three steps, you are now ready to proceed with the tutorial.

with-database and with-transaction

It is very important that when you open a database, you later close it, and also important that when you start a transaction, you complete it. As we describe in many places, **`with-database`** opens its argument database, evaluates its argument forms and closes the database when complete. And **`with-transaction`** starts a transaction, evaluates its argument forms, and then ends the transactions.

Therefore, if you use those two macros, you will never leave a transaction unfinished or a database unclosed. That is why all of our examples have code wrapped in **`with-database`** and **`with-transaction`**.

However, you may find that you want to open a database and keep it open for a while, and to start a transaction and do things interactively for a while before completing the transactions. See section 6.17 **Interactive transactions** for information on how to do these things. We recommend that users who are just starting out use **`with-database`** and **`with-transaction`**, as done in the code samples.

Our running example

We set up a database for a public library in the running example in this chapter.

5.2 The database

Creating a database

We will be using the data storage system of a Public Library as our example database. Suppose we have just started a library and we only have three books. Here is how we would create a database to describe those three books.

Define the book class

We define the book class. We give it slots for author and title. As we describe below, book is a persistent class (meaning instances will be stored in the database) and its slots are persistent slots (meaning their value will be stored in the database).

```
(defclass book ()
  ((title :allocation :persistent
          :initarg :title
          :accessor title)
   (author :allocation :persistent
           :initarg :author
           :accessor author))
  (:metaclass persistent-standard-class))
```

Note two things in the **`defclass`** form:

- the `:metaclass` argument
- the `:allocation` argument of `:persistent`.

'`:metaclass persistent-standard-class`' tells Lisp that all objects of class `book` will be *persistent*. Persistent objects will be stored in a database.

Specifying '`:allocation :persistent`' in the slot definitions tells Lisp that the value of this slot will also be stored in the database. AllegroStore permits the user to specify on a slot-by-slot basis whether the slot's value should be stored persistently.

Create a database and put some stuff in it

Now we create out database and we create three instances of books which we store in the database.

```
(with-database (db "mylib.db" :if-exists :supersede)
  (with-transaction ()
     (make-instance 'book :title "Dandelion Wine" :author "Ray Bradbury")
     (make-instance 'book :title "Marcovaldo" :author "Italo Calvino")
     (make-instance 'book :title "Baidarka" :author "George Dyson")))
```

Note the following:

- The **`allegrostore:with-database`** macro is similar in syntax to Lisp's **`with-open-file`** macro. **`with-database`** opens, or if necessary creates, a database and automatically closes it when control leaves the body of the **`with-database`** form. The '`:if-exists :supersede`' argument tells the system to create a new database file, deleting an existing file of that name if there is one. We do this at the beginning of a tutorial so you can start a fresh example. Of course, you do not add the argument when opening an existing database whose data you want to examine!
- As we describe in more detail below, the definition of the *book* class is automatically stored in the database when instances of `book` are stored (notice we did not have a database open when we defined the class above). That means another user can open *mylib.db* without having to define the `book` class first.

Transactions

Transactions

The **make-instance** forms are wrapped with the macro **allegrostore:with-transaction**. A *transaction* is a sequence of database operations. At the end of a transaction the program either *commits* the transaction or *rolls it back*. If the transaction is committed, then all of the changes made to the database are visible to other programs using the database. If the transaction is rolled back, then all changes are undone.

The **with-transaction** macro starts a transaction and then evaluates the forms in the macro's body. If control falls through the body of the macro, then the transaction is committed. If control leaves the body of the macro before it has completed evaluation (e.g., as a result of a of **throw** or **go**), then the transaction is rolled back. It is unlikely that anything will prevent the three **make-instance** calls from completing in the example above, and thus this transaction will commit.

> **Important**: transactions may be automatically *retried* (run multiple times) before committing. Code within a transaction should be free of side-effects, except for changes to the database. See the section 6.5 **Transactions**, particularly the information under the heading **Transaction restarts**.

Database operations can only be done if a database is open and a transaction is active. If we had attempted `(make-instance 'book)` without a database being open and the surrounding **with-transaction**, the system would have signaled an error.

For more information on multiple client situations and transaction rollbacks, or for information on transactions finishing and retrying: see section 6.5 **Transactions**.

Displaying and verifying the contents of the library

Displaying the contents of the database

Next, we'll verify that we have actually stored something in the database. Here is a function that will print a list of all the books in a library.

```
(defun print-books-in-library (name)
  (format t "Books in library ~a~%" name)
  (with-database (db name :if-does-not-exist :error)
    (with-transaction ()
      (for-each ((b book))
         (format t "Title: ~a   Author: ~a~%" (title b) (author b))))))
```

We use it to verify the contents of the database:

```
cl: (print-books-in-library "mylib.db")
Books in library mylib.db
Title: Dandelion Wine   Author: Ray Bradbury
Title: Marcovaldo   Author: Italo Calvino
Title: Baidarka   Author: George Dyson
t
cl:
```

print-books-in-library calls **allegrostore:for-each** to scan the database for all objects of a given class. AllegroStore keeps track of all objects of each class in the database. Unlike normal Lisp objects, a persistent object does not disappear due to garbage collection when there are no more pointers to it. In order to delete a persistent object, you must call **delete-instance** on it (we'll have an example of that later).

Now, suppose you exit the Lisp process which you used to build the database and then start Lisp again. Because the database isn't open in the restarted Lisp, the `book` class is not

defined. So what happens if you define **print-books-in-library** and evaluate (print-books-in-library "mylib.db")? It will open the database and print the list of books just as it did the first time.

That is because the database contains not only instances of persistent classes, but also persistent class definitions. When the *mylib.db* database is opened, AllegroStore notices that it contains book objects but the book class isn't defined in the current Lisp, so it defines book according to the definition stored in the database. As a result, the accessor functions **author** and **title** are also defined. If book were already defined in Lisp, but in a different way, AllegroStore would resolve the differences as described in **Changing the schema** just below.

Changing the schema

Changing the schema

The set of class definitions stored in the database is called the *schema* of the database. CLOS makes changing the schema of the database very easy. You need only redefine a class using **defclass**. CLOS determines what has changed in the old definition and updates old objects to obey the new class definition. AllegroStore applies the CLOS model of schema changes and automatic object updating to persistent objects.

Suppose we want to attach a barcode to each book in our library, and to add that information to the book instances already in the database. We can redefine the book class by adding a barcode slot, as follows:

```
(defclass book ()
  ((title :allocation :persistent
          :initarg :title
          :accessor title)
   (author :allocation :persistent
           :initarg :author
           :accessor author)
   (barcode :allocation :persistent
            :initarg :barcode
            :accessor barcode))
  (:metaclass persistent-standard-class))
```

> **Two definitions of book**. The alert reader may have noticed that we have redefined the book class in memory (i.e. in Lisp) but have not (yet) changed the definition in the database. Therefore, memory and the database are inconsistent. We could have wrapped the **defclass** form in a **with-database**, and that would have kept the definitions in sync. Or, having created the inconsistency, we can resolve it (as we do) by specifying the argument :use :memory when we next access the database (that means update the database definition with the memory definition). If we do neither of these things, we will get a continuable error when we access the database, and among the restarts will be one to store the memory definition in the database.

Suppose we bought a set of bar code stickers which begin at number 10001. Here is a program that goes through the whole database and assigns each book a bar-code-number.

```
(defparameter *next-bar-code-number* 10001)

(defun assign-barcodes (library-name)
  (format t "Assigning barcodes to new books in ~a~%" library-name)
  (with-database (db library-name :if-does-not-exist :error :use :memory)
    (with-transaction ()
      (for-each ((b book))
```

```
                (cond ((not (slot-boundp b 'barcode))
                       (setf (barcode b) *next-bar-code-number*)
                       (format t "Book ~a by ~a assigned barcode ~s~%"
                               (title b)
                               (author b)
                               (barcode b))
                       (incf *next-bar-code-number*)))))
```

We'll run it by evaluating `(assign-barcodes "mylib.db")`. The output will look like:

```
cl: (assign-barcodes "mylib.db")
Assigning narcodes to new books in mylib.db
Warning: class BOOK in memory doesn't match definition in database
     [rest of warning message deleted]
Book Dandelion Wine by Ray Bradbury assigned barcode 10001
Book Marcovaldo by Italo Calvino addigned barcode 10002
Book Baidarka by George Dyson assigned barcode 10003
t
cl:
```

As we pointed out in the indented note above, we've added the arguments `:use :memory` to the first argument to **`with-database`**. AllegroStore notices that the definition of `book` in the database differs from the one in Lisp's memory when we open the database. Specifically, the definition of `book` in memory contains the `barcode` slot, and the definition in the database does not. Since the `:use :memory` argument was specified, AllegroStore signals a warning but does not stop execution. If that argument was not supplied, a condition would be signaled asking for a choice between the memory and database versions.

By adding `:use :memory` to the **`with-database`** form, we tell AllegroStore that it should resolve the conflict by choosing the memory definition as the correct one. After the resolution is done, the database definition of `book` has been modified to match the memory definition.

When **`assign-barcodes`** is run, it looks at each `book` object in the database. As each `book` object is accessed, it is updated to include the `barcode` slot. Since we didn't specify an `:initform` for the `barcode` slot, the slot is initially unbound.

Object interrelations

Object interrelations

You can store information in an object database by having objects point to other objects. For example, we can add a class called `patron` to our database to represent the people who will borrow books from our library. We can represent the fact that a `patron` is borrowing a `book` by having one of the slots in the `patron` object point to the set of `books` that the `patron` has borrowed.

```
(defclass patron ()
  ((name :allocation :persistent
         :initarg :name
         :reader patron-name)
   (borrows :allocation :persistent
            :type book
            :set t
            :accessor borrows
            :inverse borrower))
  (:metaclass persistent-standard-class))
```

The `patron` **`defclass`** introduces two new slot options, `:set` and `:inverse`. The option `:set t` states that this slot will contain an unordered collection of Lisp objects. Since in this case we also specify `:type book`, we are stating that this slot will always contain a set of pointers to `book` objects.

We'll explain `:inverse` under the heading **Inverse functions** below.

Let's add a few `patron`s to the library:

```
(with-database (db "mylib.db")
   (with-transaction ()
        (make-instance 'patron :name "Bob Smith")
        (make-instance 'patron :name "Lynda Jones")))
```

Retrieving items from the database

Retrieving specific stored instances of a class

To make it easy to check out `book`s by title, we'll write a function which will return an instance of book once given the title and author. We should have three different books in our library at this point, but we'll add some error-checking just to be safe.

Note that this function must, of course, be run within a **`with-transaction`** form.

```
(defun get-book (given-title given-author)
   (let ((books (retrieve 'book
                          :where '((title equal ,given-title)
                                   (author equal ,given-author)))))
     (cond ((null books)
            (error "This book isn't in the library: ~a by ~a"
               given-title given-author))
           ((> (length books) 1)
            (error "There is more than one book: ~s by ~s"
            given-title given-author))
           (t (first books)))))
```

Note the **`allegrostore:retrieve`** query function. Query functions access or retrieve all objects of the given type which satisfy a certain conditions (see section 6.10 Queries and interators for more information). The conditions are expressed in the `:where` argument and are written in a form we call a *where-clause-list*.

A where-clause-list is a list of *where-clauses*. Each where-clause is a list of form:

(*accessor-name supplied-predicate supplied-value*)

which directs AllegroStore to call the *`accessor-name`* function on each object. The value returned by *`accessor-name`* is tested against *`supplied-value`* by the *`supplied-predicate`*. (In the first where clause in our **`get-book`** function, *`accessor-name`* is **`title`**, which recall accesses the title slot of a book object; *`supplied-predicate`* is the Lisp function **`equal`**; and *`supplied-value`* is the value of the *`given-title`* argument -- that is the effect of the preceding comma.)

If the *`supplied-predicate`* returns true, then the where-clause is satisfied. A where-clause-list is satisfied if all where-clauses are satisfied.

> **Odd order**. Note that the contents of where-clauses are ordered in non-Lisp fashion (the predicate goes in the middle rather than at the beginning). This order is used for historic reasons, but it does serve to emphasize that a where clause is not a standard Lisp form but a list with a special interpretation.

Changing one of the attributes of an existing instance

Now we can write a function that checks a book out to one of our patrons.

```
(defun check-out (library-name patron-name title author)
  (with-database (db library-name)
    (with-transaction ()
      (let ((patron-object
             (first (retrieve 'patron
                              :where '((patron-name equal
                                                    ,patron-name))))))
        (cond ((null patron-object)
               (error "No such patron: ~s" patron-name))
              (t (push (get-book title author) (borrows patron-object))
                 (format t "~s has borrowed ~s by ~s~%" patron-name
                         title author)))))))
```

Adding a book to the list of books borrowed by a patron means **push**ing it onto the `borrows` list. `borrows` is a list of pointers to other books the patron has already borrowed. Note, by the way, that **`check-out`** does not ensure that the book is not already checked out. This fits with a library model since a patron typically has the book in hand at the check-out desk.

The `borrows` slot is a *set slot* (because `:set t` was specified when we defined the slot in the **`defclass`** form), which means that it always contains a collection of zero or more objects. Lisp expects a list as the value of a set slot (the **`push`** macro adds elements to a list to create a new list). If an `:initform` is not specified, then a set slot is initially bound to `nil`, the empty list.

Let's check out a book:

```
(check-out "mylib.db" "Lynda Jones" "Baidarka" "George Dyson")
```

And the output should look like:

```
cl: (check-out "mylib.db" "Lynda Jones" "Baidarka" "George Dyson")
"Lynda Jones" has borrowed "Baidarka" by "George Dyson"
nil
cl:
```

Searching the database for an instance in a slot

Searching the database and retrieving an instance

Now suppose someone says, “I can't find *Baidarka* by George Dyson. Who borrowed it from the Library?” Looking at the that book in our database does not tell us who has it since there is no slot in a book object to indicate its location. One way to find out is to review all the patrons and see who has that book in their borrows list.

```
(defun who-has-1 (library-name title author)
  (with-database (db library-name)
    (with-transaction ()
      (let ((book (get-book title author)))
        (for-each ((p patron))
           (cond ((member book (borrows p) :test #'eqo)
                  (format t "~s has it~%" (patron-name p))
                  (return-from who-has-1 nil))))
        (format t "No one has that book checked out~%")))))
```

We use the function **allegrostore:eqo** to compare two objects for equality. **eqo** is **eq** for **o**bjects (why it should be used rather than **eq** is explained in section 6.12 **Implicit object creation**). Then we can do the query, and the output would be:

```
cl: (who-has-1 "mylib.db" "Baidarka" "George Dyson")
"Lynda Jones" has it
nil
cl:
```

Inverse functions

The **who-has-1** function can take a long time to run, especially when there are a large number of patrons, since it has to examine every `patron` object. It turns out that there is already a faster function that will, given a book object, return the name of the patron borrowing it. The name of this function is **borrower**, and we already defined it inside of the definition of `patron` class. Here it is again:


```
(defclass patron ()
  ((name :allocation :persistent
         :initarg :name
         :reader patron-name)
   (borrows :allocation :persistent
            :type book
            :set t
            :accessor borrows
            :inverse borrower))
  (:metaclass persistent-standard-class))
```

The :inverse slot definition

The `:inverse` slot definition argument tells AllegroStore to create a function which does the inverse of the `:accessor` or `:reader` function. Inverse functions (like readers and accessors) are defined on a slot. When the inverse function is given an object or a Lisp value that may be the value of that slot, it returns all the objects (if there are any) whose slot contains that object or Lisp value.

In most cases, inverse functions return a list of all objects having the value in the slot. However, if the slot is specified `:unique t` (meaning at most one object will have a specific value in the slot -- a slot holding a serial numbers, for example), the inverse function knows that there is at most one object with the value in the slot, so it returns that single object (not in a list), or `nil` if no object has the value in the slot.

Note that inverse functions use information stored automatically in Lisp and in the database. This information identifies all objects that contain a pointer to the object we are trying to find, and so only those objects have to be examined. In most situations, there are few such objects.

If the value of the slot is a Lisp value that is not a persistent object, Lisp maintains a hash table indicating the objects that point to the value. If, for example, we had defined an inverse function for the `name` slot of `patron`, the system would use the hash table since the type of the name is not specified and so can be any Lisp value that can be stored in the database.

Back to our example, **borrower** is the inverse function of patron's `borrows` slot. It returns a list of patrons borrowing the `book`. We can use **borrower** to write a better version of **who-has-1**, called **who-has-2**:

```
(defun who-has-2 (library-name title author)
 (with-database (db library-name)
   (with-transaction ()
      (let ((p (first (borrower (get-book title author)))))
         (cond (p (format t "~s has it~%" (patron-name p)))
               (t (format t "No one has that book checked out~%")))))))
```

Since only one patron can check out a given book at a time, we can safely assume that the list will either be empty or have one element. Thus we can use **`first`** to extract the patron from the list. Note that the `borrows` slot is a set slot, and set slots cannot also be `:unique`. Therefore, the inverse function will return a list.

`who-has-2` is much more efficient than **`who-has-1`**. Given a `book` object, it can find the patron borrowing that book with just a few pointer traversals in the database. It will work just as efficiently if the database grows to a million patrons, which is something that is definitely not true of **`who-has-1`**, which has to examine each patron in order to find a book, rather than examining only those patron objects known to point to the book object.

Both **`who-has-1`** and **`who-has-2`** find the book instance with **`get-book`**. That function examines every book until it finds the one with the right author and title. Therefore, if our book collection grows large (say, to a million books), **`who-has-2`** will also be slow. (**`who-has-1`** will be even slower, of course, since it has to traverse the same number of books, and all the patrons!)

When the collection grows large, it would be more efficient to use inverse functions to look up all books that have a certain title, and then look through those for a matching author (or vice versa). We'll leave that as an exercise for the reader. (Hint: you have to redefine the `book` class so the `title` slot has an `:inverse` specified and do not forget that titles may not be unique.)

Deleting instances

Deleting instances

Persistent objects are retained in the database unless they are deleted with **`allegrostore:delete-instance`**. Consider what happens when patron Bob Smith borrows *Dandelion Wine* by Ray Bradbury,

```
(check-out "mylib.db" "Bob Smith" "Dandelion Wine" "Ray Bradbury")
```

When he gets home, Bob's dog chews the book up. Bob comes to the library and pays for the book. The librarian then calls **`delete-instance`** on the *Dandelion Wine* book object (with the database open and within a transaction, of course) and the book is no longer in the collection. (See the code below.)

But what has happened to the value of the Bob Smith's `borrows` field? It used to contain a set of at least one object, a pointer to the book object for *Dandelion Wine*. The librarian did not specifically modify that field in the database. But, if you check the `borrows` field after the **`delete-instance`**, you'll find that the field no longer contains the book object *Dandelion Wine*.

If we delete the instance with the following (rather than the form above), we will see that the book disappears from Bob's `borrows` field. Here is the complete transcript:

```
cl: (check-out "mylib.db" "Bob Smith" "Dandelion Wine" "Ray Bradbury")
"Bob Smith" has borrowed "Dandelion Wine" by "Ray Bradbury"
cl: (with-database (db "mylib.db")
      (with-transaction ()
          (let ((bob (first
                      (retrieve 'patron
                        :where '((patron-name equal "Bob Smith")))))))
           (format t "Before delete, bob is borrowing ~s~%"
                   (borrows bob))
           (format t "Deleting the book Dandelion Wine by Ray Bradbury~%")
           (delete-instance (get-book "Dandelion Wine" "Ray Bradbury"))
           (format t "After the delete, bob is borrowing ~s~%"
                                        (borrows bob)))))
Before delete, bob is borrowing (#<book in /db/pclos/mylib.db
                                  p: #x72f0004  @#x10583d2>)
Deleting the book Dandelion Wine by Ray Bradbury
After the delete, bob is borrowing nil
nil
cl:
```

Let us also look at the library inventory:

```
cl: (print-books-in-library "mylib.db")
Books in library mylib.db
Title: Marcovaldo   Author: Italo Calvino
Title: Baidarka   Author: George Dyson
nil
cl:
```

Referential integrity

Referential integrity

AllegroStore maintains *referential integrity*; all of the pointers inside the database point to valid database objects, never to places where an object, now deleted, used to be stored.

Because AllegroStore maintains referential integrity, the programmer needn't put explicit tests in his code to check each pointer reference for validity. When an object is deleted, AllegroStore finds all references and replaces them with the appropriate value as follows:

- If the reference comes from within a set (as in the `borrows` slot), then the reference is simply removed.
- If the reference is from a non-set slot, then what happens depends on the `initform` of the slot. If `initform` is `nil`, the value of the slot is set to `nil`. Otherwise, the value of the slot is replaced with the value representing an unbound value. (The programmer can arrange for other actions, e.g. by using `:before` methods to **`delete-instance`**. See the description of **`delete-instance`**, and also **`collect-references`** and **`map-references`** -- which discover all reference to an object --in chapter 7 **Reference guide**.)

5.3 Persistent class slots

CLOS defines two types of slots, `:instance` (the default) and `:class`.

The value of an instance slot is specific to an instance. Changing it affects one instance only. A `:class` slot is accessed like a normal slot, but there is only one copy of the slot for all objects of the class. Thus, if you change the value of the slot in any instance, the slot will be changed in all instances of that class.

We have already mentioned `:persistent` slots, which are like `:instance` slots except their values are stored in the database. AllegroStore provides in addition the `:persistent-class` slot, which acts like a `:class` slot except that, again, the value of the slot is stored in the database. Let us revisit some code we wrote earlier, and see where we can benefit by using a `:persistent-class` slot.

The code below is taken from the previous section under the heading **Changing the schema**. Recall that we changed the definition of the `book` class, to provide for a `bar-code` slot. That slot should contain a unique number identifying the book. We also defined the parameter `*next-bar-code-number*` to hold the next bar code number (to be assigned to the next book added to the database.

```
(defclass book ()
  ((title :allocation :persistent
          :initarg :title
          :accessor title)
   (author :allocation :persistent
           :initarg :author
           :accessor author)
   (barcode :allocation :persistent
            :initarg :barcode
            :accessor barcode))
  (:metaclass persistent-standard-class))

(defparameter *next-bar-code-number* 10001)

(defun assign-barcodes (library-name)
  (format t "Assigning barcodes to new books in ~a~%" name)
  (with-database (db library-name :if-does-not-exist :error :use :memory)
    (with-transaction ()
      (for-each ((b book))
                (cond ((not (slot-boundp b 'barcode))
                       (setf (barcode b) *next-bar-code-number*)
                       (format t "Book ~a by ~a assigned barcode ~s~%"
                               (title b)
                               (author b)
                               (barcode b))
                       (incf *next-bar-code-number*)))))))
```

So what is wrong? This code is run in Lisp. At the end of the day, the librarian shuts Lisp down and goes home. The parameter `*next-bar-code-number*` is then lost. The next day, a new shipment of books arrive and the librarian starts to add them to the database. However, the database contains no easily-accessible record of the next available bar code number. As things stand, the librarian will have to examine every book to find the largest bar code number used. Only then can a new book be accessioned.

A better solution is to define a persistent-class slot in the book class that identifies the next available bar code number. Then the information will be available whenever a new instance of book is created. And, when a number is used and thus the next available one increased, it is updated in every book instance. Here is how we can define the book class to include a persistent-class slot for the next available barcode. We follow with a function that finds any books without barcodes and assigns barcodes to those books.

```
(defclass book ()
  ((title :allocation :persistent
          :initarg :title
          :accessor title)
   (author :allocation :persistent
           :initarg :author
           :accessor author)
   (barcode :allocation :persistent
            :initarg :barcode
            :accessor barcode)
   (next-barcode :allocation :persistent-class
            :initform 10004
            :accessor next-barcode))
  (:metaclass persistent-standard-class))

(defun assign-barcodes (library-name)
  (format t "Assigning barcodes to new books in ~a~%" library-name)
  (with-database (db library-name :if-does-not-exist :error :use :memory)
    (with-transaction ()
      (for-each ((b book))
                (if* (not (slot-boundp b 'barcode))
                   then (setf (barcode b) (next-barcode b))
                        (format t "Book ~a by ~a assigned barcode ~s~%"
                                (title b)
                                (author b)
                                (barcode b))
                        (incf (next-barcode b)))))))
```

(You might argue that an additional slot in each book object is wasteful, since it increases the size of every book object. But this is not correct. The persistent class slots are stored with the class definition, not with the instance, and thus it appears only once in the database.)

5.4 Multiple databases

The examples so far have shown us accessing a single database. It *is* possible to have more than one database open at the same time, although it *is not* supported to have pointers between the databases.

Moving an object from one database to another

Moving an object from one database to another can be done by creating a copy of an object in the 'to' database and deleting the copy in the 'from' database. Here is a function which does just that:

```
(defun move-book (from-library to-library title author)
   (with-database (db from-library)
      (with-transaction ()
          (let ((book (get-book title author)))
            (with-database (db to-library)
               (make-instance 'book
                 :title (title book)
                 :author (author book)))
            (delete-instance book)))))
```

We can test it out by moving one of our remaining books from *mylib.db* to *newlib.db*:

```
(move-book "mylib.db" "newlib.db" "Marcovaldo" "Italo Calvino")
(print-books-in-library "mylib.db")
(print-books-in-library "newlib.db")
```

The output looks like:

```
Books in library mylib.db
Title: Baidarka   Author: George Dyson


Books in library newlib.db
Title: Marcovaldo   Author: Italo Calvino
```

(Readers who have been skipping about may wonder what happened to *Dandelion Wine* by Ray Bradbury. It was chewed up by a dog in section 5.2, under the heading **Deleting instances**.)

The `to-library` database doesn't have to exist before the transfer because we used **`with-database`**. **`with-database`** will create the database and open it if it is not already on the disk.

Note that we do this in one transaction and that the **`with-transaction`** form can be outside of the **`with-database`** form. We didn't copy the `barcode` when we moved the `book`, since different libraries may use different `barcode` numbers for the same `book`.

That's the end of the tutorial.

Chapter 6 Programmer's guide

Users of AllegroStore can be broadly divided into three categories:

- **Administrators**. These users are responsible for setting up databases, providing standard functionality (for reading and writing data), maintaining databases, etc.
- **Advanced users**. These users will typically use databases set up by administrators. They will be able to write programs that query the database in particular ways and perhaps modify the data based on the results of the queries.
- **Regular users**. These users also typically use databases set up by administrators and they will use the standard query and modify functions (that are typically written by administrators or advanced users). They are not usually expected to know how to write specialized programs that query or modify the database.

Of course, these categories blur into one another. A regular user (say a data entry clerk) may learn about programming in hopes of a promotion; an advanced user may be called upon to do administrative work, and an administrator will usually be able to write custom programs. And everyone will probably do data entry from time to time. But these categories are useful to keep in mind.

This chapter tells how to write specialized programs to query and modify a database. This material is written for Advanced users and Administrators. Chapters 8, 9, and 10 give information on setting up and maintaining databases.

A quick example

Let's illustrate the functions of the types of users listed above by considering a phone company.

Consider a long distance telephone company. Their database contains customers; each customer's record will contain (or point to) a list of calls made, money owed, and money paid.

The administrator will set up the database and write functions for accessing the data and modifying it.

The company employs customer service representatives. They access the database by using the functions written by the administrator. These functions allow the service representatives to answer standard customer questions and modify the data.

The telephone company's marketing department decides to have a promotion. Any client who makes 200 calls during March will get a $50 credit. A programmer will write a program specifically for this purpose: it will examine every record, count the calls made in March, and if the number of calls is greater than 199, apply the credit.

(We will not actually write such code. This example was simply to show where the types of users listed above might appear.)

6.1 Organization of this chapter

This chapter is divided into several sections, each dealing with a specific topic.

1. **Organization of this chapter**. The section you are now reading.
2. **A review of CLOS concepts**. AllegroStore is based on CLOS. This section review some CLOS concepts and terms. AllegroStore's new metaclass, `persistent-standard-class`.
3. **The database**. What a database file is, and how AllegroStore connects to it. How to name, create, move, copy, and open a database. The current database.
4. **The schema**. What a schema is, how it is defined, modified, and reconciled with its stored definition once it has been modified.
5. **Transactions**. How transactions work to provide consistent database information to simultaneous users.
6. **Slots**. What types of slots are available, and what type of data can be stored in them. Operations on slots. How to return values from the database with methods and inverse functions. Querying speed trade-offs for inverse functions.
7. **Persistent hash tables**. When hash tables are used, and how changes in Lisp objects are reflected in the database. Why persistent hash tables cut down on the cost of converting objects from database format to Lisp format and back again.
8. **Blobs.** Dealing with large data sets.
9. **Persistent ftype arrays.** Arrays of foreign (rather than Lisp) types.
10. **Inverse functions.** What inverse functions are, and how they speed up querying.
11. **Queries and iterators**. Searching for data by two methods: iteration over a group vs. search-for-a-certain-condition. Why iteration is less memory-intensive.
12. **Pointers.** Which pointers are allowed, and when they are valid. How persistent objects are pointed to in the database. About **`preserve-pointer`**: when pointers become invalid, and how to make a pointer last through multiple transactions.
13. **Implicit object creation**. How accessing a persistent object more than once creates more than one transient object, and these objects are not **`eq`**.
14. **Object deletion**. How object deletion works**.**
15. **Referential integrity**. How AllegroStore finds all references to a just-deleted object and resolves them.
16. **Object update**. How lazy and eager updating work. Automatic object updating. How to find references to an object before updating or deletion.
17. **Multiprocessing**. AllegroStore and multiple processes within Lisp.
18. **Interactive transactions**. When learning and testing, it may be useful to keep a transaction open. This section tells you how.
19. **Persistent object check out and check in**.
20. **Reducing page lock contention**.

21. **Read-only processing**.
22. **Multi version concurrency control (MVCC) processing**.
23. **Long transactions.**
24. **Notifications.**

6.2 A review of CLOS concepts

AllegroStore is a database for CLOS objects. We assume that the reader is already familiar with CLOS, the Common Lisp Object System, and it is beyond the scope of this document to provide even a very basic introduction. We recommend that users who need more information about CLOS consult the following books:

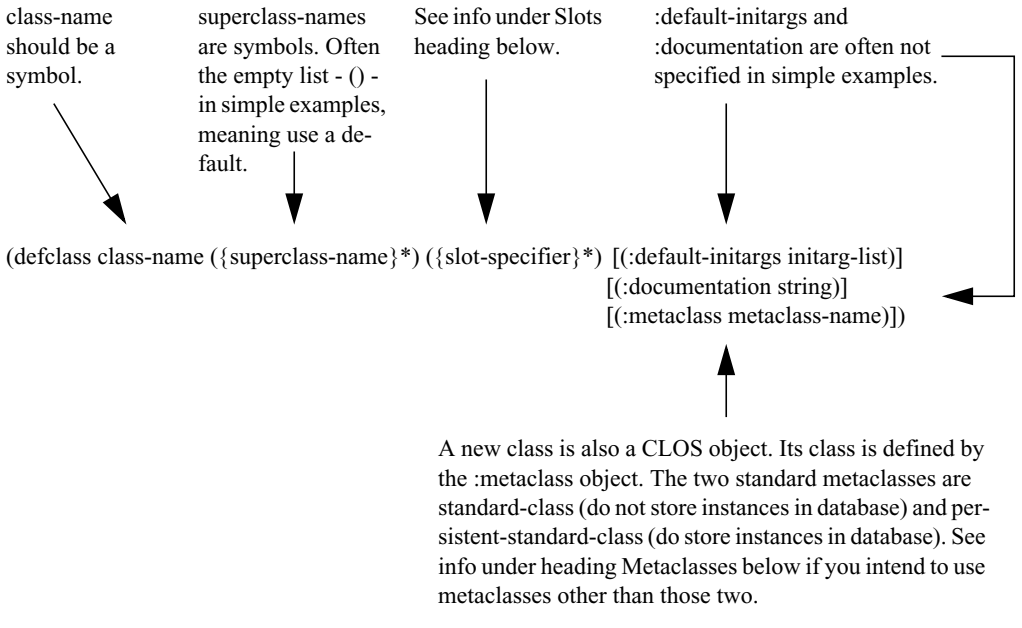
- For the technical specification: *Common Lisp, the Language* (2nd edition), by Guy Steele, Jr. 1990. Published by Butterworth Heinemann. (Do not get the first edition.)
- For a good introductory text for CLOS: *Object-oriented Programming in Common Lisp, a programmer's guide to CLOS*, by Sonya Keene, 1989. Published by Addison Wesley.
- For an introduction to the MetaObject Protocol: *The Art of the Metaobject Protocol*, by G. Kiczales, J. des Rivières, and D. Bobrow, 1991. Published by MIT Press.

In this section we will very briefly review some CLOS concepts and terms. We will show the standard defining forms, so that a user of this document can quickly analyze code samples without having to refer to the more complete texts referenced above.

Classes

A CLOS application typically defines a hierarchy of classes. A *class* is defined with superclasses (from which it inherits attributes, including some slots) and slots (which can hold data). Typically, the class hierarchy models the real world situation of interest. Classes can have subclasses, which increase specificity. For example, a company may define a class of `personnel`, with subclasses `employee` and `contractor`. Employees may be further broken down into `technical`, `clerical`, and `managerial`.

Classes are defined with the **`defclass`** macro. Note the {}'s, *'s, and []'s. A * means the indicated form may appear as many times as necessary (or not at all). A form surrounded in brackets ([]'s) may appear once or not at all. Braces ({}'s) simply indicate a construct to which we can append a *. Neither braces or brackets appear in an actual call.



For example, here is the definition of the `patron` class from chapter 5 **Tutorial**:

```
(defclass patron ()
  ((name :allocation :persistent
         :initarg :name
         :reader patron-name)
   (borrows :allocation :persistent
            :type book
            :set t
            :accessor borrows
            :inverse borrower))
  (:metaclass persistent-standard-class))
```

The class name is `patron`. There are no superclasses specified (so the superclass defaults to `persistent-standard-object`). There are two slots: `name` and `borrows`. There are no default initargs and no documentation string. The metaclass is `persistent-standard-class`.

Slots

A class has defined with it a set of *slots*. Each slot has a name and can hold a Lisp value or a set of Lisp values. Slots can be defined when the class is defined (as the `name` and `borrows` slots of the `patron` class above). Slots can also be inherited from superclasses, if any are specified. Slots are defined with a form that looks like the following (as above, the braces - {}'s - simply identify a construct and do not appear in the actual form while the * indicates the construct may appear as many times as necessary or not at all):

```
(name {arg-label arg-value}*)
```

`name` is a symbol naming the slot. *`arg-label`* is a keyword (a symbol starting with a colon). *`arg-value`* is often a symbol which either provides a name or provides information about the slot, but it is sometimes something else, for example a list. We will not go into detail here about all the slot options, but we will mention the `:allocation` option.

Each slot has an *allocation type*, the value of the `:allocation` option. The two allocation types in standard CLOS, `:instance` (the default) and `:class`, AllegroStore allows two additional types `:persistent` and `:persistent-class`. These are analogs of `:instance` and `:class`, with the additional specification that the slot value is stored in the database. See section 6.6 **Slots** below for more information.

A program can retrieve the value of any slot using the **`slot-value`** function.

A program can also define an accessor function for a slot with the `:accessor` option. Whether or not an accessor is defined, the slot value can always be accessed with **`slot-value`**. (In the definition of the patron class above, **`borrows`** is specified as the accessor function for the `borrows` slot.)

The value in a slot can be changed with **`setf`** and **`slot-value`**:

```
(setf (slot-value object 'slot-name) newvalue)
```

or with **`setf`** and the accessor function, if there is one:

```
(setf (accessor-function-name object) newvalue)
```

Refer to section **7.2.4 Schema manipulation** for some special consideration about the slot `:initform`s and class `:default-initarg`s options.

Instance creation

Having defined the class, we can create instances with **make-instance**. The first argument to **make-instance** is the class object (or class name) which will be the class of the instance created. The following form creates an instance of patron:

```
(make-instance 'patron :name "John Smith")
```

Note that only the name slot has an initarg, so only that slot can be initialized in the **make-instance** form. The borrows slot must be updated after the instance is created. (Whether or not to have an initarg depends on what you are modeling. The patron class defined above is the patron of a library. New patrons of a library always have a name, but that cannot borrow books until they get a card and go to the checkout desk. Therefore, there is no reason to have a borrows initarg.)

When a persistent object is retrieved from a database, a transient copy of the object is recreated in the Lisp heap so that code can manipulate it. The AllegroStore implementation takes care of intializing this transient copy when it is retrieved, and writing back to the database any changes made to the object's slot values at the end of each transaction.

The recreation and initialization of this transient copy presents a few issues not encountered in normal CLOS operation. In the programmer's model, of course, the object is *not* a freshly created one but rather is the *same* object as was earlier stored in the database. But in the implementation's model, the object is indeed a new object, and some aspects of normal object initialization must be performed in order to set its slot values and to do any other initialization that the object's metaclass may require.

AllegroStore recreates a transient instance of a persistent-standard-object by calling **allocate-instance**, then individually setting the values of its persistent slots, and finally by calling **shared-initialize**, passing a list of the object's transient slots but without passing any additional intialization arguments from the default-initargs. The net result is that a transient slot will be initialized to the value resulting from executing its :initform, if any, but will not be initialized from default-initargs. The default-initargs are used, of course, when the object is first created, i.e. by **make-instance**.

Metaclasses

A class defined with **defclass** is itself a CLOS object, and as such has a class of its own. This class is called the *metaclass* of the class being defined. The metaclass determines how an instance of the class being defined is created. The default metaclass is standard-class. standard-class knows how to construct CLOS objects in Lisp's memory. Here we define the auto class with metaclass standard-class:

```
(defclass auto ()
       (bumper tires engine doors)
       (:metaclass standard-class))
```

In fact, the :metaclass standard-class line is not necessary since standard-class is the default metaclass.

AllegroStore defines a new metaclass called persistent-standard-class. Instances with metaclass persistent-standard-class are created both in Lisp and in the current open database (an error is signaled if there is no database open).

To change the auto class into one whose instances are persistent, we need to change the :metaclass argument.

```
(defclass auto ()
        ((bumper :allocation :persistent)
         (tires :allocation :persistent)
         (engine :allocation :persistent)
         (doors :allocation :persistent))
        (:metaclass persistent-standard-class))
```

Notice that we also specified the `:allocation` types of each of the slots as `:persistent`. This means their values will also be stored in the database along with the auto instance. It is not necessary to define all (or any) slots as persistent and there are situations where you only want some of the slots to be persistent (to avoid wasting space in the database with information that need not be permanently stored). But remember that classes and slots are not persistent (stored in the database) unless they are defined to be so. The default in all cases in not-persistent.

Defining your own metaclasses

This is a note for advanced CLOS users. You may wish to define you own metaclasses which are subclasses of `persistent-standard-class`. This is of course permitted but note that there is a missing link in the connection between a database and Lisp: metaclass definitions are not stored in the database (although the name of the metaclass is stored). Therefore, you must load the definition of the metaclass prior to opening a database containing instances with metaclass other than `persistent-standard-class`.

6.3 The database

AllegroStore uses the *ObjectStore* persistent storage manager written by Object Design. ObjectStore supports two kinds of databases: *file databases* and *Directory Manager databases*.

A file database is stored in a normal file.

A Directory Manager database is stored in a section of the disk managed by ObjectStore. Because Directory Manager databases are harder to use, we won't discuss them in this part of the manual. You can read more about them in section 8.10 **Directory manager databases**.

Database naming conventions

Here are some possible ways a database can be named.

- A local filename, either *relative* or *absolute*. Here are examples of relative filenames, on Unix and on DOS:

  on Unix: `march/inventory`

  on DOS: `march\inventory`

  And here are some absolute filenames:

  on Unix: `/records/sales/june`

  on DOS: `d:\records\sales\june`
- A host name and an absolute filename:

  On Unix: `tiger:/dbs/people`

  On DOS: `tiger:c:\dbs\people`

- A Directory Manager name and a filename:

  On Unix: `pieces::/records/beatles`

  On DOS: `pieces::\records\beatles`

Punctuation rules for database names:

A Directory Manager filename is distinguished by a name followed by two consecutive colons.

A host and filename database name contains a single colon.

Everything else is a local name.

Special cases:

On DOS machines, a single character followed by a colon is interpreted as a local device name and not a host name

A colon is treated as a normal filename character if it follows a slash.

The Cache Manager

To create or open a database, AllegroStore connects to a process running on the same machine called *the Cache Manager*. If the Cache Manager process isn't currently running on the machine, AllegroStore will start it running.

The Database Server

The Cache Manager then contacts *the Database Server* process that controls access to the database. The Database Server for a given file is the machine controlling the disk on which the file is stored.

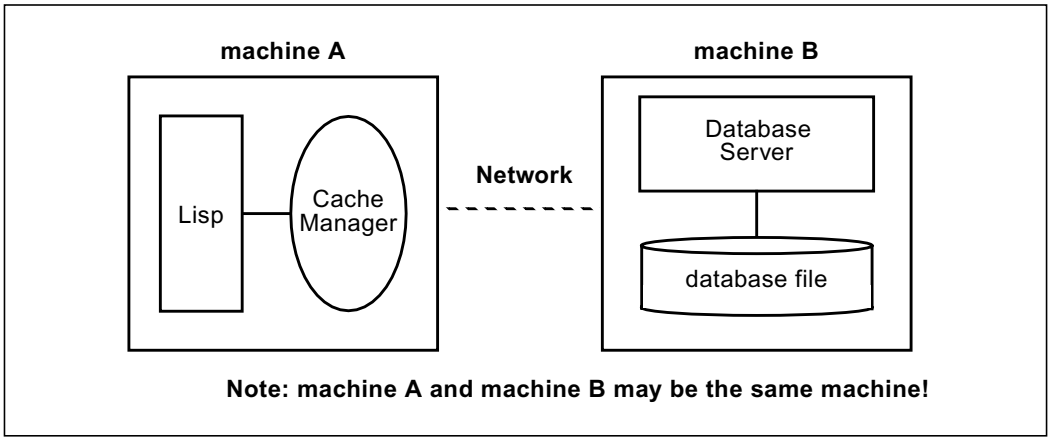
- If you are creating or opening a database, and you supply a filename like *fred:/usr/barney/rocks*, then this explicitly states that the Database Server is the machine *fred* and the database file is */usr/barney/rocks*, **which must be on one of the disks connected to the machine** *fred*. (Actually, you can set things up so the server can be on a different machine. See Technical memorandum # AS 2 for more information.)
- If you supply a *local* filename, then that filename may refer to a file on a local disk, or it may refer to a file on a disk on some other machine in the network (if the system supports a network filesystem).

The Cache Manager will determine which machine owns the disk on which the database lies, and will contact the Database Server on that machine. If the Database Server is not running, then the open database process will fail. AllegroStore cannot start a Database Server process automatically.

Even if the database file is on the local machine, accessing a database requires that three processes be running:

1. the Lisp process running AllegroStore
2. the Cache Manager

3. the Database Server



Remember! Unless you have followed the instructions in Technical Memorandum # AS-2, the Database Server must run on the same machine as the disk where the database sites are located. For more information, see section 8.4 **The Server**.

Moving and copying databases

If you plan to move or copy AllegroStore databases, be aware that the database server process may be caching some information that belongs in the file. Copying a file with **`cp`** (on Unix) or **`copy`** (on DOS) may result in an incomplete database file.

The ObjectStore *bin* subdirectory supplied with AllegroStore contains replacement file management functions that are database-aware. You should use the ObjectStore functions **`oscp`** for copying files, **`osmv`** for renaming them, and **`osrm`** for deleting them.

See the sections on **`oscp`**, **`osmv,`** and **`osrm`** in chapter 10 **User utilities**.

Creating the database

Creating or opening a database is done by using the **`with-database`** macro or with the **`open-database`** function. These are similar to Lisp's **`with-open-file`** macro and **`open`** function.

The default behavior for **`with-database`** and **`open-database`** is to open the database if it exists, otherwise to create it. Use the `:if-exists` and `:if-does-not-exist` keywords to alter this behavior.

`with-database` opens a database, evaluates the forms in the body of the macro, and closes the database when control leaves the body of the macro. We encourage you to use **`with-database`** because it ensures that the database always gets closed at some point if an error should occur. **`open-database`** does not ensure that. A database opened with **`open-database`** must be closed explicitly.

In the following code sample, **`with-database`** calls **`function1`**, **`function2`**, and **`function3`** on the database called *autotire.db*. Once **`function3`** completes, *autotire.db* is closed.

```
(with-database (db "autotire.db" :if-exists :supersede)
  (with-transaction ()
     (function1)
     (function2)
     (function3)))
```

The current database

More than one database can be open at the same time, but only one database is considered the *current database*. The current database is the implicit argument to functions such as **make-instance** and **set-schema**. The variable *db* holds the pointer to the current database.

The **with-database** macro opens a database and designates it as “current” for the duration of the macro's body by binding *db* to that database.

If you use the **open-database** function, then you must use **with-current-database** or **set-current-database** to set the current database before doing any database operations. The following is a typical call to **open-database**, with **set-current-database** setting the value of *db*:

```
(set-current-database (open-database "db-name" [other options])
```

See sections 7.2.1 and 7.2.3 for formal definitions of *db* and the various functions and macros mentioned here.

6.4 The schema

What is a schema?

In an AllegroStore database, the *schema* is a set of CLOS class descriptions. Changing the schema is a simple operation. The schema can be modified while the database is open simply by redefining one of its classes in Lisp. The redefined class definition will propagate into the database, and instances of that class will be changed to fit the new schema as they are accessed. This dynamic updating can occur while other programs are accessing the database.

In contrast, in a traditional database, the schema is a description of the records stored in the database and changing the schema is a major operation. All users must stop using the database, and then the whole database must be updated all at once. Programs using a traditional database will need to be recompiled after an update if they have the old schema compiled into them.

How a schema is set

The rules regarding which classes make up the schema of a database:

1. The class definition for each object in the database must be in the schema.
2. If class X is in the schema, and class Y is one of X's direct-superclasses, then class Y must be in the schema.

Classes may be added to the schema automatically or explicitly. The default behavior is to add classes to the schema automatically whenever an object whose class isn't already in the schema is added to the database.

Whether or not classes are added automatically is controlled by the function **allegrostore:set-schema** function. That function can also be used to control precisely which classes are in the schema. Schemas where classes must be added explicitly are said to be in *exact* mode. In exact mode it is an error to attempt to store an object in the database whose class isn't already in the schema.

See the section 7.2.4 **Schema manipulation** in chapter 7 **Reference guide** for more information.

How schema differences are reconciled

When AllegroStore opens a database, it synchronizes the definitions of classes in the database's schema with the definition of the classes in its Lisp memory. This synchronization is called *schema resolution*, which works as follows. When AllegroStore opens a database, it looks at all the class definitions in the database. For each class definition, there are three possibilities:

1. There is no class with the same name in Lisp's memory. In that case, the class definition in the database is used to define a class in Lisp's memory.
2. There is a class with the same name and same definition in Lisp's memory. In that case, there is no work need be done.
3. There is a class in Lisp's memory with the same name and *different* definition. This is the complicated case since the differences must be resolved. Either the database or the memory definition of the class must be used. The program can specify which definition to use in the **`with-database`** or **`open-database`** calls (with the `:use` keyword argument -- see the formal definitions in section 7.2.3 **Database manipulation** for information on the `:use` argument).

   If the program does not specify how to resolve the difference, AllegroStore will signal a `allegrostore-class-mismatch` condition. You can set up the condition handler in the usual Common Lisp way to handle the condition. Unhandled, it results in a continuable error with using the memory definition and using the database definition among the restarts. In the following transcript, we define a class in a database and close it, redefine the class in Lisp, and then re-open the database (without specifying the `:use` argument).

```
USER(1): (use-package :astore)
T
USER(2): (defclass tire () ((brand :allocation :persistent
                                   :initarg :brand))
            (:metaclass persistent-standard-class))
#<PERSISTENT-STANDARD-CLASS TIRE>
USER(3): (with-database (db "test-tires.db")
           (with-transaction () (make-instance 'tire :brand 'firestone)))
#<TIRE in /db/mjm/test-tires.db (CLOSED-DATABASE) @ #xc5d7da>
;;  We now redefine the class in memory (the database is currently closed).
USER(4): (defclass tire () ((brand :allocation :persistent :initarg :brand)
                            (rimsize :allocation :persistent))
            (:metaclass persistent-standard-class))
#<PERSISTENT-STANDARD-CLASS TIRE>
;;  When we re-open the database, we get an error:
USER(4): (setq *db* (open-database "test-tires.db"))
Error: class TIRE in memory doesn't match definition in database
"/db/mjm/test-tires.db"
 direct-superclasses:
   memory: (PERSISTENT-STANDARD-OBJECT)
database: (PERSISTENT-STANDARD-OBJECT)

 direct-slots:
  memory: ((:NAME BRAND :TYPE T :ALLOCATION :PERSISTENT :INITARGS (:BRAND))
           (:NAME RIMSIZE :TYPE T :ALLOCATION :PERSISTENT))
```

```
  database: ((:NAME BRAND :TYPE T :ALLOCATION :PERSISTENT :INITARGS
           (:BRAND)))
metaclasses:
  memory: PERSISTENT-STANDARD-CLASS
  database: PERSISTENT-STANDARD-CLASS

  [condition type: ALLEGROSTORE-CLASS-MISMATCH]

Restart actions (select using :continue):
 0: Use the definition of TIRE from the database on disk
 1: Use the definition of TIRE and all other classes from the database on
    disk
 2: Use the definition of TIRE currently in memory
 3: Use the definition of TIRE and all other classes currently in memory
 4: Rollback the current transaction, returning nil
 5: Rollback the current transaction, and restart it
[1] USER(6):
```

See section 6.15 **Object update** for more information on how schema changes are handled.

6.5 Transactions

A quick illustrative example

Suppose you have a personnel database. There is an object representing each employee in which the employee's name and salary is stored. There is also an object representing the company in which the total of all employee salaries is stored. You want the total salary to always be the sum of the individual employee salaries.

Consider the database operations necessary to give an employee a 5% raise:

- Find the employee object in the database,
- retrieve the employee's salary,
- multiply it by 1.05,
- and store it back in the database.
- Access the company object,
- access the total salary amount,
- add to it the amount of the employee's salary increase,
- store the value back into the total-salary slot of the company object.

If the program should fail *after* it increased the employee's salary but *before* the total-salary was increased, then the database would be left in an inconsistent state. All database operations necessary to change a salary should be grouped together in such a way that either all of them happen or none of them happens. This grouping is called a *transaction*.

Transactions prevent database inconsistency; all changes to the persistent data take place inside of a **`with-transaction`** form.

What is a transaction?

A *transaction* is a sequence of database operations.

Each transaction is *atomic*: either all of the database operations made during the transaction are made permanent or none of them are made at all.

When all of the operations are made permanent, we say that the transaction was *committed*. When none of them are made, we say that the transaction was *rolled back*. When a transaction is rolled back, all of the database objects revert to the state they were in prior to the transaction.

Transactions give each program the illusion that it is the only program accessing the database. That is, if a program were to:

- start a transaction,
- access all employee objects, and
- add up their salaries,

then it would find that its computed total matched the total-salary contained in the company object. This would be true even if other programs were accessing the database and changing employees and salaries. A consistent view of the database is only ensured within a single transaction.

If multiple transactions are used to perform computations like this one (or any activity requiring access to stored data), then a consistent view of the database is *not* guaranteed. If the program just described above were modified to:

- start a transaction,
- add the employee salaries,
- end the transaction,
- start a new transaction, and
- access the total-salary from the company object,

then it might find that the value of total-salary doesn't match what it just computed from the employee salaries. Between the time that the program ended its first transaction and started its second one, another program could have accessed the database and changed the total-salary.

A model for transactions

The following model will show how transactions are implemented. It will provide users with a mental image of what is going on and make the discussion more concrete. It will *not* explain the inner workings of AllegroStore (the actual implementation is different in ways we briefly describe afterwards). This model will show that the permanent database is always consistent because any change is either done completely or not done at all.

Assume that the model is a personnel database. We'll go through the steps required to increase the salary of an employee, Betty Scrivener, by 10%.

The database is a file containing a description of the class hierarchy and a record of each instance in the database. All of the data associated with a particular instance is located in a specific location on the disk which contains the database file. Betty's `employee` instance resides in one spot on the disk, say between hexadecimal address bytes 12a5e and 131ab, for example. The `company` instance resides in another specific spot on the disk.

In our model, the system does the following when we give Betty a raise:

1. Marks (by setting a flag at the beginning of the `B.Scrivener` instance data, byte 12a5e) the Betty Scrivener instance as 'in use'. This prevents other database clients from reading the data until the transaction commits (finishes properly).
2. Copies the data associated with the `B.Scrivener` instance to a temporary location.
3. Makes the changes to the `salary` slot in the temporary location. (The only change in the permanent location so far is to mark the instance as being 'in use'.)
4. Marks (by setting another flag at the beginning of the `company` instance) the `company` instance as 'in use'.
5. Copies the data associated with the `company` instance to a temporary location.
6. Makes the changes to the `total-salary` slot in the temporary location.
7. Copies both modified instances back into the permanent database file.
8. Changes the 'in use' marks to 'available'.

By making all the changes in the temporary location, the permanent record is not affected until it is known that all the necessary changes can be made. If they cannot be made, the transaction rolls back.

To roll back a transaction, the system must delete the information in the temporary location and change the ‘in use’ flags to ‘available’.

To commit the transaction, the system must copy the modified data into the permanent database and change the ‘in use’ flags to ‘available’.

If another client tries to access an instance that is ‘in use,’ it must wait until the instance becomes ‘available.’ The data can never be worked on simultaneously by two clients.

See **About commit** and **About roll-back** in the section 7.1.

The real database

Two comments on how the real database differs from our simplified model.

- **Pages of memory are locked rather than specific instances.** When a `with-transaction` touches an instance, the page or pages in memory containing the instance data are locked, not just the instance data. Locking a page is simply faster and more efficient than locking specific instances.
- **There are read-locks and write-locks, rather than just one type.** A read-lock says ‘Do not change this page until I am finished with it’ but does not prevent others also reading it. A write-lock says ‘I am about to modify this page, so do not even look at it’. Using two kinds of locks allows a larger number of simultaneous transactions.

The differences between our simplified model and the real implementation are invisible to users. They only affect the actual efficiency and performance of database programs, not what the programs actually do.

The problem of deadlocks

Suppose that two different clerks are attempting transactions in the personnel database:

The company has promoted and transferred an employee, Tom Bombadil. One clerk wants to access the `T.Bombadil` instance, give him a raise, and change the `total-salary` slot in the `company` instance. Another clerk wants to grab the `company` instance, change the `employees-located-here`, and change Tom’s record to reflect his new work-site.

The first clerk starts a transaction which accesses the `T.Bombadil` instance, marks it as ‘in use,’ and tries to grab the `company` instance. The second clerk has also started a transaction, and has already marked the `company` instance as ‘in use’ and is trying to grab the `T.Bombadil` instance.

Well, that is a problem. Clerk 1 has `T.Bombadil` and cannot free it until the `company` instance is free. Clerk 2 has the `company` instance and cannot free it until `T.Bombadil` is free. This is a *deadlock*.

Deadlocks are detected and resolved by a separate program called the Database Server. That program runs constantly, checking on each transaction. It notices when a transaction is blocked because an instance is ‘in use’ and determines what the problem is. If it detects a deadlock, it sends a message to one of the transactions telling it to roll back and try again later.

Suppose this message goes to the transaction started by Clerk 2. That transaction rolls back, making the `company` instance available. The transactions started by Clerk 1 can now complete, freeing the `T.Bombadil` instance. Clerk 2’s transaction can now restart and it will likely find that it can complete.

How are deadlocks detected? It is beyond the scope of this document to describe that. The Database Server implements a complicated (but very efficient) algorithm to detect deadlocks.

How are deadlocks resolved? There is a default setting, but the database administrator can provide different rules for deadlock resolution based on various criteria. See the essay **About deadlock resolution** in section 7.1.

How are transactions started and committed or rolled back?

AllegroStore transactions are started and ended by the **`with-transaction`** macro. In the example below, **`with-transaction`** starts a transaction, calls the functions **`fun1`**, **`fun2`** and **`fun3`**, and then commits the transaction.

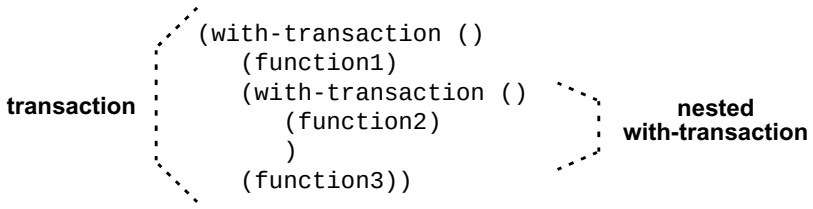
```
(with-transaction ()
   (fun1)
   (fun2)
   (fun3))
```

The transaction will be rolled back if one of the functions **`fun1`**, **`fun2`** or **`fun3`** does a non-local exit, such as a **`throw`** to a **`catch`** tag or does a **`go`** to a **`prog`** tag outside the **`with-transaction`** form.

```
(catch 'foo
   (with-transaction ()
       (some-database-action)
       (throw 'foo nil) ; will roll back the transaction
         ))
```

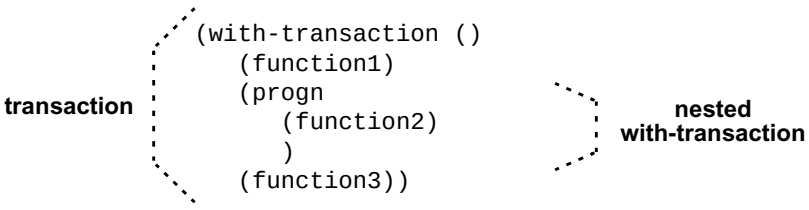
Nested transactions and top-level transactions

A *nested transaction* is a transaction started while another transaction is in progress. The following diagram shows a nested transaction:



Allegrostore does *not* support nested transactions. It is not an error, however, to nest **`with-transaction`** forms. The inner forms are just treated differently.

When AllegroStore encounters one **with-transaction** form called within another **with-transaction** form, the inner **with-transaction** is converted to a **progn**. The code in our diagram above, for example, is converted as follows:



Why is this important? Here is the principle to remember:

> **Changes to the permanent database are not guaranteed to have occurred just because a with-transaction form completes without error.**

Therefore the following code fragment may be erroneous:

```
(with-transaction ()
    (process-data))
(format t "Data has been processed~%")
```

If this **with-transaction** form is nested inside another, the report printed by the **format** statement is not guaranteed to be true. If the outer **with-transaction** form is rolled back (or perhaps fails because of an error), the changes made by **process-data** will not have occurred when the report is printed (and if there is an error, may never occur!)

Top-level transactions: committed when complete

A top-level **with-transaction** form is one that is not contained within another **with-transaction** form. When a top-level **with-transaction** form completes, the changes to the database called for within the form are guaranteed to have been made.

The **with-transaction** macro has a keyword argument `:top-level`. When this argument is specified non-`nil`, an error will be signalled if the **with-transaction** form is in fact nested within another. The following code, therefore, is not erroneous:

```
(with-transaction (:top-level t)
    (process-data))
(format t "Data has been processed~%")
```

The transaction-active-p function

The function **transaction-active-p** returns true if a transaction is active (i.e. a **with-transaction** form is being executed) and returns `nil` otherwise. Our code fragment that reports that the 'Data has been processed' could also be done correctly this way:

```
(with-transaction ()
    (process-data))
(if (null (transaction-active-p))
    (format t "Data has been processed~%"))
```

(The **format** statement will not execute when the transaction is nested, so the programmer should think of a fallback method of reporting the success. However, the 'Data has been processed' statement will not be printed erroneously.)

Transaction restarts

We have already described in our database model how a transaction sets a flag to indicate that data is 'in use.' That flag is called a *lock*.

A lock is a declaration that a certain program has certain access rights over a set of objects in the database. The Database Manager uses locks to ensure that multiple programs accessing a single database don't clash.

During a transaction, an AllegroStore program obtains locks on pages of the database it is accessing. When the transaction completes (with a commit or roll back), all locks are released. Should a program require a lock that another process has, it will wait *forever* (unless it meets a deadlock) to obtain that lock before giving up and signalling a condition. Forever is the default.[1]

If program Z requires a lock that program Y has, and then program Y requires a lock that program Z has, we have a *deadlock*. Neither Z nor Y can continue. AllegroStore detects this situation and selects either program Z or program Y for a *transaction restart*.[2]

Here are the details on what a transaction which must roll back can and should do:

Suppose program Z must roll back. A condition is signalled in program Z which causes it to do a transaction roll back, freeing all its locks, and then control to returns to the top of the **`with-transaction`** form, which starts executing again.

Code in a with-transaction form may execute many times

The rollback and restart are all done automatically, invisibly to users. Note the *important implication* of this arrangement: the body of a **`with-transaction`** form can be executed more than once. You must program with this in mind and avoid including actions that cannot safely be evaluated many times.

Here is a (correct) example which will build a list of all of the different automobile names in a tire company's database:

```
(defun show-autos-1()
   (with-database (db "autotire.db" :if-does-not-exist :error)
     (let ((namelist
            (with-transaction ()
              (let ((names nil))
                (for-each ((a auto))
                  (push (auto-name a) names))
                names))))
       (dolist (name namelist)
           (format t "~%~A" name)))))
```

Why build a list of auto-names and then print the names *after* the body of the **`for-each`** was executed? If a transaction restart occurs in the middle of the **`with-transaction`**, control is thrown back to the top of the transaction. A list of names partially printed from inside of the **`for-each`** would begin printing again after a restart, but you would not get a signal that the transaction had restarted. You might not be aware that the output, which could show some names printed more than once, wasn't accurate.

Note that we do print within transactions in some examples in this manual. That makes the examples easier, but it is not good practice!

1. See **About read-locks and write-locks** in section 7.1 for instructions on how to reset the default wait time for a lock.
2. See the **About deadlock resolution** in section 7.1 for details on how a process gets selected to roll back and restart.

6.6 Slots

Slots typically contain the data of an instance. The issue in AllegroStore is whether the data in a slot is persistent (stored in the database) or transient (stored only in memory while Lisp is running).

Types of slots

CLOS has two types of slots: *instance* (the default) and *class* (often called *shared*).

A class slot is accessed like a instance slot, but there is only one copy of the slot for all objects of the class. Thus, if you change the value of slot, the change will be visible to all objects of that class.

AllegroStore adds two new kinds of slots to persistent classes. The complete list of slot types, denoted by their slot descriptions, are shown in the following table.

Slot type	Description
`:allocation :instance`	the default, a non-persistent slot
`:allocation :class`	a non-persistent shared slot
`:allocation :persistent`	a persistent slot
`:allocation :persistent-class`	a persistent shared slot

:persistent slots

A `:persistent` slot is like an `:instance` slot except that a `:persistent` slot's value is stored in the database.

:persistent-class slots

A `:persistent-class` slot is like a `:class` slot except that a `:persistent-class` slot's value is stored in the database.

Non-persistent slots

An instance of a persistent class can have non-persistent slots. The value of non-persistent slots are stored in Lisp's memory, not in the database.

Why would you want non-persistent slots? One example is to store values derived from the values of persistent slots which are of interest to some user of the database. Since the values can be derived from already-stored data, they do not need to be stored on the disk.

Remember the personnel database example? An employee who wants a long vacation might ask the Personnel department to calculate how many vacation days he would have earned by next summer. The personnel officer who does the calculations could make them based on two or three of the persistent slots in the database.This data won't be needed after the employee gets an answer to his question; storing it as persistent data wastes valuable disk space. It is best to keep it in a non-persistent slot.

Of course, if the derived value requires complicated calculations and is of a type needed very often, it might be better to calculate it and store it in a persistent slot. Decisions on whether to save on CPU time at the expense of disk space are usually made by the database administrator.

Set-valued slots

Persistent slots are either *single-* or *set-valued*. A set-valued slot contains a unordered collection of objects. A single-valued slot holds a single value (which may itself be a list, of course) and is also called a *scalar* slot.

The advantage of storing a list of values in a set-valued slot over storing the list in a scalar slot is that a program can do a query over elements in a set-valued slot without constructing the entire list in Lisp's memory. Also, set-valued elements may be successfully used as inverse function arguments.

The accessor for a set-valued slot returns a list of the values stored in the slot. You must store a list of values (or `nil`) in a set-valued slot:

```
(setf (accessor-name object) (list object1 object2 object3))
```

The **`slot-cons`** method is the recommended way to add new values to a set-valued slot:

```
(slot-cons object 'accessor-name object4)
  ;; object4 is added to object's accessor-name set-valued slot
```

There are two ways in which a slot can be declared to be a set slot in a **`defclass`** slot description. One way is to specify `:set t` and the other is to specify `:type (set-of X)` where `X` is some Lisp type. Here are two are equivalent ways to specify that the value of the `borrows` slot is a set of `book` objects:

```
(defclass patron ()
  ((name :allocation :persistent
         :initarg :name
         :reader patron-name)
   (borrows :allocation :persistent
            :type book                  ;; we specify :type
            :set t                      ;; and :set t
            :accessor borrows
            :inverse borrower))
  (:metaclass persistent-standard-class))
```

```
(defclass patron ()
  ((name :allocation :persistent
         :initarg :name
         :reader patron-name)
   (borrows :allocation :persistent
            :type (set-of book)        ;; :type is used to specify a set
             :accessor borrows
            :inverse borrower))
  (:metaclass persistent-standard-class))
```

Set-valued slots are never unbound. If there are no values stored, the value of the slot is `nil`. You can specify `:set t` and not specify a type. In that case, any Lisp object can be in the set.

What type of values can be stored in slots

The values stored in the persistent slots of an object are encoded on the disk in a manner that is independent of the Lisp that stored them. This allows other Lisp processes to access the database and read the slot values. As a result of this encoding, some information is lost.

For example: if you store a list of numbers in a persistent slot, and then access that slot, you will get back a list that is **`equal`** (not **`eq`**) to the original list that you stored in the slot.

Another consequence of this encoding is that it only makes sense to store a restricted set of types of objects in the database. The following table lists those types. Users can add additional allowable types (as we describe below the table).

Type of the value to be stored in the slot	Notes
`integer`	fixnums or bignums can be stored. Note that bignums are stored as program-defined types (which are described at the end of the table below).
`float`	Only double floats are stored in the database. If a program attempts to store a single float, the single float is converted to a double float (silently) and the double float is stored.
`symbol`	Only part of symbols are stored. Specifically, the symbol name and package are stored The property list, symbol-value, and function value are not stored.
`nil`	The symbol `nil` can be stored.
`list`	Non-circular lists can be stored. The values in the lists should be objects that are capable of being stored in the database.
`simple-vector`	A simple vector of Lisp objects (of the types listed in this table) can be stored in the database.
`string`	Simple strings can be stored in the database.
`array`	Arrays with element type t, bit, (unsigned-byte 1, 8, 16, 32), and (signed-byte 8, 16, 32). If the element type is t, elements must be types in this table. 0-dimensional arrays cannot be stored. The fact that an array is adjustable is stored, as is the fact there is a fill pointer and the fill pointer value (the whole array is stored). Raw arrays (all arrays with :element-type specified as an unsigned-byte or signed-byte type) are stored more efficiently and accessed and set much faster than type T arrays.
`character`	Characters can be stored.
`class`	The name of the class and the fact that it is a class object are both stored. In order for Lisp to retrieve a stored class object successfully, the class definition must exist in the image doing the retrieval or be in the database schema.
`hash-table`	Hash tables can be stored in the database. But note that the cost of encoding the hash table to store and decoding to read, along with the space used and the time to transfer make this an undesirable option. See section 6.7 **Persistent hash tables** below for another alternative.
`pathname`	Pathnames can be stored.
persistent object	Persistent objects in the same database.

Table 6.1: Lisp types that can be stored in a database

Program-defined types	Additional types can be stored when methods for storing them are defined by the programmer. See the information under the heading **Program-defined types** below.

Table 6.1: Lisp types that can be stored in a database

Program-defined types

You may wish to have additional types stored in the database. To do this, you must define methods on the generic functions **`astore::encode-in-database`** and **`astore::decode-from-database`** telling Lisp how to encode and decode the types. The standard way to do this is to figure out how to encode the value, perhaps in a string, perhaps a list, or in some other type that can be stored. Reading back must then decode the coded data.

Note that the type information is not stored in the database. The Lisp image must know how the type is defined. This is not a problem for standard Lisp types but if you have defined the type, you must ensure the Lisp image knows about it (typically by evaluating the definition).

Bignums are already a type that can be stored in the database (see the `integer` entry in the table above), but if they were not, the code below would add them. We show it because it is a simple example that gives the flavor of how to add more complicated types.

Here are the methods to store and retrieve bignums. (Again, bignums are already an acceptable type. This code is for illustration only.).

```
;; This method returns two values, the first being some symbol
;; and the second being something that can be stored in the database
(defmethod astore::encode-in-database ((object integer))
  (values 'read-from-string (write-to-string object :radix t :base 10)))

;; The decode method specializes on the symbol you decided to return
;; as the first value in the encode method:
(defmethod decode-from-database ((kind (eql 'read-from-string)) val)
  (read-from-string val))
```

Now let us look at a more complicated example. Suppose we define the `newborn` structure (with **`defstruct`**):

```
(defstruct newborn
  (pounds :type integer)
  (ounces :type integer)
  sex)
```

So, how to encode a newborn? One way is a list of 3 items, two integers and the value of sex (which will be the symbol `m` or the symbol `f`). This will work since all those types can already be stored in the database (they are all in the table above). There are other ways to encode as well (write the values to a string, or in a vector).

Remember that the **`astore::encode-in-database`** must return two values, the first a symbol (which typically names the type) and the second the data.

`astore::decode-from-database` must specialize using the first value returned by the encode method. We will encode on the symbol `newborn` and decode with `(eql 'newborn)` -- that rather than on `newborn` because `newborn` is not defined as a class.

In the following transcript show us doing the correct definitions, setting up a database and trying (without defining the necessary methods) to store a newborn. That fails (with the error message shown). We then define the methods and things work.

```
cl: (use-package :astore)
T
cl: (defstruct newborn (pounds :type integer)
                    (ounces :type integer) sex)
NEWBORN
cl: (setq baby (make-newborn :pounds 7 :ounces 4 :sex 'm))
#S(NEWBORN :POUNDS 7 :OUNCES 4 :SEX M)
cl: (setq *db* (open-database "fam.db"))
#<db /db/mjm/fam.db active (1)>
cl: (defclass family ()
           ((name :allocation :persistent :initarg :name)
            (members :allocation :persistent :set t :initarg :members))
           (:metaclass persistent-standard-class))
#<PERSISTENT-STANDARD-CLASS FAMILY>
cl: (with-transaction ()
           (make-instance 'family :name "The Smiths"
                          :members (list baby)))
Error: Cannot encode object #S(NEWBORN :POUNDS 7 :OUNCES 4 :SEX M) of type
       NEWBORN in the database
  [condition type: ALLEGROSTORE-ERROR]

Restart actions (select using :continue):
 0: Rollback the current transaction, returning nil
 1: Rollback the current transaction, and restart it
[1] cl: :cont 0
NIL
cl: (defmethod astore::encode-in-database ((object newborn))
            (values 'newborn
                    (list (newborn-pounds object)
                          (newborn-ounces object)
                          (newborn-sex object))))
#<STANDARD-METHOD ALLEGROSTORE::ENCODE-IN-DATABASE (NEWBORN)>
cl: (defmethod astore::decode-from-database ((kind (eql 'newborn)) val)
           (make-newborn :pounds (first val) :ounces (second val)
                         :sex (third val)))
#<STANDARD-METHOD ALLEGROSTORE::DECODE-FROM-DATABASE ((EQL NEWBORN) T)>
cl: (with-transaction ()
           (make-instance 'family :name "The Smiths"
                           :members (list baby)))
#<FAMILY in /db/mjm/fam.db (DEAD-POINTER) @ #xb28e8a>
cl: (with-transaction ()
            (for-each ((f family))
                      (format t "Family members ~S~%"
                              (slot-value f 'members))))
Family members (#S(NEWBORN :POUNDS 7 :OUNCES 4 :SEX M))
NIL
cl: (close-database *db*)
NIL
```

Things will work fine as long as you are in this Lisp image. But another Lisp image must have the newborn structure defined, or it will not be able to read the members slot. We try this in a new Lisp image:

```
cl: (use-package :astore)
T
```

```
cl: (setq *db* (open-database "fam.db"))
#<db /db/mjm/fam.db active (1)>
;;  We now try to examine the members slot of our family. It fails
;; because there is no method to read a newborn in this Lisp image.
cl: (with-transaction ()
            (for-each ((f family))
                      (format t "Family members ~S~%"
                              (slot-value f 'members))))
Error: An object of type NEWBORN was stored in the database but there is no
       decode-from-database method to decode it
  [condition type: ALLEGROSTORE-ERROR]

Restart actions (select using :continue):
 0: Rollback the current transaction, returning nil
 1: Rollback the current transaction, and restart it
[1] cl: :pop
;; So we try to define the encode method, and it fails because
;; there is no newborn type.
cl: (defmethod astore::encode-in-database ((object newborn))
             (values 'newborn
                     (list (newborn-pounds object)
                           (newborn-ounces object)
                           (newborn-sex object))))
Error: No class named: NEWBORN.
  [condition type: PROGRAM-ERROR]

Restart actions (select using :continue):
 0: Try finding the class again
[1c] cl: :pop
;;  So we define newborn, and the encoding and decoding.
;;  Now things work:
cl: (defstruct newborn (pounds :type integer) (ounces :type integer) sex)
NEWBORN
cl: (defmethod astore::encode-in-database ((object newborn))
             (values 'newborn
                     (list (newborn-pounds object)
                           (newborn-ounces object)
                           (newborn-sex object))))
#<STANDARD-METHOD ALLEGROSTORE::ENCODE-IN-DATABASE (NEWBORN)>
cl: (defmethod astore::decode-from-database ((kind (eql 'newborn)) val)
            (make-newborn :pounds (first val) :ounces (second val)
                          :sex (third val)))
#<STANDARD-METHOD ALLEGROSTORE::DECODE-FROM-DATABASE ((EQL NEWBORN) T)>
cl: (with-transaction ()
            (for-each ((f family))
                      (format t "Family members ~S~%"
                              (slot-value f 'members))))
Family members (#S(NEWBORN :POUNDS 7 :OUNCES 4 :SEX M))
NIL
cl: (close-database *db*)
NIL
```

The right thing to do is place all the definitions, encoding and decoding methods into a file that is read into Lisp before the database is opened.

6.6.1 Caching Persistent Slot Values

If you access persistent slot values multiple times during transactions, you should be able to improve performance by using either manual caching or automatic caching, which is turned on by using the new :cached persistent slot definition keyword.

This section has the following headings:

When will caching improve performance?

How does caching improve performance?

Manual caching

Automatic caching

Manual caching vs. Automatic caching

What about write caching?

When stale caches are reinitialized

Slot values other than CLOS instances are eq with automatic caching

When will caching improve performance?

Caching should improve performance when you average 2 slot read accesses per slot per instance during a transaction. For example, if you process 10,000 instances in a transaction, accessing slot A's value 4 times in 4,000 instances, 1 time in 5,000 instances, and not at all in 1,000 instances, then caching should improve performance, because ((4 * 4,000) + 5,000)/10,000 = 2.1.

A slot access is any operation the reads a persistent instance's persistent slot.

Note that writing a slot value many times during a transaction may cancel out this benefit. See the write caching section below for more information.

How does caching improve performance?

AllegroStore does not allocate memory for persistent slots when allocating a persistent instance. Every time a slot-value read is made, foreign function calls are made to retrieve the slot value. Since the database page is locked while a transaction is active, you can be certain that only you can change the slot's value. If the returned value is cached in the instance, than the cached value may be used during the rest of the transaction, resulting in performance improvements because foreign function calls are then avoided. If a slot's value is accessed many times during a transaction, the performance improvements will be significant.

Manual caching

Manual caching is done using parallel transient slots. See the Astore 1.3 manual's 6.18 Persistent Object Check Out and Check In section for an example of how to cache using transient slots.

Automatic caching

Automatic caching is done by including :cached t in the persistent slot definition, as in:

```
(defclass foo ()
  ((id :allocation :persistent :accessor id :initarg :id :cached t)
   (id2 :allocation :persistent-class
        :accessor id2 :initarg :id2 :cached t)
   (id3 :allocation :persistent :accessor id3 :initarg :id3))
  (:metaclass persistent-standard-class))
```

Manual caching vs. Automatic caching

Automatic caching is easy to use because you don't have to change anything other than the persistent class definition.

Manual caching requires more work, and you have to take care of any issues related to cache value staleness. See automatic cache refresh section below for more information about cache value staleness. With manual caching, you can decide when to use caching and when to not; with automatic caching, it is all or nothing. Depending on your schema design and application's instance processing patterns, you may be able to achieve the best performance by using manual caching, if you are willing to do the work - this is especially true if you write a slot's value many times during a transaction (see the information under the next heading). For many applications, automatic caching can significantly improve performance without requiring new design/test/debug cycles.

Be aware that the overhead associated with automatic caching can be avoided only by not specifying :cached t for all slots in a persistent class.

What about write caching?

When using automatic caching, slot value writes automatically update the backing cache, but also use foreign function calls to update database memory. That means that if you write a slot's value many times during a transaction, you do not gain performance improvement, and you pay a cost because you are constantly updating the cache. The overall effect depends on whether you read the value more often than write the value.

Automatic caching does not cache write operations because subsequent inverse function calls or query operations will not reflect the changes made during the transaction. If you use manual caching, you are free to cache write operations, keeping in mind that query operation results will reflect the database state BEFORE the current transaction started.

When stale caches are reinitialized

Automatic caching automatically reinitializes a cache value for a persistent instance's cached slot under these conditions:

- **`delete-instance`** has been called on any instance since the last time the cache value has been accessed
- the cache value was last set in a previous transaction than the current one
- **`slot-svref`** or **`(setf slot-svref)`** has been called on the slot in question
- **`slot-cons`** has been called on the slot in question
- **`slot-delete`** has been called on the slot in question

Slot values other than CLOS instances are eq with automatic caching

As long as the cache value does not become stale (see above section), non-CLOS instance slot values (such as lists) are eq. Without caching, accessing such slots' values return lists that are equalp.

6.7 Persistent hash tables

Hash tables are data objects used to store information where the lookup key is another data object. AllegroStore implements a special type of hash table called a *persistent hash table*. Before describing how to use a persistent hash table, we will describe our motivation for building it.

Why use persistent hash tables?

Lisp objects in the database are stored in a specially encoded form. In order to perform a Lisp operation on an object in the database, it first must be converted into a normal Lisp object in Lisp's memory. Three steps are required to change the value of an object in the database and have that change reflected in the database:

1. the object must be converted into a Lisp object in Lisp's memory
2. the value of the Lisp object must be changed
3. the modified object must be stored back into the database

The cost to convert from database format to Lisp format and back to database format again is small for small objects. However, the conversion cost is significant for an object as large as a hash table. To alleviate this problem, we've implemented hash tables that can be used without being first converted into an object in Lisp's memory.

Properties of persistent hash tables

A persistent hash table is like an equal hash table, except that the table is always stored in the database. The keys and values of the hash table are persistent values.

Creating and manipulating persistent hash tables

A persistent hash table is created with **`make-instance`**:

```
(make-instance 'persistent-hash-table)
```

A persistent hash table is operated on by the functions **`generic-gethash`**, **`generic-remhash`** and **`generic-maphash`**, which take arguments similar to the corresponding standard Common Lisp functions (**`gethash`**, **`remhash`** and **`maphash`**).

See section 7.2.10 **Persistent hash tables** for more information.

6.8 Blobs

Blobs are a persistent class you can use to allocate raw persistent storage without creating a parallel Lisp object.

Why use blobs?

If you need large blocks of persistent storage that you can treat as C memory, accessing and setting the memory using the ACL foreign types functionality, then blobs provide the most efficient way to generate and manipulate that storage.

Like persistent hash tables, blobs allow you to change a part of the allocated persistent memory without converting a large database object to a corresponding Lisp object, making the change, and then converting the modified Lisp object back into database format.

Properties of blobs

Blobs contain two properties - a name and a data pointer. The name can be optionally used to retrieve the blob directly when it is not accessible in a slot of some other persistent class instance. The data pointer contains an address that can be used as an argument to foreign types functions to access or modify the blob's persistent memory.

Manipulating a blob data pointer with foreign types functionality is a low level operation, so you must take into account architectural dependencies. For example, some architectures require that an integer pointer start at a word (4 byte) boundary. Users wishing to access blobs from multiple architectures must also account for big-endian/little-endian differences when reading and writing blob data.

Creating and manipulating blobs

Here is an optimized example that allocates a 100 byte blob, and then stores the integers from 0 to 100 in those bytes:

```
(defclass thing ()
   ((slot :allocation :persistent))
   (:metaclass persistent-standard-class))

(with-database (db "foo.db" :if-exists :supersede)
   (with-transaction ()
      (let ((foo (make-instance 'thing)) pptr)
         (setf (slot-value foo 'slot) (make-instance 'blob :size 100))
         (setf pptr (blob-data (slot-value foo 'slot)))
         (dotimes (i 100 nil)
            (setf (ff:fslot-value-typed '(:array :unsigned-char) :c
                (+ pptr (* i (ff:sizeof-fobject :unsigned-char))) 0) i)))))
```

Here's code that will read them and print them out at a later time:

```
(with-database (db "foo.db")
   (with-transaction ()
      (let* ((thing (first (retrieve 'thing)))
             (pptr (blob-data (slot-value thing 'slot))))
         (dotimes (i 100)
            (print (ff:fslot-value-typed '(:array :unsigned-char) :c
                 (+ pptr (* i (ff:sizeof-fobject :unsigned-char))) 0))))))
```

See section **7.2.11 Blobs** for more information.

Blobs and files

You may use blobs to store and access files.

To store a file as a blob, you need to know the number of bytes in the file and the file name. Here is an example:

```
(with-database (db "foo.db")
   (with-transaction ()
      (let ((b1 (make-instance 'blob :name "file1" :size 40000000)))
          (blob-read b1 "file1.data")))) ;; file1.data is the file name
```

Here is how to write a blob out to a file:

```
(with-database (db "foo.db")
    (with-transaction ()
       (let ((b1 (first (retrieve 'blob
                          :where '((blob-name equal "file1"))))))
          (blob-write b1 "file1.data.copy"))))
```

6.9 Persistent Ftype (Foreign Type) Arrays

Persistent ftype arrays, like blobs, is a persistent class you can use to allocate raw persistent storage without creating a parallel Lisp object. They are more flexible than blobs because their raw persistent storage may contain pointers to other persistent objects or memory areas, which is not possible with blobs.

This section has the following headings:

Why use persistent ftype arrays?

When are persistent ftype arrays the wrong choice?

Persistent ftype array properties

Creating and manipulating persistent ftype arrays

Special Note - getting the best foreign type access performance

Using pointers to persistent Lisp objects

Dynamically determining foreign type definitions

Freeing persistent foreign type array memory

Discarding unneeded foreign types from a database

Using tags to retrieve an initial persistent address

The operators and variables associated with this facility are described in section 7.2.12 in chapter 7 *Reference Guide*.

Why use persistent ftype arrays?

If you have large amounts of data that may be represented by C structures, then persistent ftype arrays provide better performance than persistent CLOS instances. Performance gains are possible because parallel Lisp objects are not maintained, foreign function calls are not needed to access and set C foreign structure slots, and persistent memory fragmentation can be significantly reduced.

You access and set persistent ftype array data using ACL foreign type functionality, thereby accessing the persistent memory directly in the ObjectStore persistent address space. This reduces the demand on Lisp heap memory, because parallel Lisp objects are not required, and the overhead associated with ObjectStore->Lisp and Lisp->ObjectStore marshalling is avoided. Also, persistent ftype array data is not managed by the AllegroStore referential integrity system, which again reduces operating overhead. Unlike persistent CLOS instances, where accessing a single instance may require many different ObjectStore mem-

ory pages, persistent ftype array data is laid out contiguously in memory, thereby improving underlying ObjectStore performance.

In the examples below, using persistent ftype arrays resulted in a 39% speedup during object creation, a 72% speedup during an object traversal heavy calculation, and a 41% speedup during a simple iteration step.

Persistent ftype arrays also provide a simple mechanism for sharing data with ObjectStore C++ or Java programmers. Note that for complex applications, a more robust protocol such as CORBA may be necessary for effective object sharing.

When are persistent ftype arrays the wrong choice?

Once a persistent ftype is defined in a database, that definition can never be changed. If you have a dynamic application that changes class definitions during run time, then it is not a good idea to replace such a class with a persistent ftype. Note, however, that you can define new foreign types dynamically during run time.

Persistent foreign type arrays contain foreign data such as integers, floating point values, strings, and pointers. If you can't represent your objects using these atomic types, then you may have to stick to persistent CLOS instances. Note, however, that persistent foreign type array data can contain pointers to persistent CLOS instances.

Using persistent foreign type arrays requires greater programming care. Programming with foreign types usually involves "pointer arithmetic" calculations - if your code that does such calculations contains bugs, you may corrupt your database and crash the Lisp. In essence, you face the same opportunities and challenges ObjectStore C++ programmers face - good performance is attainable, but a buggy program causes much more damage and usually requires more time to debug.

Since the AllegroStore referential integrity system does not cover persistent ftype array data, you must manage that data yourself. For example, if a pointer in persistent ftype array data points at a persistent CLOS instance, and that instance is subsequently deleted, AllegroStore will not transparently change the pointer value to 0. If the pointer is subsequently dereferenced, it may point at an unexpected data element, or it may point at memory outside the current ObjectStore managed address space. Writing to the address in such a situation may corrupt the database or crash the Lisp. Again, this is the same challenge that ObjectStore C++ programmers face.

Persistent ftype array properties

Persistent ftype array instances contain four properties - a name property that can be optionally used to retrieve the instance directly when it is not accessible in another way, a number of elements property that reports how large the data array is, a type property that specifies which foreign type has been allocated, and a data property, which contains a pointer that is used to access or set the array's persistent memory.

Creating and manipulating persistent ftype arrays

To show how to use persistent ftype array, we will define a simple task, solve it first using persistent CLOS instances, and then solve it again using persistent ftype arrays.

The task:

Given a list of 10,000 random numbers between 0 and 99, create a database containing one object for each random number and an array of 100 objects, one for each integer between 0 and 99. Each random number object must point to the appropriate array element, and each array element must contain the number of random number objects that point to it.

A persistent CLOS solution:

```
;; there will be 100 of these
(defclass random-element-group ()
  ((value :allocation :persistent :accessor value :initarg :value)
    (sum :allocation :persistent :accessor sum :initform 0))
  (:metaclass persistent-standard-class))
;; there will be 10,000 of these
(defclass random-element ()
  ((value :allocation :persistent :reader value :initarg :value)
   (group :allocation :persistent
          :type random-element-group :accessor group))
  (:metaclass persistent-standard-class))
;; there will be one of these - the array will
;; contain the 100 random-element-group instances
(defclass random-element-group-array ()
  ((groups :allocation :persistent :accessor groups :initarg :groups))
  (:metaclass persistent-standard-class))
 ;; generate the random number objects
;; random-numbers is a special variable
;; containing the 10,000 random numbers
(defun generate-random-elements ()
  (with-database (db "random.db" :if-exists :supersede)
    (with-transaction ()
      (declare (type fixnum i))
      (dotimes (i 10000)
         (make-instance 'random-element :value (nth i random-numbers))))))
(defun generate-summary ()
  (with-database (db "random.db")
    (with-transaction ()
      (let ((group-array (make-array 100 :element-type 'random-element-
group)))
;; create the 100 instances corresponding to 0-99
        (dotimes (i 100)
          (declare (type fixnum i))
          (setf (aref group-array i)
                (make-instance 'random-element-group :value i)))
        (make-instance 'random-element-group-array :groups group-array)
;; for each random-element object, set the group
;; slot appropriately and increment the group sum
        (for-each ((object random-element))
          (setf (group object) (aref group-array (value object)))
          (incf (sum (group object))))
         t))))
 ;; print out the sums
(defun print-summary ()
  (with-database (db "random.db")
    (with-transaction ()
      (let ((group-array
              (groups
                (first (retrieve 'random-element-group-array)))))
        (dotimes (i 100)
          (declare (type fixnum i))
          (format t "~s: ~s~%" i (sum (aref group-array i))))
        t))))
```

A persistent foreign type solution:

```
;; there will be 100 of these
(def-foreign-type random_element_group (:struct
                                        (value :int)
                                        (sum :int)))
;; there will be 10,000 of these
(def-foreign-type random_element (:struct
                                  (value :int)
                                  (group (* random_element_group))))
;; subclassed so we can distinguish from array
;; containing random_element objects
(defclass random-element-group-array (persistent-ftype-array)
  ()
  (:metaclass persistent-standard-class))
 (defun generate-random-elements ()
  (with-database (db "random.db" :if-exists :supersede)
    (with-transaction ()
      ;; add the persistent ftypes we need to the database
      (add-persistent-ftype (list 'random_element 'random_element_group))
      ;; allocate the memory for the random element
      ;; objects and get the pointer
     (let* ((random-elements-array (make-instance 'persistent-ftype-array
                :type 'random_element :n 10000))
               (random-elements-data (persistent-ftype-array-data
                random-elements-array)))
        (declare (type fixnum random-elements-data))
;; we do the pointer arithmetic ourselves for
;; optimal performance - SEE SPECIAL NOTE BELOW
        (let ((address random-elements-data)
              (increment (sizeof-fobject 'random_element)))
          (declare (type fixnum address increment))
          (dotimes (i 10000)
            (declare (type fixnum i))
            (setf (fslot-value-typed '(:array random_element 1) :c
                        address 0 'value (nth i some-numbers))
            (incf address increment))
          )))))
(defun generate-summary ()
  (with-database (db "random.db")
    (with-transaction ()
      (let* ((group-array (make-instance 'random-element-group-array
                    :type 'random_element_group :n 100))
             (group-array-data (persistent-ftype-array-data group-array))
              (random-elements-data
                 (persistent-ftype-array-data
                    (first (retrieve 'persistent-ftype-array)))))
        (declare (type fixnum group-array-data random-elements-data))
;; set up the 100 objects related to 0-99
        (let ((address group-array-data)
              (increment (sizeof-fobject 'random_element_group)))
          (declare (type fixnum address increment))
          (dotimes (i 100)
            (declare (type fixnum i))
            (setf (fslot-value-typed
                  '(:array random_element_group 1) :c address 0 'value) i)
            (incf address increment)))
         (let ((address random-elements-data)
               (increment (sizeof-fobject 'random_element)))
```

```
            (declare (type fixnum address increment))
    ;; set the group slot appropriately and increment the sum
            (dotimes (i 10000)
              (declare (type fixnum i))
              (let ((group-address
                  (fslot-address-typed '(:array random_element_group 1) :c
                         (+ group-array-data
                           (* (fslot-value-typed
                           '(:array random_element 1) :c address 0 'value)
                                                              (sizeof-
fobject 'random_element_group))) 0)))
                     (declare (type fixnum group-address))
                    (setf (fslot-value-typed '(:array random_element 1) :c
address 0 'group) group-address)
                    (incf (fslot-value-typed '(:array random_element_group
1) :c group-address 0 'sum))
                     (incf address increment))))
              t))))
 (defun print-summary ()
  (with-database (db "random.db")
    (with-transaction ()
      (let ((group-array-data (persistent-ftype-array-data
               (first (retrieve 'random-element-group-array)))))
        (declare (type fixnum group-array-data))
        (let ((address group-array-data)
              (increment (sizeof-fobject 'random_element_group)))
          (declare (type fixnum address increment))
          (dotimes (i 100)
            (declare (type fixnum i))
            (format t "~s: ~s~%" i (fslot-value-typed '(:array
random_element_group 1) :c address 0 'sum))
            (incf address increment)))
        t))))
```

Special Note - getting the best foreign type access performance

In ACL 5.0 and ACL 5.0.1, the Lisp compiler can inline foreign type memory access calls to achieve optimal performance. However, there is a current limitation when using arrays - the compiler will not inline array accesses where the array access offset is not a constant.

For example,

```
(dotimes (i 100)
   (setf (fslot-value-typed '(:array random_element 1)
           :c random-elements-data i 'value)  (nth i some-numbers)))
```

will not be inlined.

```
(let ((address random-elements-data)
       (increment (sizeof-fobject 'random_element)))
   (dotimes (i 10000)
      (setf (fslot-value-typed '(:array random_element 1) :c address 0
'value) (nth i some-numbers))
      (incf address increment)))
```

will be inlined.

The performance gains associated with inlining are significant - in the above example, the inlined version will run more than twice as fast as the non-inlined version.

Using pointers to persistent Lisp objects

To use pointers to persistent Lisp objects:

1. Use a (* :void) structure member type for the structure member that will contain a pointer to a persistent Lisp object.
2. Use the astore::lisp-value-to-pptr function to obtain a persistent address for the Lisp object you wish to point to, as in:

```
(astore::lisp-value-to-pptr obj)
```

3. Use the astore::pptr-to-lisp-value function to transform a persistent pointer to a Lisp object into its Lisp analog, as in:

```
(astore::pptr-to-lisp-value address *db*)
```

While lisp-value-to-pptr will work with any Lisp object that can be stored in a persistent instance slot, it is a good idea to limit usage to persistent CLOS instances. You will get better performance if you handle strings, integers, floating point values, and arrays using foreign types, and other Lisp objects such as lists and symbols may be represented using the basic foreign types. Also, currently, if you call lisp-value-to-pptr on an object other than a persistent CLOS instance, an object will be created in ObjectStore that you won't be able to later delete.

Remember that AllegroStore referential integrity does not cover persistent foreign type array data. If you delete a persistent CLOS instance with delete-instance, it is your responsibility to insure that no pointers in persistent foreign type array data point at the address that is now deleted.

Dynamically determining foreign type definitions

You can use for-each or for-each* to iterate over all persistent-ftype-array and persistent-ftype-array subclassed instances, and use the persistent-ftype-array-type method to retrieve the foreign type associated with that instance. Then, given a foreign type of interest, use the describe-ftype function, as in:

```
(describe-ftype "mydb.db" 'my_struct)
```

A string will be returned that contains the foreign type definition. Note that when dealing with type symbols, the AllegroStore persistent ftype array facility ignores packages; thus, you can only store one definition called 'my_struct.

Freeing persistent foreign type array memory

If you call delete-instance on a persistent-ftype-array instance, the persistent memory associated with the instance will also be freed. It is your responsibility to insure that pointers in remaining persistent-ftype-array instance data do not point to memory that is now deleted.

Discarding unneeded foreign types from a database

Stored foreign type definitions cannot be removed from an existing database. However, if you delete all persistent-ftype-array instances that refer to a given foreign type, dump the database using **asdump**, and then create a new database from the dump file using **asrestore**, the resulting database will not contain the unneeded foreign type.

Using tags to retrieve an initial persistent address

An alternate way to retrieve the initial persistent address to be used for subsequent navigation involves database tags. Tags allow you to directly retrieve a persistent address without first retrieving a persistent-ftype-array instance and then using the persistent-ftype-array-data method. Here is an example:

Setting the tag:

```
(def-foreign-type foo_struct (:struct (f1 :int)))
(defun test-tag ()
   (with-database (db "foo.db" :if-exists :supersede)
      (with-transaction ()
         (add-persistent-ftype 'foo_struct)
         (let* ((foo (make-instance 'persistent-ftype-array
                        :type 'foo_struct :n 1))
                (foo-data (persistent-ftype-array-data foo)))
            (setf (fslot-value-typed '(:array foo_struct 1) :c
                        foo-data 0 'f1) 123)
            (set-dbtag "xxx" foo-data)))))
```

Using the tag:

```
(defun verify-tag (db-string)
   (with-database (db db-string)
      (with-transaction ()
         (let ((data (get-dbtag "xxx")))
            (when (= data 0)
               (error "couldn't find tag"))
            (when (/= (fslot-value-typed '(:array foo_struct 1) :c
                                 data 0 'f1) 123)
                (error "value didn't retrieve correctly"))))))
```

Note that a well-behaved database should contain no more than a few dozen tags.

Tags are an easy way to make persistent foreign type array data available to ObjectStore C++ or Java programmers. The relevant methods are os_database_root::find() and os_database_root::get_value().

6.10 Inverse functions

Readers and accessors

When a slot is defined with **`defclass`** a functions (actually a methods) can be created that will read and write the value of that slot from an object. Thus, the `:reader` option defines a reader method (reads the value) and the `:writer` option defines a writer method (sets the value). The `:accessor` option provides a shorthand for supplying both at once (with the name supplied providing the reader and *(setf <name supplied>)* providing the writer). See the description of **`defclass`** in the ANSI spec for a complete description.

Consider the following example, which uses the tire database:

```
(defclass tire ()
   ((brand-name :allocation :persistent :accessor tire-brand-name))
   (:metaclass persistent-standard-class))
```

The **defclass** for tire defines the method **tire-brand-name**. Given a tire object, **tire-brand-name** will return the value of the brand-name slot. **(setf tire-brand-name)** will set the value of the brand-name slot, typically with a form like

```
(setf (tire-brand-name tire-instance) new-brand-name)
```

The problem of finding an object given a slot value

Suppose we have a value that may be stored in the brand-name slot of one or more tire objects, and we want to find those tire objects. For example, we want to find all the tires in our database with brand-name michelin. The naive, brute force way to do this is to examine the brand-name slot of every tire object, and if the value is michelin, add it to our list. (We can use the query facility, described in section 6.10, to examine every tire object.)

The problem with this algorithm is, if the database gets large, it can take a long time to examine every object. To solve this problem, AllegroStore has an :inverse option to a slot definition. If this option is specified and given a symbol as an argument, that symbol will name an inverse function that, given a value, will return all objects of the class whose slot contains the specified value.

Inverse functions

To repeat, if the :inverse slot option is specified with a symbol name as an argument, that symbol will name a function that returns all instances of a class with a specified value in the slot. In our tire, example, we could define the class as follows:

```
(defclass tire ()
   ((brand-name :allocation :persistent :accessor tire-brand-name
      :inverse brand-name-tire))
   (:metaclass persistent-standard-class))
```

brand-name-tire, when passed a symbol naming a brand name (like michelin) will return a list of all tire objects that have brand-name michelin.

Inverse functions are really *methods* specialized on the type of the slot they are inverting. In the tire example, the type of the slot is not specified (with the :type option), so the method specializes on type t.

Note that only :persistent slots can have :inverse specified. :persistent-class slots cannot.

Developers examining persistent class definitions with metobject protocol methods will see one extra slot that AllegroStore adds for each slot that has an associated inverse function. The added slot's name will be:

```
.inverse-table--%class%-%slot%
```

where %class% is the class name, and %slot% is the slot name.

Inverse functions speed up querying, but may cost

An inverse function will return the answer much more quickly than a query over a collection of objects. However, there is a trade-off. Maintaining the inverse relation table can be expensive in certain cases.

Virtually no cost when slot :type is a persistent object: if the type of the slot in which the inverse function is defined is a persistent class, then there is no cost to having an inverse function -- since all persistent objects keep track of those objects that point to them.

Can be a cost when slot :type is unspecific or a non-persistent type: if the type of the slot includes non-persistent objects (and particularly when no `:type` option is specified in the slot definition), then a table must be built that holds the inverse information, and that table is updated with each store into the slot. It is those tables which can become large. The extra space in memory for the tables must be balanced against the extra space in the database for additional persistent data.

If you can guarantee that all slot-values will be fixnums, you can improve performance by specifying `:type integer` in the persistent slot definition. Note, however, that if you specify `:type integer` and then store a value other than a fixnum, you will experience undesirable results.

Unique slot values

Many tires can be Michelin tires, so if the system has found one such tire, it must keep looking to see if there are more. Some slot values, however, are unique. A serial number, for example, is (typically) unique. Once you have found an object with a given serial number, you need look no further. If you know that the value in a certain slot will be different in each object, then you should specify the slot option `:unique t` as in this form for the serial number of an automobile:

```
(defclass auto ()
  ((serial-number :inverse number-to-auto :unique t))
  (:metaclass persistent-standard-class))
```

Inverse functions ordinarily return a list (even if there is only one element). However, specifying `:unique t` makes the inverse function (**`number-to-auto`**, in this case) return either `nil` or exactly one value. It won't return a list of one value as it would if `:unique t` were not given.

Specifying a slot `:unique` imposes a cost when storing, since the system does check that no other existing slot has the specified value. It reduces the cost of fetching (as we said above). A `:unique` slot can not also be a set slot.

Remember, inverse functions are actually methods specialized on the declared type of the slot. So if this code is evaluated:

```
(defclass auto ()
   ((serial-number :type integer :inverse number-to-auto
      :unique t))
   (:metaclass persistent-standard-class))
```

then the following method:

```
(defmethod number-to-auto ((x integer)) ...)
```

will be defined.

There is one consequence of the fact that **`number-to-auto`** is a generic function: if you call **`number-to-auto`** on an object for which the generic function is not specialized, (like `(number-to-auto 'foo)`), you get a `method not found` error message, because **`foo`** is not an integer.

6.11 Queries and iterators

Query functions enable the user to access objects from that database that satisfy certain criteria. There are two types of query functions:

- those that allow the program to iterate over all objects in the database satisfying a certain condition (the *iterators* like **`for-each`** and **`for-each*`**)
- those that return a list of all objects satisfying a certain condition (the *non-iterators*, like **`retrieve`**)

The iterator forms should be used whenever possible, since they allow you to operate on objects without storing all of them simultaneously in Lisp's memory.

Iterators

The three iterators are **`for-each`**, **`for-each*`** and **`for-each-class`**.

The for-each macro

`for-each` is a macro that permits you to create nested iterators.Consider the following class definitions:

```
(defclass auto ()
  ((name       :allocation :persistent :accessor auto-name)
   (model-year :allocation :persistent :accessor auto-model-year)
   (price      :allocation :persistent :accessor auto-price)
   (wheels      :allocation :persistent :set t :accessor auto-wheels))
    (:metaclass persistent-standard-class))

(defclass wheel ()
     ((brand :allocation :persistent :accessor wheel-brand))
     (:metaclass persistent-standard-class))
```

The following form will iterate over the database and print every object whose class is `auto` or a subclass of `auto`:

```
(for-each ((a auto)) (print a))
```

The following form will again iterate over each auto, and for each auto it will iterate over each wheel of that auto:

```
(for-each ((a auto) (w (auto-wheels a)))
   (format t "auto: ~s, wheel ~s~%" a w))
```

for-each and :where

Suppose we wanted to print Firestone wheels and their associated autos. We could do so with the `:where` clause:

```
(for-each ((a auto)
           (w (auto-wheels a))
           (:where (equal (wheel-brand w) 'firestone)))
   (format t "auto: ~s, wheel ~s~%" a w))
```

The form of the first argument to the **`for-each`** macro is a set of *variable binding clauses* plus optional special clauses such as the `:where` clause shown above. A variable binding clause consists of a variable and an expression which denotes a sequence of objects. There are three valid expressions:

1. a symbol, which denotes a class name.

2. (*accessor-name* `variable`), which denotes the value of the slot returned by calling *accessor-name* on the value of the given variable. The slot being accessed must be a `:persistent` or `:persistent-class` slot and it must be a *set* slot (i.e., declared as `:set t` or `:type (set-of some-type)`).
3. (`slot-value variable 'slot-name`). (This is really the same as 2 above, for slots which do not have an accessor or reader predefined.)

The `:where` clause, if given, may contain any Lisp expression. If this expression is true, then the body of the **`for-each`** is executed. The `:where` clause may appear anywhere in the list of variable binding clauses, but it is always evaluated just before evaluating the body of the **`for-each`**.

Restricting the search to the class (no subclasses)

A **`for-each`** variable binding clause such as (`a auto`) will bind `a` to any object that is of class `auto` or a subclass of `auto`. If you wish to have it only bind `a` to objects of class `auto`, then you should add (`:subclasses nil`) to the list of variable binding clauses. `(:subclasses nil)` instructs clauses only to act on the classes specified, not on any of their subclasses. There may be more than one clause that iterates though objects of a class, and the `:subclasses` option applies to all of them.

for-each*

`for-each*` is another iterator like **`for-each`**, but it has a very different calling template and works quite differently. In a **`for-each*`** form, AllegroStore iterates over the objects of a single class (and unless otherwise specified, its subclasses), and calls a given function on objects that satisfy a special predicate clause called a where clause. Note that a **`for-each*`** where clause is quite different from a **`for-each`** where clause.

The form of a **`for-each*`** where clause is very restricted: it is a conjunction of a sequence of binary tests comparing slot accessors with constants. For example,

```
(for-each* #'(lambda (auto) (print (auto-name auto)))
           'auto
           :where '((auto-model-year >= 1990)  (auto-price > 12000)))
```

will print the names of all automobiles in the database which are 1990 models or later and which cost more than $12,000.

The `:where` argument is a sequence of lists of three items:

```
(accessor-name binary-function other-argument)
```

1. The `accessor-name`, which must be an accessor or reader for the given class.
2. The `binary-function`, which must be a function of two arguments:
   - the first argument is the result of calling the `accessor-name` on the object from the database
   - the second argument is the `other-argument`
3. The `other-argument`, which effectively must be a constant since it is not evaluated. (Experienced Lisp users will be able to have `other-argument` evaluated by back-quoting the `:where` clause and preceding the `other-argument` with a comma. Inexperienced Lisp users should stick to constants, which are in any case more efficient. Constants include, among other things, all numbers, strings, and symbols.)

for-each* also takes the :subclasses keyword argument if you want to specify that objects of subclasses of the given class aren't to be considered. See the discussion of the :subclasses argument for **for-each** above.

We should emphasize that **for-each*** is un-Lispy (particularly in the way the predicate goes between the two arguments rather than before). It is a format imported from another database system and grafted into Lisp.

for-each-class

for-each-class is an iterator macro that binds a variable to each class in the schema and then evaluates the body of the macro.

For example, to count all the objects in the database:

```
(let ((count 0))
  (for-each-class (cl)
      (for-each* #'(lambda (obj) (incf count)) cl :subclasses nil))
  (format t "There are ~d objects in the database~%" count))
```

Non-iterator: retrieve

Non-iterators

The function **retrieve** will return a list of all objects of a specified class satisfying a given predicate. Its calling template is similar to **for-each*** (without the *function* argument). The following form will return a list of 1990 or more recent model cars which cost more than $12,000.

```
(retrieve 'auto
   :where '((model-year >= 1990)  (auto-price > 12000)))
```

retrieve works by calling **for-each*** with, as the *function* argument, a function that collects the objects selected by the :where clause into a list. Note that if there are many such objects, the resulting list can be quite large and can use a fair amount of space. Always use an iterator if possible in preference to a non-iterator.

6.12 Pointers

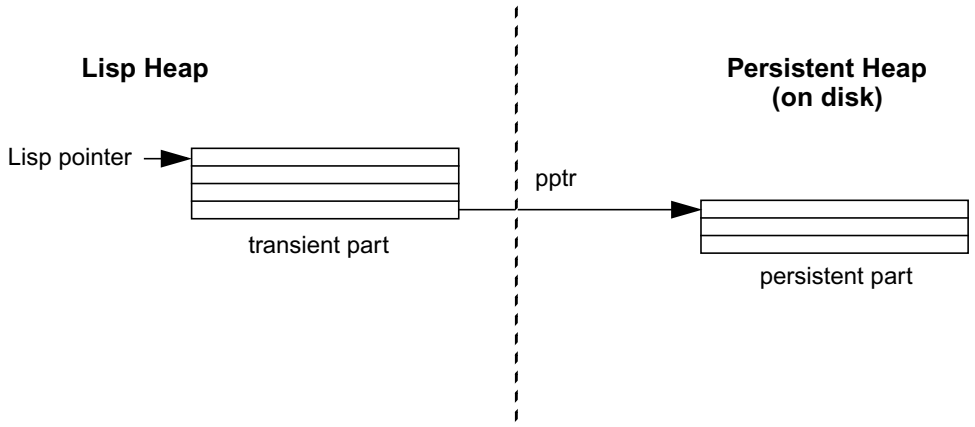
A persistent object in the database can point to other persistent objects in the same database, or to simple Lisp objects (lists, vectors, integers, or strings[1]) which are recreated in the Lisp heap when they are referenced in the database. A normal Lisp object can point to a persistent object in the database. This release of AllegroStore does not allow cross-database pointers; thus a persistent object in one database cannot point to a persistent object in another database.

If you have a pointer, which you believe to be the pointer to an object, is it valid? We discuss that point next.

The validity of pointers

A persistent object consists of two parts: one part is transient and is stored in Lisp's heap and one part is persistent and is stored in the database on disk. The transient part points to the persistent part with a *persistent pointer* (or *pptr*), as illustrated in the following diagram.

1. For a complete listing of simple Lisp objects which can be stored, see the table in the section on **What type of values can be stored in slots.**



The persistent part of an object will not change its location on the disk during a transaction: this is an important feature of transactions. Once a transaction completes, other users (or the Database Manager) may shuffle objects around on the disk, making a persistent pointer that was valid in a previous transaction now point to garbage. Thus, persistent pointers have a limited lifetime.

AllegroStore prevents programs from using invalid pointers by tagging each persistent pointer with the name of the transaction in which it is valid. The following example shows how AllegroStore detects an invalid persistent pointer.

```
(defclass employee () () (:metaclass persistent-standard-class))
(defvar *m* nil)
(with-database (db "company.db" :if-exists :supersede)
   (with-transaction nil
      (print (setq *m* (make-instance 'employee))))
   (with-transaction nil
        (print *m*)))
```

The persistent class `employee` is defined, and an object is created and assigned to the variable `*m*`. `*m*` is an instance of `employee`; it has a non-persistent part and a persistent part.

When the object is first printed, control is still inside the transaction where the persistent pointer is valid. Object `*m*` first prints like this:

```
#<employee in /acme/company.db p: #x65f123 @ #x2127g5>
```

When `*m*` is next printed, control is outside the transaction in which the persistent pointer it contains is valid. Object `*m*` next prints like this (we use a smaller type to fit it on one line):

```
#<employee in /acme/company.db p: #x65f123 (dead-pointer) @ #x2137g5>
```

The `dead-pointer` description means that the pointer inside the `employee` object `*m*` from the transient to the persistent part is invalid, because the transaction in which it *was* valid is over.

After the transaction is over, the program can still (using the value of `*m*`) access the non-persistent slots of the object, but it is an error to access the persistent slots. The persistent slots still exist in the database, they are just unreachable via the value of `*m*`.

In order to get a valid pointer to the same `employee` object, the program must start a transaction and use the query functions to locate it. You must have a valid pointer to a persistent object to access its persistent slots.

preserve-pointer

There is a way to make a persistent pointer last through multiple transactions. If, while the persistent pointer is still valid, the **`preserve-pointer`** function is called on a persistent object, AllegroStore will allocate a reference object that enables it to update the persistent pointer value in each new transaction.

For example, suppose we called **`preserve-pointer`** when we created the `employee` object:

```
(defclass employee () () (:metaclass persistent-standard-class))
(with-database (db "company.db" :if-exists :supersede)
   (with-transaction nil
      (print (setq *m* (preserve-pointer (make-instance 'employee)))))
   (with-transaction nil
        (print *m*)))
```

then what we would see printed would be:

```
#<employee in /db/acme/company.db p: #x72f0004 (ref) @ #xa126ca>
#<employee in /db/acme/company.db p: #x72f0004 (ref) @ #xa126ca>
```

In this case, the `*m*` object can be referenced in subsequent transactions and the printed representation reflects that.

Once the database is closed, the reference is no longer valid.

6.13 Implicit object creation

We mentioned above that a persistent CLOS object is composed of two parts: a transient part in Lisp's heap and a persistent part in the database. In versions of AllegroStore prior to 1.2, it was possible to have two distinct transient parts pointing to the same persistent object. As a result, the transient parts were not **`eq`**. Instead, they were **`eqo`**. Starting with version 1.16, an extension allowed programmers to ensure that **`eqo`** transient parts were also **`eq`**, as described in Technical memorandum AS-3 ‘Making transient copies of persistent objects eq’.

In version 1.2, all transient parts which should be **`eq`** are **`eq`** automatically. For this reason, Technical memorandum AS-3 does not apply to AllegroStore 1.2.

6.14 Object deletion

Persistent objects differ from normal Lisp objects in two important ways:

1. Persistent objects are *not* collected by a garbage collector. They remain in the database even when there are no references to them from other Lisp objects.
2. Persistent objects can be explicitly deleted with the **`delete-instance`** function. An object can be deleted even if other objects point to it. Section 6.14 **Referential integrity** section describes how AllegroStore handles this situation.

6.15 Referential integrity

AllegroStore maintains *referential integrity*. This means that all pointers inside the database point to valid database objects, never to places where an object, now deleted, used to be stored.

Because AllegroStore maintains referential integrity, the programmer needn't put explicit tests in his code to check each pointer reference for validity. When an object is deleted, AllegroStore finds all references and replaces them with the appropriate value. Two cases are listed next:

- If the reference comes from within a set (as in the `borrows` slot of the patron class in our library example), then the reference is simply removed.
- If the reference is from a non-set slot, then the reference is replaced with the value representing an unbound value, unless the slot's `initform` is `nil`, in which case the value is replaced with `nil`.

If the reference is not described by the two bullets above, see the definition of **`delete-instance`** in chapter 7 **Reference** for information on how the object updating is handled.

If desired, it is possible to find all the references to an object before calling **`delete-instance`**, and then employ a different strategy for replacing references. We describe that under the next heading.

Finding references to an object (collect-references)

As we described above, AllegroStore will make sure that all pointers to a deleted object in the database are eliminated once the object is deleted. This will involve the modification of data objects stored in the database.

Some applications may wish to know beforehand *which* objects will be modified if an object is deleted. The **`collect-references`** and **`map-references`** functions exist to find and process references to a persistent object.

We'll use the `auto` and `wheel` classes for this example of using references. The `wheels` slot of an auto is a set of `wheel` objects, each `wheel` object representing a brand of wheel that we can use on this automobile. Suppose we delete a wheel object from the database; we would want to know which auto instances this deletion would affect. Here are the class definitions:

```
(defclass auto ()
  ((name :allocation :persistent :initarg :name :accessor auto-name)
   (wheels :initarg :wheels :allocation :persistent :type (set-of wheel)
     :accessor auto-wheels))
  (:metaclass persistent-standard-class))

(defclass wheel ()
  ((brand :allocation :persistent :initarg :brand :accessor whell-brand))
  (:metaclass persistent-standard-class))
```

Here is a `:before` method for deleting an instance of class `wheel`. The method maps a function over each database object that refers to the `wheel` object, which is about to be deleted. If the object is an `auto` object and refers to the `wheel` object through the `wheels` slot, then the method issues a warning message.

```
(defmethod delete-instance :before ((w wheel))
  (map-references #'(lambda (obj slot-name)
                      (cond ((and (typep obj 'auto)
                                  (eq slot-name 'wheels))
                             (warn "Auto ~s is losing a wheel ~s that it uses"
                                   (auto-name obj)
                                   (wheel-brand w)))))
                  w))
```

The next function builds a database and tests the **`delete-instance`** method we wrote:

```
(defun testcase ()
  (with-database (db "inventor.db" :if-exists :supersede)
    (with-transaction ()
      (let ((ford (make-instance 'auto :name 'ford))
            (chevy (make-instance 'auto :name 'chevy))
            (mazda (make-instance 'auto :name 'mazda))
            (goodyear (make-instance 'wheel :brand 'goodyear))
            (firestone (make-instance 'wheel :brand 'firestone))
            (michelin  (make-instance 'wheel :brand 'michelin)))
        (setf (auto-wheels ford) (list goodyear firestone))
        (setf (auto-wheels chevy) (list goodyear))
        (setf (auto-wheels mazda) (list michelin firestone))

        ;; now try deleting something
        (format t "Beginning delete test~%")
        (delete-instance firestone)
        ))))
```

Running the test case looks like this:

```
cl: (testcase)
Beginning delete test
Warning: Auto ford is losing a wheel firestone that it uses
Warning: Auto mazda is losing a wheel firestone that it uses
t
cl:
```

6.16 Object update

In our examples of a library and an auto, we often decided that our class definitions were inadequate and updated them. Thus, for example, we started with the following definition of a book in the library:

```
(defclass book ()
  ((title :allocation :persistent
          :initarg :title
          :accessor title)
   (author :allocation :persistent
           :initarg :author
           :accessor author))
  (:metaclass persistent-standard-class))
```

A little later, we decided that a barcode slot would be useful, to store a unique identifier of a book (often called an inventory or accession number), so we redefined book as follows:

```
(defclass book ()
  ((title :allocation :persistent
          :initarg :title
          :accessor title)
   (author :allocation :persistent
           :initarg :author
           :accessor author)
   (barcode :allocation :persistent
            :initarg :barcode
            :accessor barcode))
  (:metaclass persistent-standard-class))
```

Now, we had instances of `book` in our database when we redefined the class. What happened to them? When the class definitions of objects in the database are changed, AllegroStore doesn't immediately update the objects. It uses a *lazy algorithm* for updating; it only updates objects when they are accessed. This allows the programmer to control when objects get transformed.

As each object is updated, the standard CLOS generic function **`update-instance-for-redefined-class`** is called to do the work. By defining methods on this generic function, a programmer can personalize the updating process, for example, to retrieve the values of slots that are being deleted.

Handling object update automatically

AllegroStore permits the structure of its classes to be redefined at runtime. As a result of redefinition, existing objects of the affected classes will have their structure updated. The objects are updated according to the standard CLOS protocol. This protocol is a *lazy* one: objects are updated when they are next referenced. This is necessary because in CLOS there is no standard mechanism for locating all the objects of a given class.

AllegroStore can of course locate all the objects of a persistent class and so it can update all persistent objects in the database as soon as the class is redefined. Lazy updating is the default, but a program can easily force *eager* updating by using a query to reference all objects and accessing a slot at each object.

Lazy updating distributes the cost of the updating over time and thus is less disruptive in a real-time environment. With lazy updating, one can redefine numerous classes before causing any objects to be updated. This means that slots can be moved from one class to another without their values being removed from existing objects.

Object updating begins with the class or classes being redefined. Redefining a class effectively marks all of the current objects of that class and all of its subclasses as out-of-date. Here is how an out-of-date object is referenced for the first time after redefinition:

1. Space is allocated for the storage of its `instance` and `persistent-instance` slots based on the up-to-date class definition.
2. The instance variables are initialized (see the table below for details).
3. The function **`update-instance-for-redefined-class`** is called. This call is made primarily to allow a program to write a `:before` or `:around` method in order to do application specific actions.

For the purpose of object updating, slots are divided into two types: class-allocated (`:allocation :class` and `:allocation :persistent-class`) and instance-allocated (`:allocation :instance` and `:allocation :persistent`).

When an object is updated, an old slot may be dropped, it may change classification (from instance-allocated to class-allocated or vice versa), or it may be unchanged in the new instance. The following table shows what happens to slot values according to how the slot is changed:

:

Existing slot classification	New slot classification: class	New slot classification: instance	New slot classification: missing
class	xcopy	copy	discard
instance	init	copy	discard
missing	init	init	---

Legend:

- `copy` means that the value is copied into the new instance,
- `init` means that the old value is discarded and the new slot is set to the value of the `:initform`, if any.
- `discard` means that the value is thrown away.
- `xcopy` means that the value is copied when the new class is created, before the first instance is updated. So, for the purposes of this operation, the value is discarded/ignored.

Note: Copying a value from a non-set type to a set type will work. Everything other than lists are stored as the single object in the set while lists are made the set. Thus, the symbol `foo` will be stored as the lone value in the set. and the value `(a b c)` will be stored as three values in the set. Copying from sets to non-sets also works: all the items in the set are bundled into a list and that list becomes the value of the scalar slot. But note: it is an error if the type of a scalar slot does not include lists (in the set to non-set transformation).

6.17 Multiprocessing

Platforms with :os-threads on *features*

On platforms that include `:os-threads` in `*features*`, all AllegroStore functionality is available when using Allegro Common Lisp multiprocessing functionality. No special syntax is required and AllegroStore processing in Lisp lightweight processes other than the initial Lisp Listener runs just as fast as processing in an application that doesn't use Lisp multiprocessing.

Note that only one Lisp lightweight process can access AllegroStore functionality at a time - an internal lock is used to synchronize access. This implies that a short transaction programming model is best suited for AllegroStore multiprocessing on :os-threads platforms.

On :os-threads platforms, by default the Lisp heap is not released when an AllegroStore ObjectStore call is made. This means that if the ObjectStore call is waiting for a read or

write lock, all other Lisp lightweight processes will be waiting also. If the other lightweight processes are waiting to make an ObjectStore call, then they would wait even if the Lisp heap were available, since only one lightweight process can make an ObjectStore call at a time. However, if another Lisp lightweight process were doing other tasks, such as waiting to process user interface events, then it is desirable to release the Lisp heap when making an ObjectStore call.

You may control this behavior using the `*allegrostore-release-heap*` variable. When its value is nil (the default), the Lisp heap is not released during ObjectStore calls; when its value is non-nil, the Lisp heap is released during ObjectStore calls, allowing other Lisp lightweight processes to run. The `*allegrostore-release-heap*` variable is examined before each ObjectStore call, allowing you to selectively control the heap release behavior.

When choosing to release the heap, keep the following in mind:

- There are time and space costs associated with releasing the heap, even when other Lisp lightweight processes are just waiting for user interface events. Also, there are specific higher costs associated with persistent instances with slots containing long strings or raw arrays.
- When `*allegrostore-release-heap*` is non-nil, accessing a persistent raw array or storing a raw array in a persistent instance slot will release the Lisp heap only when the array is a one dimensional array. When storing a Lisp raw array in a persistent slot for the first time, the Lisp heap will only be released if :allocation `:lispstatic-reclaimable` was included in the **`make-array`** arguments.

Platforms without :os-threads on *features*

On platforms that do not include `:os-threads` in `*features*`, in earlier releases, AllegroStore supported channels but channels are not supported in release 2.0. The same functionality, with better overall performance, should be possible by running separate AllegroStore sessions that communicate via the AllegroStore notification facility. See section 6.24 for information on notifications.

6.18 Interactive transactions during application development

While developing an AllegroStore application, you may desire at some point to interactively examine a database. Here is an example that shows a simple way to do that:

```
(with-database (db "mydb.db")
   (with-transaction ()
      (break "Don't forget to return from break when finished")))
```

Running this code will result in a break during an active transaction. When the break occurs, you can then interactively evaluate forms that navigate the database or inspect persistent instances. If you return normally from the break, any database changes will be committed. If you choose an abort restart or do a reset, any changes made during the interactive period are rolled back.

Remember that you will have database read or write locks during the interactive transaction; any other application or user will be locked out until you commit or abort the transaction by choosing a restart for the break.

See the 6.22 *Long Transactions* section for information about another way to initiate interactive transactions.

6.19 Persistent Object Check Out and Check In

The AllegroStore Manual's Tutorial uses a public library database example to illustrate AllegroStore persistent class design and programming. In the example, all persistent instance slot reading and writing occurs within AllegroStore transactions. The library database is updated as each transaction completes and database changes are committed.

Such behavior is optimal for a public library database - when a patron leaves the library, the database should immediately reflect the books that the patron borrowed. As new books are added to the collection, the database should immediately reflect the addition. Such behavior is especially important in libraries with many clerks simultaneously accessing and updating the same collection database.

There are other kinds of applications, however, that may not require or may wish to avoid immediate database updates. Consider, for example, a Knowledge-Based Engineering application that uses AllegroStore to manage assembly and part objects. A design team member may wish to work with an assembly, trying different parts or modifying part parameters and reconstructing the assembly. After many iterations, a new design emerges, and the design team member then wishes to update the assembly that is stored in the database.

The designer will want to go through the iterations "off-line", so that the current assembly and part objects are available to other designers. Also, since reconstructing an assembly after changing a parameter is a computationally intensive activity with persistent instance and slot access occurring thousands of times, performance is an issue, since AllegroStore persistent instance slot access is slower than transient instance slot access.

For such an application, "checking out" persistent objects in a manner that allows manipulation as transient objects outside of AllegroStore transactions is the best way to provide off-line object manipulation and improve computational intensive application performance. When design iterations are complete, the designer's off-line changes are "checked in" to the persistent object database.

In the following sections, the class design and programming steps required to support persistent object check out and check in are illustrated.

The Example Database

Here are some non-persistent CLOS classes that describe a simple assembly/part system.

```
(defclass design-object ()
    ((id :accessor id :initarg :id)
     (name :accessor name :initarg :name)))

(defclass assembly (design-object)
      ;; elements is a list containing part instances or assembly instances
    ( (elements :accessor elements :initform nil)))

(defclass part (design-object)
```

```
    ())

(defclass pblock (part) ;; avoid symbol contention
    ((height :accessor height :initform 0.d0)
     (width :accessor width :initform 0.d0)
     (depth :accessor depth :initform 0.d0)))

(defclass cylinder (part)
    ((height :accessor height :initform 0.d0)
     (radius :accessor radius :initform 0.d0)))
```

A real life assembly/part class design would include slots containing information required to position sub-assemblies in an assembly, along with rules that govern how relative positioning changes when part parameters or positioning parameters change. However, for the purposes of demonstrating persistent object check out and check in, such added complexity is not needed.

Designing Persistent Classes for Check Out And Check In

The usual way to turn a transient class design into a persistent class design involves specifying the persistent-standard-class metaclass, specifying the `:persistent` argument for the `:allocation` keyword, and optionally using other persistent slot keywords, such as `:unique` and `:inverse`. For example, the code below shows how the design-object class might be turned into a persistent class for a design that did not require persistent object check out and check in.

You need to run AllegroStore to use the code contained in the rest of his document. On Windows, choose one of the ACL startup menu choices that includes AllegroStore. On Unix, evaluate `(require :allegrostore)`. Also, all the code below assumes you are in the AllegroStore package (eval a `(in-package :astore)` form) or you are using the AllegroStore package (eval a (use-package :astore) form). On Windows, when using an image containing Common Graphics or the IDE, to prevent symbol conflicts, evaluate

`(defpackage :allegrostore (:shadowing-import-from :aclwin collect))`

before evaluating `(use-package :astore)`.

```
(defclass design-object ()
    ((id :accessor id :allocation :persistent
         :unique t :inverse find-object :initarg :id)
     (name :accessor name :allocation :persistent :initarg :name))
   (:metaclass persistent-standard-class))
```

When turning a transient class design into a persistent class design that supports persistent object check out and check in, add parallel persistent slots for each transient slot, and also add a transient slot that will let you know when a persistent object has been checked out. Here is the above assembly/part design, transformed to support persistent object check out and check in:

```
(defclass design-object ()
    ((id :accessor id)
     (id-po :accessor id-po :allocation :persistent :unique t
            :inverse find-object :initarg :id)
     (name :accessor name)
     (name-po :accessor name-po :allocation :persistent :initarg :name)
     (checked-out-p :accessor checked-out-p :initform nil))
  (:metaclass persistent-standard-class))
```

```
(defclass assembly (design-object)
      ;; elements is a list containing part instances or assembly instances
    ((elements :accessor elements)
     (elements-po :accessor elements-po :allocation :persistent
                  :initform nil))
  (:metaclass persistent-standard-class))

(defclass part (design-object)
    ()
  (:metaclass persistent-standard-class))

(defclass pblock (part) ;; avoid package problem associate with 'block
    ((height :accessor height)
     (height-po :accessor height-po :initform 0.d0 :allocation :persistent)
      (width :accessor width)
      (width-po :accessor width-po :initform 0.d0 :allocation :persistent)
      (depth :accessor depth)
     (depth-po :accessor depth-po :initform 0.d0 :allocation :persistent))
  (:metaclass persistent-standard-class))

(defclass cylinder (part)
    ((height :accessor height)
     (height-po :accessor height-po :initform 0.d0 :allocation :persistent)
      (radius :accessor radius)
    (radius-po :accessor radius-po :initform 0.d0 :allocation :persistent))
  (:metaclass persistent-standard-class))
```

Note that the above design does not include a persistent slot that contains check out status information. In a real multi-user environment, a design might include persistent slots that identify who has checked out an object and differentiate between read-only check outs and read/write check outs. The methods that control object check out and check in could use such information to allow or disallow a check out.

Check Out and Check In Methods

```
;; all methods assume that they are called within a transaction
(defmethod check-out :around ((object design-object) &optional lazyp)
   (declare (ignore lazyp))
   ;; using an around method insures that we set checked-out-p
   ;; first, to prevent infinite recursion
   ;; during full depth check out and also to avoid work when the
   ;; object is already checked out
   ;; it also allows us to reset checked-out-p to nil if there's a failure
   (if* (checked-out-p object)
     then object
     else
     (handler-case
      (progn
       (setf (checked-out-p object) t)
       (call-next-method)
       (preserve-pointer object))
      (error (condition)
        (progn
         (setf (checked-out-p object) nil)
         (error condition))))))

(defmethod check-out ((object design-object) &optional lazyp)
```

```
   (declare (ignore lazyp))
    ;; copy persistent values into parallel transient slots
    ;; note that we assume the name and id slots will never contain a
    ;; persistent instance or list
    (setf (id object) (id-po object))
    (setf (name object) (name-po object)))

;; the other methods are :after methods, so they are called
;; along with the primary method
(defmethod check-out :after ((object assembly) &optional lazyp)
        ;; assumes that elements is empty list or list containing parts
        ;; and/or assembly instances
    (setf (elements object)
        (let (transient-list)
           (dolist (element (elements-po object) (reverse transient-list))
                (push (if* lazyp then (preserve-pointer element) else
                           (check-out element))
                    transient-list)))))

(defmethod check-out :after ((object pblock) &optional lazyp)
    (declare (ignore lazyp))
    ;; assumes that height, width, depth contain numbers
    (setf (height object) (height-po object))
    (setf (width object) (width-po object))
    (setf (depth object) (depth-po object)))

(defmethod check-out :after ((object cylinder) &optional lazyp)
    (declare (ignore lazyp))
    ;; assumes that height and radius contain numbers
    (setf (height object) (height-po object))
    (setf (radius object) (radius-po object)))

(defmethod check-in :around ((object design-object) &optional lazyp)
    (declare (ignore lazyp))
    (when (checked-out-p object) (call-next-method)))

(defmethod check-in ((object design-object) &optional lazyp)
    (declare (ignore lazyp))
    (setf (id-po object) (id object))
    (setf (name-po object) (name object)))

(defmethod check-in :after ((object assembly) &optional lazyp)
    (setf (elements-po object) (elements object))
    (when (not lazyp)
        (dolist (element (elements object))
            (check-in element t))))

(defmethod check-in :after ((object pblock) &optional lazyp)
    (declare (ignore lazyp))
    (setf (height-po object) (height object))
    (setf (width-po object) (width object))
    (setf (depth-po object) (depth object)))

(defmethod check-in :after ((object cylinder) &optional lazyp)
    (declare (ignore lazyp))
    (setf (height-po object) (height object))
    (setf (radius-po object) (radius object)))
```

In the above methods, the *lazyp* argument specifies whether check out or check in should also process persistent objects contained in an instance's slots. In a real life assembly/part system doing a full depth check out of a complex assembly could result in a check out of thousands of persistent objects. If the intention is to check out the assembly, and then navigate through a particular sub-assembly path, it may be desirable to check out instances "as you go".

Notice also the use of **`preserve-pointer`**. The **`preserve-pointer`** function is crucial for enabling persistent instance reconciliation outside transactions and in subsequent transactions. Preserved pointers require that the user have write permission on the database and that the database remain open between transactions. Thus, you should use **`open-database`** and **`with-current-database`**, rather than **`with-database`** when using functionality that involves persistent object check out and check in.

Creating New Instances

As with usual AllegroStore programming, create new persistent instances within a transaction. In an object check in and check out environment, where the majority of object manipulation code will access transient slots, it will usually be easier to create a minimal new instance, immediately check it out, and then begin working with it as a checked out object.

Some Example Sessions

The following code creates some parts and an assembly. Instance attributes are manipulated outside of a transaction.

```
(defun make-an-assembly ()
   (let ((db (open-database "f:/tmp/design.db"))
         part1 part2 assembly)
      ;; make some objects and check them out
      (with-current-database db
        (with-transaction ()
          (setf part1 (check-out (make-instance 'pblock :id 1
                                                :name "block")))
          (setf part2 (check-out (make-instance 'cylinder :id 2
                                                :name "cylinder")))
          (setf assembly
                (check-out (make-instance 'assembly :id 3
                                          :name "assembly")))))
      ;; now work with the checked out objects outside of a transaction
      (setf (height part1) 1.d0)
      (setf (width part1) 2.d0)
      (setf (depth part1) 3.d0)
      (setf (height part2) 4.d0)
      (setf (radius part2) 5.d0)
      (push part1 (elements assembly))
      (push part2 (elements assembly))
      ;; now check the assembly back in, using full depth check in,
      ;; which will check in the parts, also
      (with-current-database db
        (with-transaction ()
          (check-in assembly)))
      ;; we're done, so close the database
      (close-database db)))
```

The following code simulates a design session using the parts and assembly created above. It may be run in a separate Lisp session.

```
(defun simulate-design-session ()
  (let ((db (open-database "f:/tmp/design.db"))
        block assembly)
    ;; check out the block
    (with-current-database db
      (with-transaction ()
        (setf block (check-out (find-object 1)))))
    ;; verify that part contains attributes we expect
    ;; note that there is no active transaction
    (format t "part: ~s height: ~s width: ~s depth: ~s~%"
      (name block) (height block) (width block) (depth block))
    ;; modify the height
    (setf (height block) 10.d0)
    ;; check out the assembly
    (with-current-database db
      (with-transaction ()
        (setf assembly (check-out (find-object 3)))))
    ;; print out element heights - notice that our modification
    ;; has not been lost
    (dolist (element (elements assembly))
      (format t "part: ~s height: ~s~%" (name element) (height element)))
    ;; check in the block - that's the only thing that changed
    (with-current-database db
      (with-transaction ()
        (check-in block)))
    ;; we're done - close database
    (close-database db)))
```

Here is the output that running this function produces:

```
> (simulate-design-session)
part: "block" height: 1.0d0 width: 2.0d0 depth: 3.0d0
part: "cylinder" height: 4.0d0
part: "block" height: 10.d0
18
>
```

Conclusions

The above example shows another persistent check out/check in design benefit - code that must be added to support persistent objects is minimal and can be localized to areas away from where the important processing occurs.

If your persistent object application does not require "real time" persistent object update, you should consider using object check out and check in; it may make application development easier, and the performance advantages (you don't make foreign function calls when you access transient slots) may be dramatic.

6.20 Reducing Page Lock Contention

If you are developing a multi-client application, where more than one process will be accessing the same database at the same time, you should be concerned about page lock contention. Page lock contention occurs when a process attempts to read or write to a database page that another process currently has locked. The page in question will not become available until the process that has currently locked it completes the transaction it is working on.

Page lock contention occurs when one process has a write lock on a page and another process seeks a read lock or write lock for that page, or when one or more processes have a read lock on a page and another process seeks a write lock on that page.

ObjectStore automatically manages what kind of lock a process will request. A write lock will be requested only when an attempt to write to persistent memory occurs in the transaction. If all transaction operations are persistent memory reads, then a read lock will be requested. If a transaction starts out reading and then writes, the initial read lock will be upgraded to a write lock. The upgrade attempt will result in page lock contention if another process also currently has a read lock for the page in question.

For most AllegroStore operations, it is obvious whether an operation will read persistent memory or write persistent memory. For example, if you use a slot accessor to fetch the current contents of a persistent slot in a persistent object, you are performing a read operation. If you use 'setf to change the slot value, you are performing a write operation.

Here are the non-obvious write operations:

- preserve-pointer

  When you create a preserved pointer, a database write occurs. When you close the database in which a preserved pointer was created, a database write occurs as the preserved pointer table is deleted.

- fetching a stale instance

  If an instance's persistent class definition has changed since the last time the instance was accessed, AllegroStore will update the instance with a write operation. See the next section about read-only processing to see how to manage when stale instances get updated, and how to prevent unexpected persistent class definition changes.

If all transaction operations are read operations when your application runs in multi-user mode, then page lock contention will never occur. If at least one transaction performs write operations, then the potential exists for page lock contentions. Here are some strategies for reducing the chance that page lock contention will occur:

- keep all instances fresh

  See the next section about read-only processing for more information.

- keep transactions as short as possible

  Short transactions reduce the probability that page lock contention will occur, and if contention does occur, reduces the time that a process waits for a lock to be free. If your application is not readily suited to short transactions, consider using an object check-out mechanism that allows processing outside of transactions. See section **6.18 Persistent Object Check Out and Check In** for more information. Object check-out uses preserved pointers, which require write locks, but the transactions required to create preserved pointers during object checkout are very short.

- cluster related objects

  If two clients are performing write operations on unrelated database areas, then lock contention will not occur. Currently, you can only influence object clustering by managing when objects are created or by using a multi-database design. For example, suppose you have a sales database, and the salespeople who will use your application manage different geographical areas. If, when creat-

ing a single sales database, you first create all the California objects, and then create all the New York objects, the two salespeople responsible for those areas are unlikely to contend for the same page when updating an object. Alternatively, if you design your application so that the New York objects and California objects are in distinct databases, then lock contention will be even less likely.

- consider using MVCC database opens for read-only processing

  If your read-only multi-user clients can work with a consistent database snapshot, where database updates by other clients are not visible until the current transaction is completed and a new transaction is started, then you should consider using Multi Version Concurrency Control (MVCC). See section 6.21 **Multi Version Concurrency Control (MVCC) Processing** for more information.

6.21 Read-Only Processing

You can use read-only processing to prevent unwanted attempts to secure a write lock. When a database is opened in read-only mode, any attempt to write to persistent memory associated with that database results in an AllegroStore error. When a transaction is started in read-only mode, any attempt to write to persistent memory during that transaction results in an AllegroStore error.

To open a database in read-only mode, specify t as the argument to the `:read-only` **`with-database`** or **`open-database`** keyword argument. To start a read-only transaction, specify t as the argument to the `:read-only` **`with-transaction`** keyword.

Here are some examples:

```
 (with-database (db "foo.db" :read-only t)
   ...  ;; transaction in which write occurs will result in error
)

(with-database (db "foo.db")
  (with-transaction ()
    ...  ;; write can occur in here
    )
  (with-transaction (:read-only t)
     .. ;; write in here will result in an error
    ))
```

Since creating a preserved pointer involves writing to persistent memory, attempting to create a preserved pointer during read-only processing will result in an AllegroStore error.

Attempting to fetch a stale instance during read-only processing will result in an AllegroStore error. Stale instances are instances written to a database before their associated class definition has been updated in the database. There are two ways a class definition can be updated in the database:

- Intentional class redefinition

  You change a class definition, open a database containing the same class, and use the new definition to resolve the error condition raised by AllegroStore.

- Transparent class definition update

If a class definition exists in transient memory that matches a class in a newly opened database, AllegroStore will transparently rewrite the class definition to the database, even if the two definitions are identical. AllegroStore does this because it cannot be sure that a change in a persistent superclass did not occur, and it must rewrite the definition when a superclass changes. This situation will occur if:

- The class in question is in the image you start when you start your application.
- The class in question is in a compiled fasl file or Lisp source file you load into your application.
- You open a database containing the class in question, then open another database containing the same class.
- You open a database containing the class in question, later close the database, and then later open the database again.

To prevent transparent class definition update, call the CLOS **`finalize-inheritance`** method on all persistent class definitions before opening a database. The easiest way to do this is to add **`finalize-inheritance`** calls to the bottom of the last source file that is loaded when your application is built or loaded. Wrap the calls within an `(eval-when (load))` form.

If you make existing instances stale by redefining a class, you can update all the instances using the **`for-each*`** and **`allegrostore::validate-instance functions`**, as in:

```
(for-each* #'(lambda (obj) (allegrostore::validate-instance obj))
           'foo :subclasses t)
```

For very, very large databases, if there is not enough addresses space to update all instances in a single transaction, use the **`for-each*`** function :start-block and :block-count keywords to break the operation up into smaller transactions, as in:

```
(dotimes (i 10)
   (with-transaction ()
      (for-each* #'(lambda (obj) (allegrostore::validate-instance obj))
             'foo
             :subclasses t :start-block (* i 10000000)
             :block-count 10000000)))
               ;; process 10,000,000 objects per transaction
```

6.22 Multi Version Concurrency Control (MVCC) Processing

You can use MVCC processing to eliminate lock contention in a multi-user environment if:

- The client you are considering using MVCC processing with does not do any database writes during all transactions.
- The client can effectively operate with a consistent database snapshot - it is not required to immediately see the results of write operations by another client.

To open a database using MVCC mode, specify t as the argument to the :mvcc with-database or open-database keyword argument.

Here is an example:

```
(with-database (db "foo.db" :mvcc t)
   ... ;; read only transactions
)
```

Here are the benefits related to MVCC processing:

- A client using MVCC database opens will never wait for a read lock, regardless of what any other client is doing.
- A client not using MVCC database opens seeking to do a database write will not wait for a write lock if the only other clients accessing the database page in question are MVCC clients.

Here are issues related to MVCC processing:

- While a MVCC snapshot will always be consistent within itself, multiple database application developers must be concerned about database sets that are not consistent with each other. If a non-MVCC client is committing transactions on multiple databases at the same time that another client is doing MVCC opens, there is a timing risk that can lead to some of the MVCC opened databases reflecting the commit changes, and others not reflecting those changes. It is the programmer's responsibility to prevent this from happening or to detect this condition and deal with it appropriately.
- If non MVCC clients do massive amounts of database writes while there are MVCC open databases, the ObjectStore transaction log may grow very large, and you may run into resource problems, such as not enough disk space.

6.23 Long Transactions

The recommended AllegroStore transaction model is a short transaction model - keeping transactions short reduces the probability that lock contention or deadlock will occur.

A long transaction model - where large amounts of processing occurs within a single transaction, or where the application may pause within an open transaction while awaiting user input, is a safe choice only when you can prevent lock contention. For example, opening a database in MVCC mode guarantees lock contention free read-only processing for the application doing the open, along with guaranteeing that any other application using the same database won't encounter lock contention. Another example is an application that generates a distinct file name (using 'sys:make-temp-file-name, for example) when creating and using a database during a run session.

Even if you do not plan on developing a multi-user application, you should consider the risks related to lock contention. For example, a user may start a standalone AllegroStore application and forget to exit it. If the application is in the middle of a long transaction on a database file that is used every time the application runs, then no one, including the user who left the first session running, will be able to do any work until the original session is terminated.

Most applications that seem best suited for a long transaction model usually also work well with a short transaction model design involving multiple databases, preserved pointers, and transient slots. For example, consider a CAD application where a user's project is contained in a single database file. In a long transaction model, an entire CAD application session might occur within a single transaction, with the transaction committing when the

user saves his work or aborting if the user exits without first saving. In a short transaction model, a temporary database would be used for the session. When the database user opens his project, the required objects are copied from the permanent database into the temporary database, and all object modification is done within the temporary database using short transactions. If the user wishes to save the session work, the permanent database is reopened and the relevant objects are updated.

If you decide a long transaction is best and you are sure that lock contention is not an issue, large amounts of processing and pauses for user interaction may occur within a with-transaction form. If your application is event driven (such as a Common Graphics or CLIM application), you may prefer to use a functional analog to the 'with-transaction macro - the begin-transaction, commit-transaction, and abort-transaction functions.

Here is an example of their use:

```
;; db is the return value from an earlier
;; successful open-database call
(set-current-database db)
(begin-transaction)
;; myfun returns t if the user wishes to save
(if* (myfun)
   then (commit-transaction)
   else (abort-transaction))
```

Note that commit-transaction or abort-transaction will raise an error condition if they are called within a top-level with-transaction form. To force a commit in a top-level with-transaction form, cause the form to complete normally. To force a rollback, transfer control outside the form; error or throw is a way to do that.

When using the transaction functions instead of the with-transaction macro, special care must be taken when handling allegro-exception conditions (see the 7.2.XX Conditions section). To prevent runtime problems and possible database corruption, you must call abort-transaction in your condition handler.

6.24 Notifications

Multi-client AllegroStore applications may use notifications to send and receive messages between client processes.

Why use notifications?

- Notifications are easy to use - there is no need to write your own socket code and design a messaging protocol
- Notifications may be tied to AllegroStore two-phase commits - you may specify that notifications be sent only after the transaction in which they were created successfully commits.
- Notifications allow clients to refer to the same persistent objects - you don't have to include a unique id slot in a persistent class just so clients can communicate about a desired instance.

Setting up notifications

To receive notification, a client must *subscribe* for them. A client subscribes for notifications concerning a specific instance with the **`notification-subscribe`** method:

```
(notification-subscribe foo) ;; foo is a persistent instance
```

This method must be invoked while a transaction is active. Once the subscription is made, when any client sends a notification regarding the specified persistent instance, a notification will be placed in this client's notification queue. The subscription remains in effect as long as the database remains open.

If you no longer wish to receive notifications regarding an instance you previously called notification-subscribe on, use the **`notification-unsubscribe`** method:

```
(notification-unsubscribe foo) ;; foo is a persistent instance
```

This method must be invoked while a transaction is active.

Sending a notification

A client notifies all subscribing clients about a specific instance with the notify method:

```
(notify foo 1 "this is a string" :on-commit t)
                ;; foo is a persistent instance
```

The second argument to **`notify`** is an integer value that the application can use to distinguish kinds of notifications - you are free to use any integer value. The third argument specifies an arbitrary string that may be used to pass additional information. The :on-commit keyword argument specifies whether the notification will be sent only if the wrapping transaction commits successfully. If the :on-commit argument value is nil, then the notification will be sent immediately.

Waiting for notifications

Depending on how your multi-client application operates and the expected notification traffic, there are a variety of strategies for notification waiting or polling. The correct strategy also depends on whether the :os-threads feature is present in the lisp image.

If a client's only current task is waiting for a notification, you can call the **`notification-receive`** function:

```
(notification-receive -1)
```

The -1 argument specifies to wait forever until a notification arrives. The waiting is done in a manner that does not impinge on computing resources. An integer argument value greater than -1 specifies the number of microseconds to wait for a notification. If a timeout occurs, nil is returned; otherwise, a notification object is returned.

Note that notification-receive need not be called within an active transaction.

For clients working on other tasks, periodically checking for pending notifications, and not expecting heavy notification traffic, you have multiple choices.

You can use notification-receive with a reasonable timeout value or you can poll to see if there are any pending notifications.

Alternatively, you can use the notification-queue-status function:

```
(notification-queue-status)
```

This function returns a list of three integers; the first value is the number of pending notifications, the second value is the number of dropped notifications resulting from queue overflow, and the third value is the queue size. The default queue size is 50; you can change the default by setting the OS_NOTIFICATION_QUEUE_SIZE environmental variable before starting your application.

On Unix systems, you can also check for pending input on the file descriptor associated with notification:

```
(stream-listen (astore::os_notification_get_fd))
```

The stream-listen method is an Allegro Common Lisp streams method; it returns t if there is some pending input, nil otherwise.

The **`notification-queue-status`** and **`astore::os_notification_get_fd`** functions may be called outside of active transactions.

If you anticipate heavy notification traffic, a more robust design involves using Allegro Common Lisp multiprocessing functionality to reduce the chance that notification queue overflows occur.

Here is an example that is appropriate for architectures where :os-threads is on the feature list:

```
(mp:process-run-function
    "receive-notification"
    #'(lambda ()
         (let ((*allegrostore-release-heap* t)
                (notification (notification-receive -1)))
               ;; examine notification or place it in
               ;; global list that other ACL "threads" may examine
               )))
```

The above example process function will terminate after receiving one notification; it is a simple exercise to add a looping mechanism that will run until you wish to terminate the notification processing. Note that the *allegrostore-release-heap* symbol is bound to t; this releases the heap so other threads running Lisp code will continue to run.

Here is a similar example appropriate for Unix architectures where :os-threads is not on the feature list:

```
(mp:process-run-function
    "receive-notification"
    #'(lambda ()
          (let ((fd (astore::os_notification_get_fd)))
               (mp:wait-for-input-available fd))
           (let ((notification (notification-receive -1)))
                 ;; examine notification or place it in
                 ;; global list that other ACL "threads" may examine
                )))
```

Note that Windows is an :os-threads architecture.

Examining notification objects

Once a notification object is received, notification details may be examined using:

`database-of`

The **`database-of`** method returns the database object associated with the notification.

`notification-kind`

The **`notification-kind`** method returns the kind integer specified by the notification sender.

`notification-string`

The **notification-string** method returns the string specified by the notification sender.

The above methods may be called outside an active AllegroStore transaction on any Lisp lightweight process.

notification-object

The **notification-object** method returns the persistent instance specified by the notification sender. This method must be called while a transaction is active. It may fail if any client called **unpreserve-pointer** on the instance in question since the notification-subscribe call on the particular instance was made. Note that closing a database will result in an implicit unpreserve-pointer call on the instance in question.

Chapter 7 Reference guide

This chapter is divided into two parts. In the first part, section 7.1 **General information**, are essays about various topics. Each is labeled **About *topic***. This part contains the following essays:

About saving and restoring databases

About multitasking

About the configuration database

About moving and copying database files

About deleting database files

About persistent standard class

About persistent standard object

About read-locks and write-locks

About schema

About transaction

About commit

About roll back

About shell environment variables

About deadlock resolution

About deleting database files

About shrinking the transaction log

The second part, section 7.2 **Definitions**, is the reference section proper. It contains formal definitions of the variables, functions, macros, and so on in AllegroStore. The definitions are grouped by general topic (rather than, say, listed alphabetically). Therefore, you look at the section on **Transactions** for information on functionality associated with transactions. These sections are numbered, the section number being 7.2.*n*, where *n* is the number in the list below.

1. **Variables**
2. **Database manipulation**
3. **Databases: saving and restoring**
4. **Schema manipulation**
5. **Transactions**
6. **Object manipulation**
7. **Query language**

8. **References**
9. **Object identifiers**
10. **Persistent hash tables**
11. **Blobs**
12. **Lock timeouts**
13. **Conditions**

The topics are ordered roughly by complexity. If you read through the chapter from front to back, you will gain a progressively deeper understanding of the software.

If you are looking for a more general discussion or examples, see the **Programmer's guide**. If you haven't already done so, please read chapter 5 **Tutorial** before beginning to use AllegroStore.

7.1 General information

About saving and restoring databases

The programs **`asdump`** and **`asrestore`**, documented in detail in section 7.2.2 below, respectively write the contents of a database to a machine-independent ASCII file and recreate a 1.3 database from such a file. AllegroStore 1.3 can read an AllegroStore 1.2 **`asdump`** file.

About multiprocessing (:os-threads version)

All AllegroStore functionality is available when using Allegro CL multiprocessing functionality (see *<Allegro directory>/doc/cl/multiprocessing.htm*). An internal Lisp lock forces other lightweight processes wishing to start an AllegroStore transaction to wait until a lightweight process currently working on a transaction commits or aborts the transaction. While a lightweight process is in ObjectStore foreign code, other Lisp lightweight processes not waiting for the transaction can run Lisp code.

About multitasking (non :os-threads version)

On platforms that do not include `:os-threads` in `*features*`, in earlier releases, AllegroStore supported channels but channels are not supported in release 2.0. The same functionality, with better overall performance, should be possible by running separate AllegroStore sessions that communicate via the AllegroStore notification facility. See section 6.24 for information on notifications.

About the configuration database

The structure of the primitive objects that AllegroStore uses to build the CLOS database are described in the *configuration database*. Allegrostore locates and opens the configuration database file before it opens its first database. The system will perform a sequential search, and will stop as soon as it locates the file.

The search for the configuration database proceeds as follows:

1. The file that contains the C-coded part of AllegroStore has the name of the configuration file stored inside it (using the **ossetasp** program). AllegroStore first looks for the configuration file in the location name stored in this file.
2. Next, AllegroStore looks in each directory mentioned in the shell environment variable `AS_CONFIG_PATH`. The format of the value of this environment variable is similar to the shell `PATH` variable:
   - In Unix, a sequence of directory names separated by colons
   - In DOS, a sequence of directory names separated by semicolons
3. Finally, AllegroStore looks in each directory mentioned in `*as-config-path*`. The initial value of `*as-config-path*` is a list of one directory: the current working directory.
4. If AllegroStore cannot locate the file, it will issue an error message:

```
cannot find configuration database
```

About moving and copying database files

If you plan to move or copy AllegroStore databases, be aware that the database server process may be caching some information that belongs in the file. Simply copying a file with **tar** or **cp** (on UNIX) or **copy** (on DOS) may result in a corrupted database file.

The ObjectStore *bin* subdirectory supplied with AllegroStore contains replacement file management functions that are database-aware. You should use **oscp** for copying files and **osmv** for renaming them.

See the sections on **oscp** and **osmv** in **Database user utilities** chapter.

About deleting database files

If you want to delete a database file, use **osrm** (rather than the Unix utility **rm**). See the **osrm** documentation in chapter 10 **User utilities**.

About persistent-standard-class

A Lisp program that uses AllegroStore creates and accesses two kinds of objects: *transient* objects and *persistent* objects.

- A transient object exists entirely in the Lisp heap and ceases to exist when there are no more pointers to the object (or when the Lisp process quits).
- A persistent object has two parts: a transient part and a persistent part.

  The transient part of a persistent object exists in the Lisp heap and ceases to exist when there are no more pointers to it (or when the Lisp process quits).

  The persistent part of a persistent object exists in the database and ceases to exist when the program explicitly deletes the object or the object's class. (Its existence does not depend on whether or not the Lisp process is running.)

Both transient and persistent objects are created with **make-instance**. Whether an object is transient or persistent is determined by its class definition (done with **defclass**).

If the *metaclass* of an object is `persistent-standard-class` (or a subclass of that class), the object will be persistent. If the metaclass is `standard-class` (or a subclass of it), the object is transient. The following form defines a *persistent class* of objects:

```
(defclass persistent-foo ()
  ()
  (:metaclass persistent-standard-class))
```

A persistent class can be a subclass of non-persistent classes, but it rarely makes sense for a non-persistent class to be a subclass of a persistent class since a non-persistent class cannot process slots with persistent allocation types.

Although you can define persistent classes at any time, you cannot create instances of persistent classes unless a database is open.

About persistent slots

Slots in persistent classes can themselves be persistent or transient. (This feature is useful since you may not want all the slots of a class to be stored in a database, since doing so may waste space unnecessarily.) Whether a slot is persistent or transient depends on its `:allocation` option. `:allocation` is a standard CLOS option, but AllegroStore permits four rather than two possible values. AllegroStore also adds three new slot options, appropriate to database management. The following table shows the four options (`:allocation` and the three new ones) along with descriptions of their values. Note that certain options are only legal if other options have specified values.

Slot type	Options available
`:allocation`	As well as the standard CLOS options `:instance` and `:class`, this can be `:persistent` or `:persistent-class`.
`:unique`	Can be specified only if `:allocation` is `:persistent`. Can be `t` or `nil`. If `t`, then this is a promise that the value of the slot will be unique among instances of this class and subclasses. The system verifies that the slot value is unique, so storing values takes longer if `t`. Retrieving can be faster, however. `:set` slots (see below in table) cannot have `:unique t` specified. If you declare a slot to be `:unique t` then AllegroStore will enforce that declaration and will signal an error if you attempt to store the same value into different instances.
`:inverse`	Can be specified only if `:allocation` is `:persistent`. If specified, the value names a function that will perform the inverse mapping, i.e., given a value it will return all the instances that have the value in this slot.
`:set`	Can be specified only if `:allocation` is `:persistent` or `:persistent-class`. If `t`, then this is a set-valued slot. `:unique` cannot be `t` if `:set` is `t`.
`:vector`	Similar to `:set t`, except you are declaring that the slot, if bound, contains a vector. Furthermore, if there is an inverse defined for the slot, then the inverse is defined over each element of the vector and not the vector object itself.

For information about what types of pointers can exist between persistent objects, transient objects, and other Lisp objects, see the section 6.11 **Pointers**.

About persistent-standard-object

`persistent-standard-object` is the superclass of all persistent classes. If you define a persistent class and don't specify any superclasses, then `persistent-standard-object` will automatically be added to the superclass list. If you do specify superclasses, then you must be sure that `persistent-standard-object` is a superclass.

About read-locks and write-locks

When a persistent data object is read, the page on which it resides (in memory) is *read-locked* by the system. When a data object is written, the page is *write-locked* by the system.

Read-locked pages:

> When another process attempts to gain write-access to an item on a read-locked page, the access is delayed until the page is unlocked (the outermost transaction of the process which has control is committed or rolled back).
>
> Read-access to a read-locked page is not delayed.

Write-locked pages:

> When another process attempts to gain read- or write-access to an item on a write-locked page, the access is delayed until the page is unlocked (the outermost transaction of the process which has control is committed or rolled back).

See section 7.2.12 **Lock timeouts** for information on how to determine and reset the wait time for locks.

About schema

An AllegroStore schema is a set of class definitions.

The schema doesn't include function definitions per se: when a class is defined, you can name the function that will be used to access the slots of instances of that particular class (called an *accessor function*). Reader- and writer-functions are special cases of accessor functions. (Readers -- the value of the `:reader` option -- can read but cannot set. Writers -- the value of the `:writer` option -- can set but cannot read. A true accessor -- the value of the `:accessor` option -- can do both. Usage in the field is somewhat sloppy, with 'accessor' being used to refer to readers, writers, and true accessors, thus avoiding having to say 'reader, writer, or accessor' when 'accessor' gets the point across, as in the next paragraph.)

When you exit the current Lisp, start a new one, and open a database, the schema definition has to be re-read by the system. Class definitions are read from the database and the accessor functions are defined.

To learn what happens when a schema is changed so class definitions in the current Lisp don't match the ones stored in the database, see section 6.4 **Schema**, particularly the information under the heading **How schema differences are reconciled**.

See the section 7.2.4 **Schema manipulation** later in this chapter for definitions.

About transaction

A *transaction* is a sequence of database operations which form a logical unit of work.

A transaction completes either by being *committed* (all changes made permanent) or by being *rolled back* (all changes undone). Any changes made by one program to the database will not be visible to other database users unless and until the transaction commits.

Transactions are isolated from one other. Simultaneous transactions cannot make changes to one set of data. Once a transaction gains control over a page of data, the system sets up a flag (called a *lock*) which marks that page as 'in use.' Once the transaction has finished, the page is freed of all locks -- effectively marking it as 'available.' Locks preserve database consistency.

Databases contain interrelated information and can be said to be in a consistent state when certain relations exist between the data. To change a database from one consistent state to another usually requires a number of database operations. Placing those operations inside of a transaction means that the database will always appear to other processes to be in a consistent state.

In AllegroStore, transactions are started by the **`with-transaction`** macro. Transactions are *atomic*: either all of their database changes are made (the transaction commits) or none of them are made (the transaction rolls back).

AllegroStore does not support nested transactions. It is possible to dynamically nest **`with-transaction`** forms, but the inner **`with-transaction`**s are converted to **`progn`**s. This means that just because a **`with-transaction`** form completes, you cannot be sure that the changes made within the form have been committed to the database. See section 6.5 **Transactions**, particularly the information under the heading **Nested transactions** for more information. You are told there how to ensure that a transaction is a *top level* transaction (where successful completion means the changes have been committed to the database).

About commit

When does a transaction commit? If the body of the **`with-transaction`** macro completes execution without a transfer of control outside the **`with-transaction`** form, then the transaction commits, and all of the pending changes are made.

About roll back

When does a transaction roll back? If control transfers out of the **`with-transaction`** form, then the transaction rolls back, and all pending changes are undone. For example, errors cause control to transfer out and thus cause the transaction to roll back.

The system will sometimes select a transaction to roll back during a deadlock. See the information under the heading **The problem of deadlocks** in section 6.5 **Transactions**.

About shell environment variables

There are three shell environment variables that each AllegroStore user must set in their programming environment:

`OS_ROOTDIR` **[shell environment variable]**

■ The value of this variable should be the name of the top directory of the ObjectStore files (e.g., `/usr/objectstore`).

OS_AS_START [shell environment variable]

■ Specifies the starting address for the persistent virtual memory address space.

■ **WARNING**: Because of an **apparent bug in ObjectStore on Solaris 2.x**, using the **default value** of this variable can result in data corruption resulting in AllegroStore **crashing**. On Solaris 2.x only the variable must be set to a value at least 0xd0000000.

OS_AS_SIZE [shell environment variable]

■ Specifies the maximum size of the persistent virtual memory address space.

LD_LIBRARY_PATH [shell environment variable]

SHLIB_PATH [shell environment variable]

■ (Unix users only.) LD_LIBRARY_PATH (on Sparcs and some other UNIX machines) and SHLIB_PATH (on HP workstations) are environment variables that communicate to running programs information on where to find shared object libraries. There is no standard name for this variable in UNIX and, as indicated, LD_LIBRARY_PATH is used by the Solaris operating system on Sparcs (and by some other variants of UNIX) while SHLIB_PATH is used by the HP/UX operating system on HP machines. If you are not on a Sparc or an HP, please check your operating system documentation for the equivalent environment variable. Whatever it is, it should include $OS_ROOTDIR/lib, as well as /usr/local/lib and /usr/lib. The following C shell command shows how to update LD_LIBRARY_PATH so all three directories are prepended to its current setting. (In fact, */usr/lib* and */usr/local/lib* will typically be included if the variable is set at all.) Because space limitations forces us to use two lines, we have put a backslash at the end of the first line:

```
setenv LD_LIBRARY_PATH \
 $OS_ROOTDIR/lib:/usr/lib:/usr/local/lib:$LD_LIBRARY_PATH
```

The same command, with LD_LIBRARY_PATH replaced by SHLIB_PATH updates SHLIB_PATH.

AS_CONFIG_PATH [shell environment variable]

■ The value of this variable is the pathname of the AllegroStore configuration directory. This directory contains files needed by AllegroStore when it runs. On the distribution tape, this directory has the name *as-vN* where *N* is the version number. For example, the release on Sparcs running Solaris 2 when this manual was printed, the directory was named *as-v1.0.54*. This directory is copied (usually with the same name) to some permanent location. Again, on the Sparc release installation guide, we gave */usr/allegrostore/as-v1.0.54* as the permanent location, so that directory should be in the list that is the value of the variable.

■ More than one such directory can be specified (and they will be searched in the order specified) to allow for different machines and configurations, but specifying one directory only is typical, at least at first. Multiple directories should be separated by colons, as those in LD_LIBRARY_PATH are.

■ The value in the following example comes from the Sparc/Solaris 2 installation. Check your installation guide (chapter 1) for appropriate values for your environment.

```
setenv AS_CONFIG_PATH /usr/allegrostore/as-v1.0.54
```

PATH **[shell environment variable]**

■ The value of this variable include the location of the AllegroStore image, the AllegroStore configuration directory, and `$OS_ROOTDIR/bin`. Users who are also database administrators should include `$OS_ROOTDIR/admin`. In the example showing how to set the variable, we assume the AllegroStore image is in the configuration directory */usr/allegrostore/as-v1.0.54*. If the image is in another directory, that should also be in the PATH.

```
setenv PATH $OS_ROOTDIR/bin:/usr/allegrostore/as-v1.0.54:$PATH
```

These variables are also listed in chapter 1 **Installation guide**.

About deadlock resolution

AllegroStore resolves a deadlock by selecting one of the transactions involved and forcing it to roll back. After the transaction releases its locks on the database, the other transaction(s) can proceed. When this happens, the system automatically retries the transaction which was selected to roll back.

When a deadlock occurs, the transaction is automatically retried until either it completes successfully or the maximum number of retries has occurred. The default number of retries is 10.

How does AllegroStore decide which transaction to select? It uses a *Deadlock Victim algorithm*, which can be either `Current`, `Age`, `Oldest`, `Work`, or `Random`. The value is specified in the parameters file for the Database Server,

`$(OS_ROOTDIR)/etc/`*`hostname`*`_server_parameters`

where *`hostname`* is the name of the machine on which the Server is installed. You may find a line like this one in the parameters file:

```
Deadlock Victim: age
```

If there is no such line, then the code default (`Work`) will be used. You will need to edit this file, either changing or adding such a line, if you wish to change the default. For information on how the other algorithms select a victim, see section 8.6 **Server parameters**.

About shrinking the transaction log

If you are using normal files for databases then you should also have a file called the *transaction log*. The location of the transaction log is specified when AllegroStore is installed. You can find out where it is by looking in the file named by the following:

`$OS_ROOTDIR/etc/`*`hostname`*`_server_parameters`

The transaction log holds temporary information during a transaction. It can grow large if many changes are made during a single transaction. ObjectStore will never shrink the file automatically, even if there are no active client processes. In rare cases where the file grows large and you want to recover disk space, you can shrink the transaction log. (Note that the file only grows if you do transactions which access a lot of data. If you never do such transactions, the file will never grow large.)

To shrink the transaction log, you have to shut down the ObjectStore server and tell the server to reallocate the log file. These are the steps:

```
% su                                - this must be done as root
# $OS_ROOTDIR/bin/ossvrshtd hostname
                                    - shut down the hostname server
# cd $OS_ROOTDIR/lib                - change directories
# ./osserver -ReallocateLog         - performs the shrinking process
# cd ../etc                         - change directories
# ./osserver                        - restart the server
# exit                              - stop being root
```

You're done.

New arguments to open-database and with-database allowing instance/pointer/segment allocation

The AllegroStore 2.0 release contains some experimental **open-database** and **with-database** arguments that allow the Astore developer more control over instance/pointer/segment allocation. The arguments will be useful in situations where large database size and/or large blob data size lead to ObjectStore address full errors.

Before using these arguments, make sure you are doing the following:

1. Set the OS_AS_SIZE environmental variable in your client process as large as possible.
2. Set the OS_RELOPT_THRESH environmental variable to 0.
3. Set the OS_INBOUND_RELOPT_THRESH environmental variable to 0.
4. Set the OS_FORCE_DEFERRED_ASSIGNMENT environmental variable to 1.

Note that steps 2, 3, and 4 are recommended only when you are experiencing ObjectStore address full errors.

Here is an example that uses the n experimental arguments:

```
(defun add-some (instances blob-size max-per-segment max-blob-bytes)
   (with-database (db "foo.db" :if-exists :supersede
                               :instances-per-segment max-per-segment
                               :blob-bytes-per-segment max-blob-bytes)
      (let ((tick (floor (/ instances 10))))
         (dotimes (i instances)
             (when (= (mod i tick) 0)
                 (format t "~%~s~%" i))
            (with-transaction ()
                 (make-instance 'foo :id i
                     :foo (make-instance 'blob :size blob-size)))
    ))))
```

The two arguments are the keyword arguments :instances-per-segment and :blobs-per-segment.

The :instances-per-segment argument specifies the maximum number of instances and/or lists per ObjectStore database segment. When the value is exceeded during object creation, AllegroStore automatically adds a new segment. The default value is 50,000.

The :blob-bytes-per-segment argument specifies how the maximum number of blob bytes per blob data segment. The default value, 0, directs AllegroStore to

place blob data areas in instance and list segments. For positive values, blob data areas are placed in separate segments. This argument is particularly important when large blobs are responsible for address full errors.

Note the following important matters:

1. These values are set once, and only once, for each database, at database creation time. Specifying (or not specifying) these arguments during subsequent database operations will not change the original values.
2. The asdump and asrestore utilities currently are unaware of this feature; if you dump and then restore a database, the database will have default segment count and blob data segment attributes.

Here is some general advice about using this functionality:

- When you are creating huge blobs, use the :blob-bytes-per-segment argument.
- If possible, limit the size to less than 10% of your OS_AS_SIZE value.
- Try to choose an :instances-per-segment argument value that keeps the total size of the blob data pointed to by instances in that segment below 10% of your OS_AS_SIZE value.
- If your blobs are of varying sizes, either try to divide them into separate databases or use arguments appropriate for your largest blob. If this results in too many segments (you may see performance problems when there are more than 5000 segments), use smaller values and include a dummy slot where you can place a list of lists to pad instances with larger blobs.

7.2 Definitions

In this section we describe the functions, macros and variables that make up the programmer's interface to AllegroStore. Unless otherwise indicated, symbols used to name the various objects are in the `allegrostore` package (nickname `astore`).

7.2.1 Variables

These variables hold information about the current AllegroStore image.

`*allegrostore-version*` **[Variable]**

■ The value of this variable is a string holding the current AllegroStore version number.

`*as-config-path*` **[Variable]**

■ A list of directories to search for the location of the AllegroStore configuration database. Allegrostore must locate this file before opening its first database. See the information under the heading **About the configuration database** in section 7.1 for details.

`*db*` **[Variable]**

■ The value of `*db*` is the implied argument to functions such as **`make-instance`** when they are creating an object in a database. The **`with-database`** macro binds the symbol `*db*` to the database which it opens.

■ If you open a database using **`open-database`**, then you must arrange that `*db*` is set to the value returned by **`open-database`**. This is typically done with the function **`set-current-database`** as in the following:

```
(set-current-database (open-database "file.db")
```

where *file.db* names the database file.

`astore::*channel*` **[Variable]**

■ This variable is no longer supported. In earlier releases, it was used in connection with the channels facility. On platforms that do not include `:os-threads` in `*features*`, in earlier releases, AllegroStore supported channels as a method of multiprocessing, but channels are not supported in release 2.0. The same functionality, with better overall performance, should be possible by running separate AllegroStore sessions that communicate via the AllegroStore notification facility. See section 6.24 for information on notifications.

`astore:*allegrostore-release-heap*` **[Variable]**

■ On platforms with `:os-threads` on `*features*`, this variable controls whether the Lisp heap is released when making an ObjectStore call. When `nil`, the Lisp heap is not released; when non-`nil`, the Lisp heap is released. See section **6-16 Multiprocessing** for more information. On platforms without `:os-threads` on `*features*`, this variable is ignored.

7.2.2 Databases: saving and restoring

The following two shell commands (not Lisp functions) can be used convert an existing database (version 1.1 or version 1.2) into a platform independent ASCII file and to convert such a file to a version 1.2 database.

asdump [Program]

Arguments: [-v] [-o *dump-file*] *database-file*

- Reads the allegrostore database-file and generates a platform independent ASCII text dump of the database contents. The companion program **asrestore** can read the dump file and rebuild the allegrostore database-file.
- **Note**: for **asdump** to work, the entire database must be small enough to fit into the available persistent virtual memory address space. See the documentation for the ObjectStore environment variables OS_AS_START and OS_AS_SIZE on how to control the address space. (They are documented under the heading **About shell environment variables** in section 7.1 above.)
- The optional -v argument specifies verbose operation. Comments about the progress of the dump will be sent to standard error (stderr). If a *dump-file* is not specified as the value of the -o option or is specified as just a hyphen (-), then the dump output is sent to standard output (stdio).
- **asrestore** in version 1.3 can read files dumped by **asdump** in version 1.2.

asrestore [Program]

Arguments: [-v] -o *database-file* [*dump-file*]

- Reads an ASCII dump-filed created by **asdump** and regenerates an AllegroStore database. If no dump-file is specified, standard-input is used (thus allowing **asdump** and **asrestore** to be piped together).
- The optional -v argument specifies verbose operation. Comments about the progress of **asrestore** will be sent to standard error (stderr).
- The -o argument specifies the database-file to create. A value must be supplied.

7.2.3 Database manipulation

with-database [Macro]

Arguments:
```
(db-var filename &key :if-does-not-exist
:if-exists :mode :use :warn
:read-only :mvcc) :instances-per-segment
:blob-bytes-per-segment
&rest body
```

- Opens (or creates) a database named *filename* and makes it the current database for the duration of the *body*. It creates an object that represents a connection to the database and binds the variable *db-var* (which should be a symbol) to that object. When control leaves the dynamic extent of the **with-database** macro the database is closed.

Keyword	Possible values	Effect
:if-does-not-exist	:create :error	Specifies what action to take if a database with the given filename does not exist. :create (the default) makes a *filename* database file on the disk. :error signals an error.
:if-exists	:open :error :supersede	Specifies what action to take if a database with the given filename already exists. :open (the default) will open the *filename* database. :error will signal an error when the database exists. :supersede will delete the existing database and create a new one.
:mode	#o666 (which is an octal number)	This is the (default value) file mode used for creating the file in Unix. On Unix systems, the file mode is combined with the user's umask to determine the default value for creating the file. (see not at bottom of table for further information). #o666 allows everyone to read and write the database. MS-DOS filesystems don't implement group and other permission, so this argument is ignored.
:use	:ask :db :memory	When opening an existing database, the schema in the database and the schema in Lisp's memory are compared. If there are any differences, then this variable will determine how the differences are resolved. If :ask (the default) is specified, then a continuable error will be signaled to permit the user to select the correct definition. If :db is specified, then the version in the database will be used. See the notes on :warn below. If :memory is specified, then the version in memory will be used and the database will be modified to reflect the new class definition. See the notes on :warn below.

Keyword	Possible values	Effect
`:warn`	`t` `nil`	When `:use` keyword is given the `:db` or `:memory` option and, upon opening a database, a conflict is found and resolved, then this argument determines whether a warning message is printed. When `:warn` is `t` (the default), the warning message is printed. When `:warn` is `nil`, no warning message is printed.
`:read-only`	`t` `nil`	When t, an attempt to write to the database will result in an AllegroStore error. The default value is nil.
`:mvcc`	`t` `nil`	When t, the database is opened in MVCC mode.
`:instances-per-segment`	non-negative-integer	This argument specifies the maximum number of instances and/or lists per ObjectStore database segment. When the value is exceeded during object creation, AllegroStore automatically adds a new segment. The default value is 50,000. See the information under the heading *New arguments to open-database and with-database allowing instance/pointer/segment allocation* in section 7.1.
`:blob-bytes-per-segment`	non-negative-integer	The argument specifies how the maximum number of blob bytes per blob data segment. The default value, 0, directs AllegroStore to place blob data areas in instance and list segments. For positive values, blob data areas are placed in separate segments. This argument is particularly important when large blobs are responsible for address full errors. See the information under the heading *New arguments to open-database and with-database allowing instance/pointer/segment allocation* in section 7.1.

`with-current-database` [Macro]

Arguments: *db* `&rest` *body*

■ While more than one database can be open at the same time, there is a notion of a *current* database. All object creation is done in the current database. The current database is typically bound by using the **`with-database`** macro. You can change the current database with the **`with-current-database`** macro.

■ The value of *db* is the database to become current. The only valid values for *db* are those values bound to the *db-var* argument to **`with-database`** and those returned by **`open-database`**.

open-database [Function]

Arguments: *filename* &key :if-does-not-exist :if-exists :mode :use :warn :read-only :mvcc :instances-per-segment :blob-bytes-per-segment

■ Opens or creates a database and returns an object that references the database.

■ The keyword arguments to **open-database** are the same as **with-database** (defined above). There are two major differences. First, **open-database** opens a database and keeps it open. Your program must close it (with **close-database**). Second, **open-database** does not make the newly-opened database the *current* database. You should use **set-current-database** or **with-current-database** for that.

Note that:

```
(with-database (var file)
      . body)
```

is equivalent to:

```
(let ((var (open-database file)))
   (unwind-protect
      (with-current-database var
            . body)
      (close-database var)))
```

close-database [Function]

Arguments: *db*

■ Close the connection to the database. This function only needs to be called when the database was opened with **open-database**. If the database was opened with **with-database**, then it will be closed automatically as soon as the **with-database** form completes.

set-current-database [Function]

Arguments: *db*

■ Make *db* be the current database. *db* should be a value returned by **open-database** or bound by **with-database**.

Program to verify database consistency

asverify [Program]

Arguments: *[db1]+*

■ **asverify** will check the given databases for consistency. If it prints any diagnostics then there may be a problem in the database. **asverify** may also describe situations that are unusual but not necessarily incorrect. If you see any diagnostics printed please send mail to allegrostore-bugs@franz.com.

■ The following transcript shows **asverify** in action. User input is in **bold**. Databases *t2.spl* and *v1.spl* had no problems while database *spj2.spl* is inconsistent.

```
% asverify *.spl

verifying t2.spl
```

```
verifying v1.spl

verifying spj2.spl
            Illegal typecode 3801856 (0x3a0300) found in object
object at 302090ac </crow/c:/dbs/spj2.spl | 14 | a90ac>
         backpointer item 3
      In instance object at 301b98d4 </crow/c:/dbs/spj2.spl | 14 |
698d4>
   in instance number 41 of type DB-COMMON::CHANGE-OBJECT
            Illegal typecode 3801856 (0x3a0300) found in object
object at 302090ac </crow/c:/dbs/spj
2.spl | 14 | a90ac>
         backpointer item 1
      In instance object at 30208adc </crow/c:/dbs/spj2.spl | 14 |
a8adc>
   in instance number 76 of type DB-COMMON::CHANGE-OBJECT
In database SPJ2.SPL
```

7.2.4 Schema manipulation

`defclass` [Macro]

Arguments: *class-name superclasses slots* &rest *options*

Package: common-lisp

■ The **`defclass`** macro defines CLOS classes. It is fully described in the ANSI Common Lisp spec. Only the new slot keywords added by AllegroStore for persistent classes will be described here.

A persistent class is defined by specifying the following as one of the **`defclass`** options

```
(:metaclass persistent-standard-class)
```

■ When a class with metaclass `persistent-standard-class` (or a subclass of same) is defined, the following standard slot options either have additional possible values or are restricted in some fashion:

Keyword argument	Possible values and effect
`:allocation`	In addition to standard possible values, `:instance` and `:class`, can be `:persistent` or `:persistent-class` to specify that this slot's value should be stored in the database. The default is `:instance`. See section 6.6 **Slots** for more detailed information.
`:initform`	If the `:allocation` type is `:persistent` or `:persistent-class`, then this argument cannot be an expression containing a persistent object as a literal. It can be an expression that computes or locates and then returns a persistent object.

Keyword argument	Possible values and effect
`:type`	If the type is specified as `(set-of X)` then this is the same as `:type X :set t`. AllegroStore doesn't currently signal an error if you attempt to store a value of incorrect type in a slot.

The following new options are also available:

Keyword	Function
`:set`	If true, then the value of this slot is considered to be an unordered collection of objects. The value of the slot will be returned as a list. Cannot be true when `:unique` is true. The main benefit to specifying a set slot is that when an inverse function is specified, it acts on each value in the set, rather than on the set as a whole (so the whole set does not have to be read into Lisp from the database).
`:unique`	If true, then this is a declaration that the value of this slot is different for each object. (A slot storing a serial number is an example, since serial numbers are typically unique). Set slots cannot be unique. AllegroStore verifies that declaration is valid. If an inverse function is specified for a slot declared as `:unique`, then the inverse function will return a single value (rather than a list of values).
`:inverse`	This keyword names a function which, given a value `X`, will return a list of objects from the class being defined which have `X` stored in this slot. (A list will be returned even if there is only one such object, unless `:unique` is specified -- see below). If `:unique` is also specified as true, then the return value from the inverse function will be a single object, not a list of objects.

For examples of **`defclass`** usage, see chapter 5 **Tutorial** and chapter 6 **Programmer's guide**.

Executable subforms of the defclass macro and lexical environments

There are two places where arbitrary executable subforms can appear in a **`defclass`** form. One is the :initform slot option, and the other is as a value in the :default-initargs class option. When these forms are executed (typically inside a call to **`make-instance`**) Common Lisp specifies that they are executed in the lexical environment of the **`defclass`** form in which they appear. The MetaObject Protocol details how the **`defclass`** macro accomplishes this; each such form is wrapped in a zero-argument lambda function that is funcalled when the value is needed.

In normal Common Lisp these functions can modify the lexical environment automatically because when a class metaobject is created it and its associated initialization functions exist in the Lisp heap along with any objects referenced by the lexical environment. But a class metaobject that is part of a persistent database schema must also be stored in the database and later retrieved, possibly into a different executing Lisp image. This presents a problem, because Lisp functions are not externalizable objects that can be stored in a database. There are two reasons: Firstly, a function captures the surrounding lexical environment, and this environment would also have to be stored in the database. Secondly, a function (especially if compiled) is not portable between different hardware platforms or different Lisp releases, thwarting the goal of heterogeneous operation.

AllegroStore works around this in a way that covers most typical usage. The schema (i.e. the class metaobject) captures both the form and the function created from it. When the class metaobject is stored in the database schema, the function is not stored, and when a class metaobject is reconstructed from the database copy, the function is reconstituted by freshly wrapping that initform or default-initarg in a new function. What is lost in this process, of course, is the lexical environment in which the original **`defclass`** form appeared. Therefore, slot initforms and class default-initarg forms in persistent class definitions may not refer to their surrounding lexical environment. This is not usually a problem, since it is normal coding practice for defclass forms to appear at top level anyway, i.e., in a null lexical environment.

`set-schema` [Function]

Arguments: `&key :auto-class-addition :classes :delete-function :exact :from-db`

■ **`set-schema`** sets the schema for the current database or it sets the auto-class-addition flag for the database (see next bullet down), or both. This function may be run only while a database is open and current. This function can radically modify a database, particular when `:exact` is true, since that may result in classes and all instances of those classes being deleted from the database. You may wish to back up a database (with, e.g., **`oscp`**) before applying **`set-schema`** to ease recovery in case you do not get what you wanted.

This function does permit you to customize a database, and to prune the class definitions in it. But note that it is not necessary to call this function. AllegroStore will manage the schema perfectly well without this function ever being called (except unnecessary class definitions may be stored, thus wasting some space).

■ When a new database is opened by AllegroStore (with **`with-database`** or **`open-database`**), it has the property that definitions of new classes are automatically stored in the database when an object of the new class is stored in the database. The examples in chapter 5 **Tutorial** make use of this property.

The `:auto-class-addition` argument controls this property in the current database. If **`set-schema`** is called with `:auto-class-addition` true, new definitions will be stored automatically. If the value is `nil`, then an error will be signaled if an attempt is made to store an instance of a new class in the database.

The default of `:auto-class-addition` depends on the value of `:exact`. If `:exact` is specified, `:auto-class-addition` defaults to the logical opposite of `:exact`. If `:exact` is also not specified, `:auto-class-addition` defaults to `t`.

■ The `:classes` argument should be a list of classes or class names to store in the database. Additional classes can easily be added with this argument. But note that if `:exact` is true, the classes in the list that is the value of `:classes` will be the only classes defined in the database after **`set-schema`** completes, with all other classes and all instances of those classes deleted. `:classes` can not be specified when `:from-db` is specified.

■ One use of **`set-schema`** is to customize the database to contain a specific set of class definitions and no more. That customization is performed with the `:exact` argument. `:exact` defaults to `nil`, but if specified true, **`set-schema`** will prune the database as necessary to remove all classes not specified in the `:classes` list (or present in the `:from-db` database, if that argument is used) and to further delete all objects that are instances of the deleted classes. (This means that the database may be radically changed by **`set-schema`** called with `:exact` true.)

Note that AllegroStore will not check whether the set of classes is consistent (that is, all necessary classes are defined, including classes of the values of slots of other classes). It is the programmers responsibility to ensure that the class list is consistent.

Note too that **`set-schema`** can be used to delete specific classes from a database and that there is no other way to delete classes from a database. If you want to remove class `foo` from the database (and you know that doing so will not lead to any inconsistency), you can do the following (we set `:auto-class-addition` to `t` to preserve the default behavior, but that is not necessary for the deletion to work):

```
(with-database (db "myfile.db")
  (let ((class-list (schema db)))
    (set-schema :auto-class-addition t :exact t
                :classes (delete (find-class 'foo) class-list)))))
```

■ `:delete-function` only has effect if `:exact` is true. In that case, as we said above, all instances of classes which are deleted from the database are also deleted from the database. They are deleted with **`delete-instance`**, but if this argument is specified, it should name a function that accepts one argument. That function will be called on each instance slated for deletion before **`delete-instance`** is called.

■ The `:from-db` argument allows you to copy a schema from another database. The value should be a string naming a database. You cannot specify both `:classes` and `:from-db`.

dump-schema [Function]

Arguments: *`database`* `&key :file`

■ Write the schema for the given database to the named file if `:file` is given, else to `*standard-output*`. The schema is a sequence of **`defclass`** forms that describe the classes in the database, followed by a **`set-schema`** expression that lists all of the classes just dumped.

■ The *`database`* argument can be:

- `nil`, in which case the current database is described
- a filename, in which case the database with that name is opened and described
- a database object (such as is returned by **`open-database`**), in which case that database is used

schema [Function]

Arguments: *database*

■ Returns the list of class objects which make up this database's schema. The possible values for the *database* argument are the same as those for **dump-schema** defined just above.

describe-db [Function]

Arguments: *dbname*

■ Prints out information on the database named *dbname*. The format is designed to be more human readable than **dump-schema** but the information is much the same.

7.2.5 Transactions

with-transaction [Macro]

Arguments: (&key :top-level :read-only) &rest *body*

■ Start a transaction, evaluate the *body*, and then commit the transaction. If control leaves the transaction other than by the completion of the *body* forms, then the transaction will be rolled back and all changes made during the transaction will be undone.

■ Changes to persistent data may take place within a **with-transaction** form.

■ If the :top-level keyword argument has the value t, then AllegroStore will signal an error if the **with-transaction** form is within the dynamic scope of another transaction. When a top-level **with-transaction** completes, the database transactions are guaranteed to have committed.

■ AllegroStore supports nested **with-transaction** forms but does not support nested transactions. If a **with-transaction** is encountered while within the dynamic scope of another **with-transaction**, then the inner **with-transaction** is converted into a **progn**.

■ If the value of the :read-only keyword argument is true, AllegroStore will signal an error when an attempt is made to write to any database during the transaction.

■ In release prior to 2.0, this macro took a :channel keyword argument. The channels facility is no longer supported. See section 6.24 for information on notifications, which are in part an improved replacement for channels.

transaction-active-p [Function]

Arguments:

■ returns true if there is a transaction in progress.

begin-transaction [Function]

Arguments: *(&key :top-level :read-only)*

■ Starts a transaction.

■ Changes to persistent data may take place after a successful call. A subsequent commit-transaction call commits changes to the database; a subsequent abort-transaction call rolls back any changes.

■ The :top-level keyword and :read-only keyword behave as they do in the with-transaction macro.

■ A successful begin-transaction call will return a non-nil result when a transaction is not currently active. If there is a transaction currently open, and :top-level is nil, then begin-transaction will return nil. Note that, as with **`with-transaction`**, nested transactions are not supported.

`commit-transaction` [Function]

Arguments:

■ Commits the current transaction. this function signals an error if there is no current transaction or if the current transaction was initiated in a with-transaction form.

`abort-transaction` [Function]

Arguments:

■ Rolls back the current transaction. This function signals an error if there is no current transaction or if the current transaction was initiated in a with-transaction form.

7.2.6 Object manipulation

`make-instance` [Generic function]

Arguments: *`class`* `&key [...]`

Package: `common-lisp`

■ Persistent objects are created with **`make-instance`**, just like other CLOS objects. All objects of metaclass `persistent-standard-class` are persistent even if they don't have any persistent slots. Persistent objects can only be created within a transaction (i.e. in the body of a **`with-transaction`** form).

The *`class`* argument can be either a symbol naming the class or a class object. See a standard reference for a full description of **`make-instance`**.

`slot-value` [Function]

Arguments: *`object slot-name`*

Package: `common-lisp`

■ **`slot-value`** returns the value of a slot of a persistent object, just as it does for normal CLOS objects.

■ `(setf slot-value)` stores a value in a persistent object's slot.

slot-svref [Method]

Arguments: (*instance* persistent-standard-class) *slot-name index*

■ Returns the *index*'th value from the vector stored in the slot *slot-name* of *instance*. Using **slot-svref** is more efficient than **svref** and **slot-value**, since the entire vector doesn't have to be read into Lisp memory to access a single value from it. (setf slot-svref) will set a slot in the vector.

database-of [Generic function]

Arguments: *obj*

database-of [Method]

Arguments: (*obj* persistent-standard-object)

■ Returns an object corresponding to the database in which the object resides.

delete-instance [Generic function]

Arguments: *obj*

delete-instance [Method]

Arguments: (*obj* persistent-standard-class)

■ Removes *obj* from the database and removes all pointers to it from other database objects.

If scalar slot S of persistent object X points to *obj*, then after *obj* is deleted, slot S will be unbound unless the :initform for slot S is the expression nil, in which case slot S's value will be set to nil.

If a set-slot named T of persistent object X includes a pointer to *obj*, then that reference to *obj* in the set will be removed.

■ Automatic removal of references to the deleted object from other objects is necessary to preserve the referential integrity of the database. If a program wants to determine which objects will be affected when an object is deleted, it can add a :before method to **delete-instance**, and in that :before method it can call **map-references** or **collect-references**, which operate on or list (respectively) every reference to *obj* in the database (see section 7.2.8 below).

■ See the section 6.14 **Referential integrity**.

preserve-pointer [Function]

Arguments: *obj*

■ Objects in a database may be moved about by the database manager. (Objects are moved for a variety of reasons which we will not go into here.) During a transaction while an object is referenced, it will not (of course) be moved. Thus during a transaction, when a persistent object is read from the database, a transient object is created that references the persistent object and that reference is valid for the duration of the transaction. However, when the transaction completes, the reference to the persistent object may soon become invalid.

The **preserve-pointer** function makes an object accessible during a subsequent transaction. It does this not by preventing the object from being moved but by storing the object location is a table that is kept current.

slot-valid-p [Generic function]

Arguments: *obj slot-name*

■ Returns `t` if it is permitted to *try* to access the value of *slot-name* from *obj*. The only time that this function will return `nil` is if *slot-name* is the name of a persistent slot and the persistent pointer can't be referenced. This occurs when:

- the database it points to is closed
- there is no active transaction
- the transaction in which the persistent pointer was created is no longer current and **preserve-pointer** wasn't called on this object.

■ **Note** that the function *cannot* be used to check whether a slot identified by *slot-name* exists. If *slot-name* doesn't exist in *obj*, **slot-valid-p** will return `t`. If you want to know whether a slot exists at all, use the standard CLOS function **slot-exists-p**.

slot-cons [Method]

Arguments: (*instance* persistent-standard-object) *slot-name new-value*

■ **cons**'s the given value onto the current value in the given slot of the given object. If the slot is a set slot then the *new-value* is added to the set. This function returns `nil`.

slot-delete [Method]

Arguments: (*instance* persistent-standard-object) *slot-name value-to-delete*

■ Deletes *value-to-delete* from the value in the slot *slot-name* of *instance*. If *slot-name* is a set slot then the value is removed from the set.

7.2.7 Query language

for-each [Macro]

Arguments: (&rest *clauses*) &rest *body*

■ Where *clauses* is a list of one or more *var-bind* clauses, zero or one *filter* clause and zero or one *subclasses* clause.

A var-bind clause has one of the forms:

```
(var class-name)
(var (accessor var2))
(var (slot-value var2 'symbol))
```

Where:

var-bind clause element	Contents
var	is a symbol.

`var2`	is a symbol which appears as the first object in a `(var class-name)` expression in a preceding clause. -- thus `((a auto) (b (auto-wheels a))`
`class-name`	is the name of a persistent standard class.
`accessor`	is the name of an accessor for a persistent slot.

A filter clause has the form:

```
(:where expr)
```

where `expr` is any Lisp expression.

A subclasses clause has the form:

```
(:subclasses t-or-nil)
```

■ **`for-each`** acts like **`let*`** in Lisp (in that the bindings are sequential), but differs from **`let*`** in that instead of binding each variable to the value of an expression, each variable is bound to a sequence of objects of a given class (or subclass) or of a given set. Specifying `nil` in the subclasses clause directs **`for-each`** to only look for objects of the specified class, not its subclasses.

■ For example

```
(for-each ((x foo)) (print x))
```

binds `x` to each object in the current database whose class is `foo` or a subclass of `foo` and then evaluates the body, which in this case just prints the value of `x`.

Given the definitions:

```
(defclass wheel ()
   ((brand :allocation :persistent))
   (:metaclass persistent-standard-class))
(defclass car ()
   ((wheels :type (set-of wheel)
            :allocation :persistent :accessor wheels))
   (:metaclass persistent-standard-class))
```

we can execute a query to print all wheels currently on all the cars:

```
(let ((wheel-list nil))
  (with-transaction ()
   (for-each ((c car) (w (wheels c))) (push w wheel-list))
    )
   (print wheel-list))
```

(Notice that we collected all wheels into a list and then printed the list outside the transaction. We did so to avoid multiple printings of wheel objects caused by transaction rollbacks. See section 6.5 **Transactions** for details on rollbacks and forms with side effects.)

■ Despite the fact they are called the same thing, the **`for-each`** `:where` clause is very different from the **`for-each*`** `:where` clause. Again, the value of `:where` in **`for-each`** can be any Lisp expression. See below under the discussion of **`for-each*`**.

for-each* [Function]

Arguments: *function class-name* &key :where :subclasses :start-block :block-count

■ The following table describes the arguments. More detail is given below.

Argument	Value
function	specifies a function of that accepts one argument
class-name	specifies the name of a persistent standard class.
:where	Filters the objects examined. Its value must evaluate to a list that is an expression used to select the objects of interest. The format of the list is described below.
:subclasses	Controls whether subclasses of *class-name* are examined. Its value is a boolean. If true (the default), **for-each*** will iterate over the class specified by *class-name* and over all its subclasses. If nil, **for-each*** iterates over the class specified by *class-name* only.
:start-block	specifies the instance block where processing begins. The default value is 0, which specifies the current instance block. Instance processing begins with the most current instance block and works backwards. There are at most 30 instances per block. This keyword may be useful when working with very large databases. See **instance-count** documented immediately below.
:block-count	specifies the number of instance blocks to process when the value is a positive integer. The default value, :all, specifies that all instance blocks starting with and including the :start-block instance block be processed. See **instance-count** documented immediately below.

Note that **for-each*** is a function, not a macro, so the arguments are evaluated. **for-each*** calls *function* on each instance of the class specified by *class-name* and all its subclasses (unless :subclasses is nil) in the current database.

■ The :where clause has a different form than in the **for-each** macro described above. The form of the :where clause is a list of *selector* clauses. Each selector clause is a list of three elements:

(*accessor-name binary-function other-argument*)

A selector clause is considered true if the given *binary-function* returns true when called on these two values:

1. the value taken by applying the function named by *accessor-name* to one of the objects that **for-each*** is iterating over (i.e. instances of *class-name* and, if :subclasses is true, its subclasses)
2. the *other-argument* value.

The :where clause is considered true if all the selector clauses are true. For example:

```
(for-each* #'dowhatever 'foo
   :where '((foo-a < 3)
            (foo-b eq zip)))
```

takes each `foo` object `obj` and tests:

```
(and (< (foo-a obj) 3) (eq (foo-b obj) 'zip))
```

Look closely at the example. The elements of the selector clauses are typically constants. You can arrange to have them evaluated (using backquotes and commas in the standard Lisp way) but that makes **`for-each*`** less efficient and is usually not necessary. Note too that the format of selector clauses is un-Lisp-like, since it puts the predicate between rather than before the arguments.

`instance-count` [Function]

Arguments: *`persistent-standard-class database`*

■ This function returns the non-negative integer count of instances of the class specified by the *`persistent-standard-class`* argument (the argument's value may be a class object or a symbol naming a class) in the database specified by the *`database`* argument. This function may be useful when debugging and when deciding on values for the block arguments to **`for-each*`**.

`for-each-class` [Macro]

Arguments: (*`var`* `&optional` *`db`*) `&rest` *`body`*

■ Evaluate the *`body`* repeatedly with *`var`* bound to each class stored in the schema for the current database (or the database specified by *`db`*, which must be a database object).

`eqo` [Function]

Arguments: *`obj1 obj2`*

■ In earlier versions, where the same persistent object might not be **`eq`** to itself. this function determined whether two objects were or were not the same. In version 1.2 and later, the same persistent objects are always **`eq`**. This function still works but using **`eq`** instead is preferred.

`equalo` [Function]

Arguments: *`obj1 obj2`*

■ Returns `t` if the two objects are **`equal`**. In version before 1.2, the test for **`eq`**'ness is done with the **`eqo`** function but in version 1.2 and later, the test is done with **`eq`** because the same persistent object is always **`eq`** to itself. (Recall that **`equal`** is defined recursively: two objects are **`equal`** if either they are **`eq`** or they are both lists with **`equal`** contents.) Still supported to maintain existing code but using **`equal`** is preferred.

retrieve [Function]

Arguments: *class* &key :where :subclasses

■ Returns a list of all objects of a given *class* (and its subclasses or not as the :subclasses argument is t -- the default -- or nil) that satisfy the :where clauses. This function is similar to **for-each*** except rather than applying an argument *function* to each item as **for-each*** does, it bundles all items into a list. The *class*, :where, and :subclasses arguments are all as for **for-each***. **retrieve** could be implemented in terms of **for-each*** as follows:

```
(defun retrieve (class &key where subclasses)
  (let ((items nil))
    (for-each* #'(lambda (x) (push x items))
            class :where where :subclasses subclasses))))
```

■ **for-each*** should generally be used if possible in preference to **retrieve** since retrieve must copy all items that it collects into Lisp at one time while for-each* only brings them in one at a time, potentially saving space.

7.2.8 References

AllegroStore keeps track of all the objects in the database that point to a given persistent object. We call a pointer from one persistent object to another persistent object a *reference* to that object.

collect-references [Function]

Arguments: *obj*

■ *obj* must be a persistent object. This function returns a list of all references to *obj*. Each reference descriptor is a list, such as (X a b c d) where X is a persistent object and a, b, c, and d are the names of slots in the X object whose values contain references to *obj*.

The slot may not point to *obj* directly, it may hold a Lisp object (such as a list, vector, or hash table) that points to *obj*.

See also the function **map-references**.

map-references [Function]

Arguments: *function obj*

■ *obj* must be a persistent object. *function* must be a function that accepts two arguments. It will be passed a persistent object and a slot name.

map-references finds all of the persistent instance slots of all of the persistent objects in the database that point at *obj* and calls *function* on each of them.

7.2.9 Object identifiers

An *object identifier* is a number that denotes a particular persistent object during a certain transaction. The number is useful for denoting a persistent object in a transient hash table.

`object-id` [Function]

Arguments: *obj*

- Returns the object identifier.

`object-from-object-id` [Function]

Arguments: *object-id*

- Returns the persistent object associated with the object identifier.

7.2.10 Persistent hash tables

Hash tables can be quite large, so it can take a fair amount of time to copy them from a database into a Lisp image (converting the database object into a lisp object) and then copying the modified hash table back. Therefore, AllegroStore defines a `persistent-standard-class` called `persistent-hash-table`. This hash table is stored entirely in the persistent store. and functions are provided that operate in the database without having to create a Lisp hash table at all.

Persistent hash tables are created with **`make-instance`**:

`make-instance` [Function]

Arguments: `'persistent-hash-table &key :size`

Package: `common-lisp`

- Returns a persistent hash table object. Because **`make-instance`** creates a persistent object, it can only be called when a database is open and a transaction has been started.

`generic-gethash` [Generic function]

Arguments: *key obj*

`generic-gethash` [Method]

Arguments: *key (obj* `persistent-hash-table`*)*

- Returns two values. The first is the value associated with the *key* in the given persistent hash table or `nil` if there is no value associated with *key*. The second value is `t` if a value was found with the given *key*, otherwise the second value is `nil`. (Thus you can distinguish between the value `nil` associated with *key* and no value associated with *key*.)
- In order to store in the persistent hash table, use:

```
(setf (generic-gethash key obj) value)
```

`generic-remhash` [Generic function]

Arguments: *key obj*

`generic-remhash` [Method]

Arguments: *key* (*obj* `persistent-hash-table`)

- Removes the given *key* and its associated value from the persistent hash table. Returns `t` if a value was found with the given *key* in the table

generic-maphash [Generic function]

Arguments: *function obj*

generic-maphash [Method]

Arguments: *function (obj* persistent-hash-table*)*

■ Calls the given *function* on each key and value pair in the persistent hash table *obj*.

generic-puthash-push [Generic function]

Arguments: *key ht value*

generic-puthash-push [Method]

Arguments: *key* (*ht* persistent-hash-table) *value*

■ Pushes *value* onto the value already stored in the table *ht* and returns *value*.

Persistent hash tables as slot values

You can store a persistent hash table in a slot of any persistent object. These hash table are different than the ones created by (make-instance 'persistent-hash-table ...) in that these tables can't exist on their own, they only exist as the value of a certain slot of an object. You *cannot* use **slot-value** to extract one of these tables and then use **(setf slot-value)** to store the value into another object.

make-slot-hash-table [Method]

Arguments: (*instance* persistent-standard-object) *slot-name* &key *size*

■ Creates the hash table in the specified slot.

slot-gethash [Method]

Arguments: (*instance* persistent-standard-object) *slot-name key*

■ Returns the value from the hash table (and a second value of t if the value was found). This function can be **setf**'ed.

slot-puthash-push [Method]

Arguments: (*instance* persistent-standard-object) *slot-name key value*

■ Pushes the given value onto the current value in the hash table for the given key. This function is more efficient than

```
(setf (slot-gethash inst 'foo key)
      (cons (slot-gethash inst 'foo key) val))
```

since the value from the slot isn't copied into Lisp's memory before being updated. This function returns *value*.

slot-maphash [Method]

Arguments: (*instance* persistent-standard-object) *slot-name function*

slot-remhash [Method]

Arguments: (*instance* persistent-standard-object) *slot-name key*

■ These functions are the analogs of **maphash** and **remhash** for slot-valued persistent hash tables.

7.2.11 Blobs

Blobs are a persistent-standard-class you can use to allocate raw persistent storage blocks without creating a parallel Lisp object. Blobs are appropriate when you require large amounts of persistent storage that you can treat as foreign C memory.

Like persistent hash tables, blobs allow you to operate on portions of the allocated persistent memory without doing conversions back and forth between database format and Lisp object format.

Blobs are created with **cl:make-instance**. The form

```
(make-instance 'blob :size SIZE :name NAME)
```

Returns a persistent blob object. This operation can only be done when a database is open and a transaction is active. The `:size` argument specifies the number of bytes to allocate, and the `:name` argument specifies an optional name.

blob-data [Generic function]

Arguments: *obj*

blob-data [Method]

Arguments: (*obj* blob)

■ Returns a foreign address suitable as an argument to **ff:fslot-value-typed** function, using the :c allocation type.

blob-size [Generic function]

Arguments: *obj*

blob-size [Method]

Arguments: (*obj* blob)

■ Returns the number of raw persistent bytes allocated when the blob instance was created.

blob-name [Generic function]

Arguments: *obj*

blob-name [Method]

Arguments: (*obj* blob)

- Returns the name specified when the blob instance was created.

blob-read [Function]

Arguments: *blob-instance filename*

- This function reads the data about a blob stored in *filename* (which should have been created with blob-write) and stores it in *blob-instance*.

blob-write [Function]

Arguments: *blob-instance filename*

- This function writes a database *blob-instance* to *filename* so that *filename* is suitable for reading with **blob-read**. If *filename* exists when this function is called, it is overwritten.

7.2.12 Persistent ftypes

The persistent-ftype-array class is a persistent-standard-class you can use to allocate raw persistent storage blocks without creating a parallel Lisp objects. They are more powerful than blobs because the persistent storage area may contain addresses pointing to other persistent memory addresses.

persistent-ftype-array instances are created with **make-instance** after first storing the desired foreign type in the database with the **add-persistent-ftype** method:

add-persistent-ftype [Generic function]

Arguments: *ftype-symbol-or-symbol-list*

add-persistent-ftype [Method]

Arguments: *(ftype symbol)*

- Adds the specified foreign type to the current database. This method must be called within a transaction. The symbol's package is not stored in the database; thus there can be only one foreign type definition stored in the database for any given name.
- See the ACL **def-foreign-type** documentation (start with *doc/cl/ftype.htm*) for more information about defining a foreign type. The following additional restrictions apply to types used as add-persistent-ftype arguments:
- The outer definition must be a :struct composite type. Note that:
  - Anonymous :struct definitions may not be used.
  - :struct fields must be named.
  - Structure names and field names must be legal C names (for example, they cannot contain hyphens, i.e. "-").

- `:union` composite types may not be used.
- `:array` composite types must be specified with exactly one dimension.
- `:struct` bit-fields are not supported.

add-persistent-ftype [Method]

Arguments: *(ftypes list)*

■ The list contains symbols identifying foreign types to be added to the database.

make-instance [Function]

Arguments: *'persistent-ftype-array &key :type :n :name*

■ Returns a persistent foreign type array object. This operation can only be done when a database is open and a transaction is active. The :type argument specifies the foreign type used to allocate persistent memory (the foreign type must have been previously stored in the database using add-persistent-ftype), the :n argument specifies the number of array elements of the specified foreign type to be allocated, and the optional :name argument specifies a string to be associated with the object.

persistent-ftype-array-data [Method]

Arguments: *(obj persistent-ftype-array)*

■ Returns a foreign address suitable as an argument to foreign type manipulation facilities, such as ff:fslot-value-typed.

persistent-ftype-array-type [Method]

Arguments: *(obj persistent-ftype-array)*

■ Returns the foreign type associated with the instance's persistent data. A Lisp symbol will be returned; note, however, that the type is stored in the database without a package designation.

persistent-ftype-array-n [Method]

Arguments: *(obj persistent-ftype-array)*

■ Returns the number of foreign type array elements in the instance's associated persistent memory.

persistent-ftype-array-name [Method]

Arguments: *(obj persistent-ftype-array)*

■ Returns the name string specified when the instance was created.

describe-ftype [Function]

Arguments: *database-name foreign-type*

■ Returns a string containing the foreign type definition stored in the named database. This function may be called outside of a transaction and with no databases open. The foreign-type argument is a symbol - it does not have to be in the same package as the package containing the symbol when the foreign type was saved, since package information associated with a foreign type is not saved in the database.

allegrostore::lisp-value-to-pptr [Function]

Arguments: *object*

■ Returns a persistent address suitable for storing the address of a persistent Lisp object in a persistent foreign type array data element's :void * pointer structure member. It is recommended that you use this function only on persistent CLOS instances.

allegrostore::pptr-to-lisp-value [Function]

Arguments: *address db-object*

■ Returns a Lisp object that represents the persistent object stored in ObjectStore. If the ObjectStore object represents a persistent CLOS instance, the returned object will be eq to any existing Lisp variable or instance slot that refers to the same persistent object. The db-object must be the database object associated with the database containing the persistent address. A database object is returned from open-database, is available in with-database, and the *db* variable contains the current database's database object.

set-dbtag [Function]

Arguments: *tagname address*

■ Creates a database tag that enables address retrieval without using CLOS instance navigation. The tagname argument is a string and the address argument is a persistent address. If a tag with the specified name already exists, then the previous associated address is overwritten. A well-behaved database should contain no more than a few dozen tags.

■ For accessing a tag value from an ObjectStore C++ or Java program, see the os_database_root::find() and os_database_root::get-value() methods.

get-dbtag [Function]

Arguments: *tagname*

■ Returns the address associated with the tagname string argument. If the specified tag does not exist, 0 is returned.

7.2.13 Lock timeouts

read-lock-timeout [Function]

■ Returns the number of milliseconds that AllegroStore will wait to obtain a read-lock on a page before it gives up and signals an exception. A value of -1 (which is the default) means that AllegroStore will wait forever, unless it encounters a deadlock. This value can be changed as follows:

```
(setf (read-lock-timeout) newvalue)
```

write-lock-timeout [Function]

■ Returns the number of milliseconds that AllegroStore will wait to obtain a write-lock on a page before it gives up and signals an exception. A value of -1 (which is the default) means that AllegroStore will wait forever, unless it encounters a deadlock. This value can be changed as follows:

```
(setf (write-lock-timeout) newvalue)
```

7.2.14 Notifications

Notifications are an easy and effective way for multiple AllegroStore clients to communicate about persistent instances of interest. The notification facility may also be used as a general client-to-client messaging facility.

Notification class instances are returned by the **notification-receive** function. There are a number of methods for examining notification instance data.

astore::os_notification_get_fd [Function]

Arguments:

■ Returns the Unix file descriptor used by **notification-receive** for receiving notification objects. An Allegro CL multiple lightweight process application may use this file descriptor to wait for pending notifications. For architectures without :os-threads on the feature list, calling **notification-receive** will lock out all other lightweight processes until a notification is received. An Allegro CL multiple lightweight process may use **mp:wait-for-input-available** to wait for a notification to arrive without locking out other lightweight processes. When input arrives, use **notification-receive** to receive the notification object.

database-of [Method]

Arguments: *(object notification)*

■ Returns the database object associated with the notification *object*. Notification objects are returned from **notification-receive**. This function may be called outside of an AllegroStore transaction and in any ACL lightweight process.

notification-kind [Method]

Arguments: *(object notification)*

■ Returns the integer kind value associated with the notification *object*. Notification objects are returned from **notification-receive**. This function may be called outside of an AllegroStore transaction and in any ACL lightweight process. The kind value is specified by the notifying client as a notify argument.

notification-object [Method]

Arguments: *(object notification)*

■ Returns the persistent instance associated with the notification *object*. Notification objects are returned from **notification-receive**. This function must be called inside an AllegroStore transaction.

notification-queue-status [Function]

Arguments:

■ Returns three integers - the first value is the number of pending notifications, the second value is the number of dropped notifications resulting from queue overflow, and the third value is the queue size. The default queue size is 50; you can change the default by setting the OS_NOTIFICATION_QUEUE_SIZE environmental vari-

able before starting your application. This function may be called outside an AllegroStore transaction and may be called in any Allegro CL lightweight process. Calling this function before opening a database will result in an incorrect queue size return value.

notification-receive [Function]

Arguments: *timeout*

■ Waits to receive a notification object. This function may be called outside of an AllegroStore transaction and in any Allegro CL lightweight process. A positive *timeout* value specifies the number of milliseconds to wait for a notification object to arrive. A -1 *timeout* value specifies an infinite timeout - the function will not return until a notification object is received. The function returns the notification object when a notification object is received, or nil when a timeout occurs before an object is received.

notification-string [Method]

Arguments: *(object notification)*

■ Returns the string value associated with the notification object. Notification objects are returned from **notification-receive**. This function may be called outside of an AllegroStore transaction and in any ACL lightweight process. The string value is specified by the notifying client as a notify argument.

notification-subscribe [Method]

Arguments: *(obj persistent-standard-object)*

■ Informs the AllegroStore server that notifications concerning the specified instance should be sent to the subscribing client. Since this method calls preserve-pointer on the instance argument, the instance's database must be opened for read/write activities and the subscription must occur during a read/write transaction. You should not call **unpreserve-pointer** on the instance while the subscription is effect on any client. Note also that closing the relevant database on any client will result in an **unpreserve-pointer** call on the instance in question.

notification-unsubscribe [Method]

Arguments: *(obj persistent-standard-object)*

■ Informs the AllegroStore server that notifications concerning the specified instance should no longer be sent to the subscribing client.

notify [Method]

Arguments: *(obj persistent-standard-object) kind string &key :on-commit*

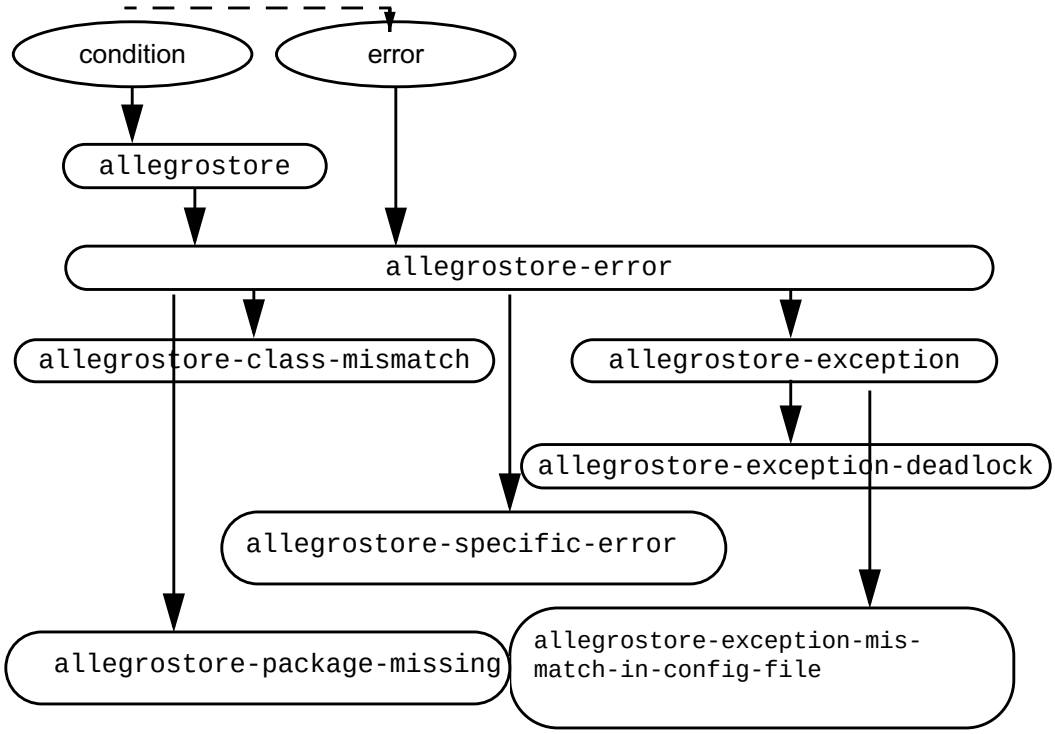
■ Directs the AllegroStore server to send notifications to all clients that have subscribed for notifications regarding the specified instance. The kind argument specifies an integer value that can be used for a custom client-to-client protocol. The string argument specifies a string value that can be used for a client-to-client protocol. If :on-commit is not nil, the notification will be queued for sending until the current transaction successfully commits; otherwise the notification is sent immediately. The default :on-commit value is nil.

7.2.15 Conditions

A *condition* is an object that holds the description of a situation that the program wishes to express. The situation can be a warning (e.g., disk space is running low) or an error (e.g., too few arguments supplied to a function).

The condition hierarchy

Each condition is a CLOS object and thus has a class. The classes are arranged in a hierarchy. AllegroStore extends the built-in condition hierarchy in the following way: (The dotted line indicates that `error` is a subclass of `condition` but that is not important for this discussion.)



`allegrostore` **[Condition]**

■ The `allegrostore` condition is never signaled in the current version, but will be used in the future to express AllegroStore non-error situations.

`allegrostore-error` **[Condition]**

■ The `allegrostore-error` condition is signaled to express a situation that must be resolved before execution can continue. If the situation isn't resolved programmatically (via **`handler-bind`** or a related function) then Lisp will enter the interactive debugger.

The `allegrostore-error` condition (and any condition which is a subclass of it) contains these slots:

Slot	Contents
`astore::format-control`	A format string used to describe the error situation.
`astore::format-arguments`	A list of the arguments for the **`format-control`** string.

`allegrostore-class-mismatch` [Condition]

■ The `allegrostore-class-mismatch` condition is signaled when:

- A database was opened with the `:use` keyword unspecified or specified as `:ask` (which is the default); and
- the definition of a class in the database differs from the definition currently in Lisp's memory.

If there is no handler for this condition, then the process will enter Lisp's interactive debugger. Four additional restarts may be invoked whenever this condition is signaled:

`allegrostore::use-db` **[Restart]**

Use the definition of the class in the database.

`allegrostore::use-db-all` **[Restart]**

Use the definition of the class in the database for this class and all subsequent class mismatches found in the opening of this database.

`allegrostore::use-memory` **[Restart]**

Use the definition of the class in memory.

`allegrostore::use-memory-all` **[Restart]**

Use the definition of the class in memory for this class and all subsequent class mismatches found in the opening of this database.

The `allegrostore-class-mismatch` condition has three additional slots:

Slot	Contents
`allegrostore::database`	The database being opened.
`allegrostore::disk-dbclass`	A `dbclass` object (see below) describing the definition of the `class` on the disk.
`allegrostore::memory-dbclass`	A `dbclass` object describing the definition of the `class` in memory.

A *dbclass object* is a **defstruct** structure with the following fields (plus others for internal use only).

Field	Contents
name	A symbol naming the class.
class	A class object in memory.
direct-superclasses	A list of direct superclass names.
direct-slots	A list of simple slot descriptions. Each simple slot description is a list of keyword values pairs where the keywords are those used to define a class in a **defclass** form. The order of the keywords and values is always the same, so two simple slot descriptions can be tested for equality with equal.
metaclass	The name of the metaclass.
finalized	True if the class has been finalized (i.e., the class precedence list and the list of slots have been computed).

The slots in the **defclass** objects can be accessed with functions named in the normal **defstruct** manner, e.g., **allegrostore::dbclass-name**.

A handler for the allegrostore-class-mismatch can analyze the differences between the two class definitions and uses **invoke-restart** to choose one of the two definitions but it should not modify whichever definition it chooses in any way.

allegrostore-specific-error [Condition]

■ This condition is signaled when AllegroStore has trouble starting up. It has one additional slot that identifies the specific error:

Slot	Contents
astore::code	A keyword denoting the specific problem (see below).

The following table shows the possible keywords at the time this manual was printed. The table also shows the correct interpretation and the additional information (if any) available in the astore::format-args slot (inherited from allegrostore-error).

Keyword	Meaning	Info in astore::format-args slot
:alloc-no-db	An attempt was made to create a persistent instance when there is no active database.	[none]

Keyword	Meaning	Info in astore::format-args slot
`:alloc-no-trans`	An attempt was made to create a persistent instance when there is no active transaction.	[none]
`:class-no-db`	An attempt was made to set a class slot in a persistent class when there is no active open database.	[none]
`:class-no-trans`	An attempt was made to set a class slot in a persistent class when there is no active transaction.	[none]
`:db-already-exists`	The open-database :if-does-not-exist :create keywords were used (meaning create a new database), and the database already exists.	The astore::format-args contains the database name.
`:db-cannot-supersede`	The open-database :if-exists :supersede keywords were used, and the specified database is currently already open. An additional restart named astore::close-existing-db is available that will close the existing database object and then continue with the open-database creation overwrite.	The astore::format-args contains the database filename and the currently open database object.
`:db-does-not-exist`	The open-database :if-does-not-exist :error keywords were used, and the database does not exist.	The astore::format-args contains the database name.
`:missing-config`	The AllegroStore configuration file cannot be found. To fix this, set the AS_CONFIG_PATH environmental variable to the Allegro Common Lisp installation directory.	The astore::format-args contains the configuration file name and the directories that have been searched

Keyword	Meaning	Info in astore::format-args slot
:missing-rootdir	OS_ROOTDIR environment variable is not set. This variable must be set to the ObjectStore installation directory.	[none]
:no-transaction	A preserved pointer is referenced when there is no active transaction.	The astore::format-args contains the referenced object.
:non-unique-value	The attempted database insertion would violate the persistent slot's :unique :t attribute.	The astore::format-args contains the slot name, the instance, and the data value.
:object-deleted	A preserved pointer was referenced that points to an object that is now deleted.	The astore::format-args contains the stale object.
:old-object	A preserved pointer was referenced after the database in which it resides has been closed, thereby making the reference stale.	The astore::format-args contains the stale object.

allegrostore-exception [Condition]

■ This condition is signaled whenever ObjectStore signals an exception. It has three additional slots:

Slot	Contents
astore::message	A string describing the exception.
astore::value	A value passed along with the string describing the exception.
astore::id	A symbol naming the ObjectStore class of the exception.

allegrostore-exception-deadlock [Condition]

■ This condition is signaled when ObjectStore recognizes that a deadlock has occurred between two or more processes accessing a single database. A deadlock is a situation where no individual process can proceed because each process needs a database lock that another one currently holds. To make any progress, one of the processes must free its locks by rolling back its transaction.

ObjectStore selects one of the deadlocked processes to receive the allegrostore-exception-deadlock condition. The selected process

should immediately end the current transaction by performing a roll back. After the selected process does the roll back, it can restart the transaction.

AllegroStore's **`with-transaction`** macro automatically handles this condition in the correct manner. It sets up a condition handler so that the transaction is rolled back and restarted when `allegrostore-exception-deadlock` is signaled.

Users may wish to add their own handlers for `allegrostore-exception-deadlock` when there are situations where the body of a transaction should not be executed more than once.

See the section 6.5 **Transactions** for more on deadlocks.

`allegrostore-exception-mismatch-in-config-file` [Condition]

■ This condition is signaled when you attempt to open a database that has a different AllegroStore configuration than the current AllegroStore configuration

`allegrostore-package-missing` [Condition]

■ This condition is signaled when AllegroStore can't read a symbol from the database because the package doesn't exist in Lisp. Its superclass is `allegrostore-error` and its slots are `database`, `package-name`, and `symbol-name`.

`allegrostore-specific-error-code` [Generic function]

Arguments: *`allegrostore-specific-error-condition`*

■ accesses the code slot from an `allegrostore-specific-error` condition.

`allegrostore-exception-id` [Generic function]

Arguments: *`allegrostore-exception-condition`*

■ accesses the id slot from an `allegrostore-exception` condition.

[This page intentionally left blank.]

Chapter 8 Database maintenance & administration

8.1 Using the ObjectStore documentation

This chapter and chapters 9 and 10 are reproduced from the ObjectStore manuals with minor modifications: **Database maintenance & administration**, **Administration utilities**, and **User utilities**.

This section of the manual is included as an extended reference. Not all of the existing ObjectStore documentation is provided. Ignore spurious references to other ObjectStore documents.

ObjectStore manuals are written for C++ users; the C++ notation you see here does not apply to AllegroStore. If a Lisp equivalent exists, we have already introduced it in the AllegroStore manual; these chapters are provided so that you may read the relevant source documentation.

8.2 In this chapter

This chapter describes the creation and maintenance of ObjectStore databases, and the ObjectStore daemon processes that must be running before any ObjectStore application (such as AllegroStore) can run: the ObjectStore *Server* and the ObjectStore *Cache Manager*. In addition, this chapter describes the ObjectStore *Directory Manager,* a daemon process that can be used to manage a hierarchy of Directory Manager (rawfs) databases.

The main topics are:

1. **File databases**, the default storage mechanism for ObjectStore. Creating and naming them.
2. **The Server**, which manages storage for ObjectStore databases. Server parameters are described in detail. Password and license management is also described, since it is controlled by the Server.
3. **The Cache Manager**, which handles the caching of data objects for ObjectStore clients. Cache manager parameters are described in detail.
4. **The Client Environment**. How to use environment variables to modify the characteristics of the Client.
5. **Directory Manager databases**. The storage model, Directory Manager parameters, and how to start the Directory Manager.
6. **The Ports file**, and how to change the port settings for network services.
7. **Error reporting by ObjectStore daemons**, and how it works.
8. **On-line backup and restore of ObjectStore databases**. A general description of their features. More information on these can be found in chapter 9 **Administration utilities**.

8.3 File databases

Creating databases

When an application allocates an object in persistent storage, it specifies the database to contain that storage. Databases are created by applications, with a call to one of the member functions **`with-database`**, **`open-database`**. The database is named by a pathname argument to the function that creates the database.

By default, ObjectStore databases are stored as *file databases*—that is, operating system files that contain databases. In most cases, you can manipulate these databases with standard operating systems commands as well as the ObjectStore commands listed in the following chapters: **Database maintenance and administration**, **Administration utilities**, and **User utilities**.

You specify a file database with an operating system pathname. For example:

```
with-database(db "/usr3/fauntleroy/my_file_db")
```

You can also specify a relative pathname.

Links are treated normally, and automounter pathnames are acceptable. If you are using `automount`, note that the pathname you specify should not include the automounter prefix -- usually `/tmp_mnt`. Referring to the file with this prefix does not cause the automounter to keep it mounted and could cause the file to appear deleted.

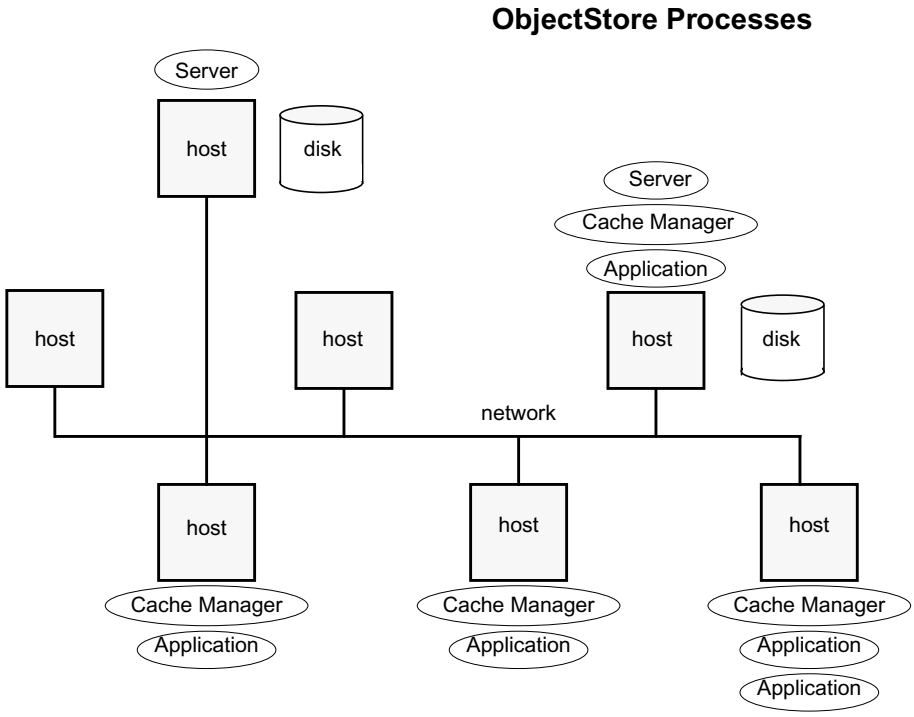
File databases must be stored on a host that is running a Server (see **The server**). ObjectStore determines which Server to use based on the NFS mountings of the client that creates the database.

Colons in file pathnames are interpreted as alphabetic characters if a slash character precedes the colon in the pathname. For example:

`/usr1/moe/a:b`	A file named `a:b` in the `/usr1/moe` directory, in the local host's namespace.
`bill:/usr2/dbs:abc`	A file named `dbs:abc` in the `/usr2` directory, on Server `bill`, in the Server's namespace.
`fifi/mimi:lulu`	A file named `mimi:lulu` in the `fifi` directory, relative to the working directory, in the local host's namespace.

8.4 The server

The Server is a daemon process that mediates all access to ObjectStore databases, including the storage and retrieval of persistent data, and controls ObjectStore license management. A Server must be running before any AllegroStore application can access the data on the host. The ObjectStore Server and Cache Manager (see **The cache manager**) manage ObjectStore databases, as shown in the following diagram.



If you are using Directory Manager databases, the Directory Manager participates in this process. See **Directory manager databases**.

The Server is usually started at system boot; regardless of when it is started, you can configure it with various command line options and parameter file options that are read at start-up. The pathname of the Server executable is `$OS_ROOTDIR/lib/osserver`. (See the installation instructions for information on setting `OS_ROOTDIR`.)

Starting the server

The command line to start a Server on a particular host is:

```
$OS_ROOTDIR/lib/osserver options
```

The Server is normally started by the script in `$OS_ROOTDIR/etc/start_servers`.

8.4.1 Server command line options

Ordinarily, you use Server parameters (see below) to control the Server's behavior. However, you can also give command line options to **`osserver`**.

The following command line options are recognized by the Server:

Server command line option	Function
`-c`	Forces all data to be propagated from the log to the database. The Server is not started up following the checkpoint.
`-F`	Foreground. This reverses the normal behavior, where the Server is forked off as a background process.
`-p` *`pathname`*	Specifies a parameter file. *`pathname`* names a parameter file to use during the start-up process. This option is generally not required.
`-v`	Shows Server parameter values at start-up.

After starting up, the Server displays the message: `Server started` on the system (`host`) console to let you know that it is ready to accept requests from clients on the network.

When run without `-F`, **`osserver`** returns `0` from a successful start-up of the background daemon, otherwise it returns `1`.

8.4.2 Server access control

ObjectStore has a client/server architecture. When an application reads or writes a database, it sends messages to an ObjectStore Server, which in turn reads and writes the data in the file that contains the database.

Because each ObjectStore Server handles requests from many different users, it is responsible for enforcing access control to files containing databases. It must use privileged access to read or write any user's database, and it must ensure that only users entitled by the host system's rules for access control are allowed access to the databases. To implement access control, the Server must know the access identity (on Unix, the user names and groups) of the client process that is requesting that it operate on a database.

Server access control is configured via the `Authentication Required` parameter. The mechanism associated with each of the parameter's values is described in **Server parameters**.

8.5 Authentication

If you start an application that asks the Server to read or write a protected file, the Server determines whether you have proper read/write permission for the file. This access checking has two parts. The first part is authentication, the operation in which the Server learns who the client is, and on behalf of which user it is performing operations. The second part is checking whether that user has permission to perform the requested operation.

The type of authentication ObjectStore uses is determined by the value of the Server parameter `Authentication Required`. (See **Server parameters**.) Its possible values are `NONE`, `SYS`, `DES` (Sun only), and `Unix Login`.

Note that certain Server operations that deal with file databases require authentication. If the client does not provide authentication, the Server will refuse to perform the operation. These include looking up, creating, and deleting file databases, or asking their size or access modes.

Administrative commands for which authorization is required send authentication to the Server first. ObjectStore applications send authentication to the Server the first time the application does an operation on that Server for which authentication is required. (That is, once an application has sent authentication information to a Server, it need not do it again, for the lifetime of the process.)

`err_authentication_failure` is signaled when authentication is done on a Server requiring `Unix Login` and the Server detects something wrong, generally that there is no user by the specified name, or the password is incorrect.

If the Server gets a command that needs authentication, but the connection has not been authenticated, and `Authentication Required` is not `None`, the Server rejects the command, and ObjectStore signals a condition ("Client credential too weak"). In practice, this should never happen, because ObjectStore always sends authentication before sending any such command, when authentication is required by the Server.

The Server performs several administrative operations only on behalf of an "authorized" client. These are the Server operations invoked by **`ossvrshtd`** and **`ossvrclntkill`**, with these exceptions:

- **`ossvrshtd`** is also allowed by the user who owns the Server process; with file databases, this is the `root`, so the exception doesn't matter.
- **`ossvrclntkill`** is allowed if the client thread to be killed is owned by the user issuing the command.

8.5.1 User interface to authentication

By default, the interface to `Unix Login` authentication communicates with the user interactively using the Unix `/dev/tty device` and the **`stdin`** and **`stdout`** streams, prompting for a user name and a password.

Your application can control this interface with the functions **`objectstore::set_simple_auth_ui()`** and **`get_simple_auth_ui()`**, which are described in the ObjectStore Reference Manual. This can be useful for applications that incorporate a graphical interface.

8.6 Server parameters file

All Server parameters have default values. We recommend that you use the default value for each parameter, which requires no action on your part. You also have the option of resetting these values, as described below.

Parameter files consist of a set of lines, each containing a parameter and a value. Parameters have textual names (such as `Host Access List`), and are followed by a colon, some whitespace (tabs or spaces), and a value (either numeric or text, depending on the parameter). Comment lines must contain a "#" as the first non-whitespace character. Case is insignificant for parameter names. Embedded spaces, however, are significant. If the same parameter name appears twice in the parameters file, only the first occurrence is read.

A simple parameters file might contain the following line:

```
Log File:/elvis1/elvis_server_logfile
```

You can specify the Server parameters file with the `-p` option to **`osserver`** or with one of the default locations. ObjectStore uses the following search order to find information on parameter settings:

Search order

1. The file specified by the `-p` command line option to **`osserver`**.
2. `$OS_ROOTDIR/etc/`host`_server_parameters` where host is the name of the current host as returned by the `hostname` program. `OS_ROOTDIR` is an environment variable.
3. `$OS_ROOTDIR/etc/server_parameters`.

`OS_ROOTDIR` is an environment variable set to the top-level directory in the part of the source hierarchy containing ObjectStore files.

If no parameters file is found, a message is displayed.

8.6.1 Parameter terms

Many of the Server parameters use the following terms:

Log. ObjectStore implements a *transaction model*. This means that your transactions are treated atomically—if a transaction modifies databases, either all or none of the modifications are made, but never some of them, even if the Server machine crashes during the transaction.

Log

To accomplish this, modifications sent to the database server are stored in a `log` database maintained by the Server. If your transaction aborts, the log entries are discarded. When your transaction commits, your program waits until the log information is safely on disk.

The log database consists of two *log segments* and a *data segment*. Both log and data segments are used to store data.

Log segment. Log records, such as commit records, are written to a *log segment.* When a log segment fills up, logging switches to the other log segment.

Log buffer. The *log buffer* is used to form log records across sectors. Its size defines the maximum size that you can write to a log record segment in one write operation.

When choosing the log record buffer size, you should consider balancing the cost of formatting data into a log record versus the cost of a separate write operation.

Data segment

Data segment. Data returned as part of a commit can be written as part of the commit record, but this occurs only if the total data being committed is small enough to fit in the log record buffer. If it is not, the data is written to the log *data segment*.

Data propagation

Propagation. The Server moves committed data from the log to databases through *propagation*. The information accumulated in the log is propagated from the log to the "material" (that is, real) database transparently in a way that does not interfere with client performance.

Sector

Sector. A *sector* is a 512-byte disk block.

8.6.2 Server parameters

This section describes each Server parameter. Parameters are presented in alphabetical order.

`Allow shared communications`

Controls whether the Server allows shared memory communications between itself and the client when they are on the same host, to improve performance. It is a Boolean that defaults to `yes` (meaning shared memory communications are enabled). If these communications are enabled, the Server and client exchange data via shared memory.

`Authentication required`

Specifies how the Server controls database access. The default is `SYS`. There are four modes:

1. `NONE`—No authentication is required. The Server will refuse access to file databases in this mode. That is, if no partition is supplied, the Server will not start.
2. `SYS`—ObjectStore uses the Sun ONC RPC `AUTH_SYS` authentication method, previously called `AUTH_Unix`. The client sends the Server a Unix user ID and a set of group IDs, which the Server trusts. The ObjectStore client library will always send the current effective user id and group set. This is the same mechanism used by the NFS protocol.

   Warning: If you use this method, be aware of the following two points:

   - If nontrustworthy users have access to the `root` password, they can assume the identity of any user.
   - A malicious user might contrive to patch the ObjectStore client so that it sends some access identity information other than the current effective user ID and group set.

   Note that not all systems can generate this information in a client. If you run an ObjectStore Server on such a system, this method is not available.

3. `DES`—Sun, AIX, and System V.4.0 only. ObjectStore uses the Sun ONC RPC `AUTH_DES` authentication, also known as "secure RPC". To enable this mechanism, you must first set up the Network Information System, run the **`keyserv`** daemon on all client and Server hosts, and register public keys in the `publickey` NIS map. (See the **`newkey`**, **`chkey`**, **`keyserv`**, and **`keylogin`** manual pages.)

   For a full explanation of the secure RPC mechanism, see the documentation supplied with your system.

   DES authentication protects against abuse of the `root` password and against straightforward attempts to patch an ObjectStore client to send a fraudulent access id. However, it is not secure against a determined intruder, due to bugs in both its design and in the reference implementation.

4. `Unix Login`—ObjectStore requires each client application to send a user name and password to the Server, which validates them. The advantage of this method is that there is no way to fool the Server—it grants exactly the access available to the user it authenticates. The disadvantage is that ObjectStore cannot arrange a "trusted path." That is, there is no way that a user, prompted for a password, can be sure that the password will be sent to the ObjectStore Server and only to the ObjectStore Server. A malicious program author could solicit and then retain passwords.

`DB expiration time`

The number of seconds, expressed as an integer, to wait before garbage collecting the virtual memory structures maintained for each database that is opened.

A value of `0` means that the structures should be garbage collected as soon as all clients have closed that database.

The default is `300` seconds.

This parameter is used only for Directory Manager (rawfs) databases.

`Deadlock victim`

Selects one of five algorithms—`Current`, `Age`, `Oldest`, `Work`, or `Random`—used for victim selection in the event of a deadlock.

- `Current` (the default) specifies that the client that completes the "waits for" graph deadlock cycle is selected as the victim.
- `Age` specifies that the youngest client is selected as the victim.
- `Oldest` selects the oldest client as the victim.
- `Work` specifies that the client who has done the least amount of work (as measured by RPC calls to the server during the current transaction) is selected.
- `Random` specifies that a random client is selected as the victim.

`Direct to segment threshold`

Sets, in sectors, the threshold value that determines whether segments are first written to the log and then to the database, or directly to the database.

If less than this threshold is written past the current end of segment during a transaction, the data is written to the log before being written to the database. If the

number of segments written is greater than this threshold, the data is written directly to the database.

The default is `128` sectors.

Choosing a value depends on how big you want the log to be versus the cost of writing and flushing the data separate from the write and flush of the log record.

The following tests are applied in order; the first one that matches determines how the data is stored:

1. If the data is being written to a newly created segment, data are always written directly to the database segment.
2. If the following conditions are met, the data goes directly to the database (this formula means the decision to write data directly to a database is not changed by the size of individual write operations):
   - Data are being written past the current end of the database segment, and
   - The last sector number being written minus the current committed size of the segment is greater than the new `Direct to Segment Threshold`.
3. If neither of the previous conditions apply, data is put in the log until the commit is done.

`Host access list`

Specifies a pathname of a file. This file must contain a set of primary names of hosts, one per line. (If DNS is in use, they must be fully qualified domain names). The Server refuses connections from any host not on the list, or whose name cannot be determined.

This mechanism is only as secure as the available means for translating hosts addresses to host names. On some networks, such as NETBIOS, there may be no secure means.

This parameter is intended for use in environments where a machine is on a network with untrustworthy hosts, and DES authentication is unavailable or unworkable.

`Log data segment growth increment`

Specifies how many sectors to grow the log data segment when more room is needed.

The default is `2048` sectors.

`Log data segment initial size`

Sets the initial size of the log data segment in sectors.

The default is `2048` sectors.

`Log file`

The pathname of the log file. Note that this cannot be a raw partition. If no pathname is given, this information will go in the rawfs. If you don’t have a rawfs, you must specify a pathname for this parameter.

`Log record segment buffer size`

Defines how much buffer space is reserved for log records.

The default is `1024` sectors.

`Log record segment growth increment`

Specifies how many sectors to grow the log record segment when more room is needed.

The default is `512` sectors.

`Log record segment initial size`

Sets the initial combined size of the two log record segments in sectors.

The default is `1024` sectors.

`Max data propagation per propagate`

This is the maximum number of sectors propagated in one propagate operation. This limits the impact of propagation on the handling of client requests.

The default is `512` sectors.

When counting the amount of data being propagated, the effect of noncontiguous data is weighted by 64 sectors. This means that with `Max Data Propagation Size` set to its default of `512` sectors, at most eight noncontiguous writes can occur in a single propagation.

`Max data propagation threshold`

If more than this number of sectors is waiting to be propagated, the Server forces propagation to run faster.

The default is `8192` sectors.

`Message buffer size`

Sets the size of the buffer used for reads and writes.

The default is `512` sectors (256KB), with the value being rounded to a multiple of 64KB.

`Message buffers`

Sets the number of `Message Buffer Size` buffers. The number defines the maximum number of clients that can simultaneously do `get_block`s, `return_block`s, and `commit`s.

The default is `4`.

`Notification retry time`

The time between retries for Server-to-Server communication; used during two-phase commit recovery.

The default is `60` seconds.

`PartitionN`

This parameter controls Directory Manager database storage only.

A filename of a partition or file to be used in the ObjectStore file system. `PartitionN` is specified in the Server parameters file as follows:

```
PartitionN TYPE pathname {expandable | nonexpandable}
```

TYPE is mandatory, and can be either `PARTITION` or `Unix`. In general, unless you find administration of ObjectStore databases simpler, or are storing more data than can fit in one file (that is, one disk), you should use Unix file databases, which provide greater flexibility.

pathname must begin with a slash. If you have specified type `PARTITION`, it is the "raw" device name (for example, `/dev/rsd0d`) of a Unix disk partition to be used as the nth partition in the ObjectStore raw file system. If you have specified `Unix`, it is the absolute pathname of a Unix file that is to be used for the ObjectStore raw file system.

The keyword `expandable`, if present, indicates an expandable partition and nonexpandable indicates the partition cannot be expanded. By default, regular file partitions are expandable, and raw device partitions are not.

The raw device is also known as a "character special" file in Unix. Typically, each partition is accessible both by a raw device and by a "block special" device. The Server requires the use of the raw device.

The partitions

```
Partition0
Partition1
...
PartitionN
```

may appear in any order in the Server parameters file. However, no empty slots are allowed: all partitions `0,1,...N` must appear.

Both Unix files and raw partitions may be used in the raw file system.

You add a partition by creating a new entry for it. For example:

```
Partition0:  PARTITION /dev/rsd4x expandable
Partition1:  PARTITION /dev/rsd4y
```

To increase the size of one or more existing partitions, you first use `dd`(1) to copy the data from an existing small partition to a new larger one (any of the existing partition(s) may be expanded) or use database backup/restore. Check the sizes of the partitions with the Unix command **`dkinfo`**(8); the new partition must be no smaller than the partition it is replacing. Then substitute each new name for the corresponding old one in the Server parameters file.

When you use multiple Unix files in an ObjectStore raw file system, you control expansion as follows:

If exactly one Unix file is used, it will expand dynamically as needed, as long as there is room in the Unix partition containing the file. If multiple Unix files, or a mixture of files and partitions, are used for the raw file system, they are treated differently with regard to expansion. Raw partitions may be expanded only at Server start-up. Unix files are expanded dynamically, if allowed. You may allow a Unix file in the raw file system to expand by specifying it as `expandable`.

For example, suppose the raw file system contains one raw partition `/dev/rsd4x`, and three Unix files (`/usr1/one`, `/usr2/two`, and `/usr3/three`). If you want to permit only two of the Unix files to expand, your Server parameters file might read:

```
Partition0: Unix /usr1/one expandable
Partition1: Unix /usr2/two expandable
Partition2: Unix /usr3/three nonexpandable
Partition3: PARTITION /dev/rsd4x
```

Currently, there is no way to shrink the size of a raw file system by removing one of its partitions or Unix files.

Note: It is unnecessary to use more than one Unix file in the same Unix partition for your raw file system.

`Propagation buffer size`

Selects the amount of buffer space reserved for propagation. This space is used for reading data from the log for writing to target databases. If the buffer size is large enough, data written to the log can be kept in memory until propagation occurs, thus avoiding the need to read the log. The default is `8192` sectors.

`Propagation sleep time`

When there is data to be propagated, determines the number of seconds between propagates. The default is `60` seconds.

8.7 Password and license management

Earlier versions of ObjectStore supported password protection but ObjectStore 5.0 does not.

8.8 Cache manager

Client cache

A Cache Manager is an auxiliary process that runs on each machine that runs ObjectStore clients. It participates in the management of an application's *client cache*, a local holding area for data mapped or waiting to be mapped into virtual memory.

If additional ObjectStore applications are started on the same machine, the same Cache Manager handles the caches of these applications as well. Although the same machine can run several ObjectStore applications at once, only a single Cache Manager is ever running on a given machine.

The Cache Manager runs automatically when it is needed; no command to start it is necessary. It runs as `root` to enforce the security of caches.

8.8.1 Cache manager parameters

Search order

The Cache Manager parameters file is similar to the Server parameters file in terms of location and internal format. The following search order is used:

1. `$OS_ROOTDIR/etc/host_cache_manager_parameters`, where `host` is the name of the current host as returned by the `hostname` program. `OS_ROOTDIR` is an environment variable.
2. `$OS_ROOTDIR/etc/cache_manager_parameters`.

Parameters files consist of a set of lines, each containing a parameter and a value.

Parameters have textual names (such as `Cache Directory`), and are followed by a colon, some whitespace (tabs or spaces), and a value (either numeric or text, depending on the parameter). Comment lines must contain a "#" as the first non-whitespace character. Case is insignificant for parameter names. Embedded spaces, however, are significant. If the same parameter name appears twice in the parameters file, only the first occurrence is read.

The following options are recognized in the Cache Manager parameters file.

Cache Manager parameters option	Function
`cache directory`	The directory in which ObjectStore places the cache file. The default is `/tmp/ostore`.
`commseg directory`	The directory in which ObjectStore places the communications segment. The default is `/tmp/ostore`.
`hard allocation limit`	The upper bound, in bytes, on the amount of disk space that the Cache Manager will allocate for its cache files and commseg files. If you try to exceed this limit, you receive an error message similar to the following at ObjectStore initialization time (for example, when you call **`with-database`**, **`with-current-database`**, or **`open-database`**): Cache Manager hard limit (NNNNN) exceeded by request for MMMMM bytes of cache and/or commseg This parameter is provided to allow the site administrator to prevent ObjectStore files from using too much disk space.
`soft allocation limit`	The suggested upper bound, in bytes, on the amount of disk space that the Cache Manager will allocate for its cache files and commseg files. An application is allowed to exceed this limit in order to run to completion; when it finishes, the Cache Manager automatically deletes the cache and commseg files if the soft limit has been exceeded. Doing so frees disk space, but can make application start-up slower by forcing ObjectStore to create new cache and commseg files. This parameter is provided to allow the site administrator to prevent ObjectStore cache files from using too much disk space.

8.9 The client environment

You can use environment variables to modify the characteristics of the client environment.

8.9.1 Client environment variables

The following environment variables can be set for any ObjectStore client application. For information on settings for ports for network services, see the **Ports file** section.

`OS_CACHE_DIR`

Pathname of the directory used for the cache. By default, ObjectStore automatically places cache and commseg files in the `/tmp/ostore` directory. Specifying an alternate pathname can be useful if your `/tmp/ostore` directory is small.

ObjectStore places the cache file into the specified directory, and assigns a filename to avoid conflicts between multiple processes that are all running ObjectStore and using the same directories.

If `OS_CACHE_DIR` is not set, ObjectStore places the cache file in the directory given as the value of the `Cache Directory` parameter in the Cache Manager parameters file, if one exists. For more information on using parameters files, see **Cache manager parameters**.

`OS_CACHE_SIZE`

Size of client cache in bytes. The cache size defaults to `8` Mb.

`OS_COMMSEG_DIR`

Pathname of the directory used for the communication segment. By default, ObjectStore automatically places cache and commseg files in the `/tmp/ostore` directory. If the Unix file system containing `/tmp/ostore` is very small, it might be desirable to locate the communication segment elsewhere.

ObjectStore places the commseg file into the specified directory, and assigns a filename to avoid conflicts between multiple processes that are all running ObjectStore and using the same directories.

You can also assign the directory pathname by setting the value of the `Commseg Directory` parameter in the Cache Manager parameters file. For more information, see **Cache manager parameters**.

`OS_DEF_EXCEPT_ACTION`

Unhandled exceptions

Controls what happens if an unhandled exception is signaled. If set to “abort”, ObjectStore calls **`abort`**(3) (this generally results in creation of a core file—see the man page for **`abort`**).

If set to an integer greater than or equal to 1, ObjectStore calls `exit`(3) with the specified integer as argument (see the man page). If set to anything else, or if not set, ObjectStore calls **exit**(3) with an argument of *`1`*.

`OS_DIRMAN_HOST`

Used by applications to specify a Directory Manager host to contact. Pathname parsing changes when this variable is set.

OS_DISABLE_PRE2_QUERY_SYNTAX_SUPPORT

For use with the C++ Library Interface. When set to `yes`, causes the query translator to treat all uses of "`[`" and "`]`" found in query expression strings as array subscripting operations. Any setting of the environment variable is ignored if the application calls **`os_coll_query::set_disable_pre2_query_syntax_support()`**.

OS_ENABLE_PRE2_QUERY_SYNTAX_WARNINGS

For use with the C++ Library Interface. When set to a non-null string, enables warnings from the C++ Library Interface query translator about the use of the obsolete nested element selection query syntax ("`[`"and "`]`").

The string should name the file to which the warnings should be written. Any setting of the environment variable is ignored if the application calls **`os_coll_query::set_enable_pre2_query_syntax_warnings()`**.

OS_HANDLE_TRANS

If your program references an illegal address that was not an ObjectStore persistent address (for example, if you happen to dereference a null pointer), ObjectStore simply returns the signal to the operating system to handle. If you don't set up your own `SIGSEGV` handler before ObjectStore is initialized, the error message is `Segmentation violation: core dumped`.

When you set `OS_HANDLE_TRANS` to any value, ObjectStore signals a TIX exception (`err_null_pointer` or `err_deref_transient_pointer`). This causes dereferences to illegal non-ObjectStore addresses to signal a TIX exception and print out a message, and lets you get a stack trace.

OS_INC_SCHEMA_INSTALLATION

If set, new databases are created in incremental schema installation mode; if not set, new databases are created in batch schema installation mode. The effect of this environment variable can be overridden for a particular process by using **`objectstore::set_incremental_schema_installation(`**).

OS_INHIBIT_TIX_HANDLE

Specifies an error message substring for which exception handling is to be disabled.

Many end-user applications have omnibus error handlers to catch all errors being signaled, and present them in an easily readable format to the user. This sometimes makes debugging difficult, because the backtrace information has disappeared.

When you specify a substring to `OS_INHIBIT_TIX_HANDLE`, if the substring appears in the formatted error message, exception handling is disabled for the specific error. You can then generate an unhandled exception dump for analysis, or view the backtrace in a debugger.

OS_LOG_TIX_FORMAT

The name of a log file to record all exceptions signaled. This file logs all of the **`printf`** control strings signaled, regardless of whether the exception is handled.

This facility is especially useful for debugging two situations: recursive exceptions (common if you get exceptions during message processing), and bad **`printf`** strings.

`OS_PORT_FILE`

The name of a ports file for network services.

`OS_RESERVE_AS`

mmap system call

An optimization to ObjectStore on the Sun architecture increases performance, sometimes by a very significant factor. However, this optimization can cause trouble if your own program calls the `mmap` system call with zero as the first argument, or if your program calls some subroutine library that does so.

If your program does one of these things, you should disable the optimization, either by calling the entry point **`objectstore::set_reserve_as_mode(os_boolean new_mode)`**, or by setting the environment variable `OS_RESERVE_AS` to any value. (If you both call the entry point and set `OS_RESERVE_AS`, the entry point takes precedence.)

`OS_ROOTDIR`

The top-level directory in the part of the file system hierarchy containing ObjectStore files. Serves as the prefix of various directory names used in search paths.

8.10 Directory manager databases

ObjectStore 5.0 does not support this facility (supported in earlier versions).

8.11 Ports file

Normally, the default settings for ports for network services are sufficient. To modify the settings, you change entries in the file `$OS_ROOTDIR/etc/ports`. You can also set the variable `OS_PORT_FILE` to the name of a file you create.

Each line in the file changes the port for some service over some network. The syntax of a line is:

```
NET:SERVICE:VERSION:PORT
```

`NET` is one of the following network types:

`Unix`	Local connections via Unix domain sockets
`TCP/IP`	`TCP/IP` connections. `TCP/IP` has a nickname of `IP`.

`SERVICE` specifies one of the following network services:

- `cache manager client`

  How a client finds the Cache Manager. Only meaningful on `Unix`.

- `cache manager server`

  How a Server finds a Cache Manager. Only valid on `TCP/IP`.

- `server client`

  How a client finds the Server.

`VERSION` is `4` for all services.

`PORT` is a pathname of a socket file for `Unix`, and a `TCP/IP` port number for `TCP/IP`.

Example:

```
TCP/IP:server client:4:54432
```

If the file `$OS_ROOTDIR/etc/ports` is not present, the default settings are used.

8.12 Error reporting by ObjectStore daemons

When one of the ObjectStore daemon processes does output on `stdout` or `stderr`, ObjectStore routes the output to a corresponding file, as follows:

`/tmp/ostore/osc4_out`	Cache Manager
`/tmp/ostore/oss_out`	Server

If the file does not already exist, ObjectStore creates it; if the file already exists, it appends to the file.

Normally, the daemons do not do such output. However, under certain unusual error conditions, they might print an error message in this way. This information might be helpful in understanding and resolving any malfunction. When you report a problem to Franz that might involve one of these daemons, please see if such a file exists, and let us know the contents.

When the daemon process is not running, you can safely delete the corresponding file. Usually very little is ever printed to these files, so they are not likely to occupy much disk space.

8.13 On-line backup and restore of ObjectStore databases

The ObjectStore backup and restore facility transparently backs up running applications without affecting concurrency, and restores databases into a directory. This facility works with either Directory Manager or file databases, as long as the backup consists entirely of one type.

On-line backup provides efficient, transparent backup without affecting the concurrency of running applications. It achieves this by taking advantage of any operations already being performed by the Server on behalf of various client applications, thereby amortizing the cost of performing these operations on their own. It gives priority to databases that are already open at the time the backup starts, and within a database, to those sectors that are being actively used.

You cannot restore databases whose pathnames contain an embedded directory separator in them. This can occur only when moving databases across architectures. For instance, if you back up database `c:\OS/2.odb` on an OS/2 system, you cannot restore it on a Unix system, because the "/" in the database name is interpreted as a directory separator, causing database creation to fail.

ObjectStore provides facilities for archiving databases to secondary storage with the **`osbackup`** command, and for restoring databases from an archive with the **`osrestore`** command. These utilities provide protection from catastrophic data loss due to hardware failure, and can also be used to transport databases from one location to another.

8.13.1 On-line backup

On-line backup includes this set of features:

- **Inter-database transaction consistency.**

  Databases are consistent both internally and across the entire set of databases being backed up.

- **Concurrent read- and write-access by ObjectStore applications to databases in the backup set.**

  That is, on-line backup never obtains locks on data in the backup set, so standard ObjectStore transactions can proceed normally. This capability prevents lock conflicts between the backup process and ObjectStore application programs, providing faster access to data, and eliminating the possibility of deadlocks being caused by the backup process.

- **Support for full and incremental backup.**

  Backup levels 0-9 are supported, where level 0 produces a full backup of all databases in the backup set, and levels 1-9 produce incremental backups that save only those ObjectStore segments that have changed since the most recent backup at a lower level.

- **Support for incremental backup of file databases requires a non–backward-compatible change in database format, and therefore is not supported.**

That is, file database backup always produces a full backup, regardless of the incremental level requested.

- **Optimizations that take advantage of disk reads generated by concurrent access by ObjectStore applications to databases in the backup set.**

  The backup process normally operates as a background thread, reading contiguous regions of data from disk and writing them out to secondary storage. However, it also intercepts read requests from other threads for data which have not yet been backed up, and copies the data at this time. This reduces the cost of backing up this data from a disk seek and disk read to the cost of an in-memory copy operation.

  Since data must generally be read in order to be modified, data modified by ObjectStore applications will generally be backed up before the modifications occur.

- **Architecture-independent format of backup archives, allowing databases to be moved across architectures.**

  All databases in the backup set must reside on the same Server host.

8.13.2 On-line restore

The **`osrestore`** command restores databases from a backup archive file, either the entire backup set or selected databases within the set. While **`osrestore`** runs with the ObjectStore Server on-line, ObjectStore applications cannot access databases that are being restored until the entire restore process has completed. Restore has the side effect of defragmenting database storage within the filesystem.

See the **`osbackup`** and **`osrestore`** command descriptions in **Administration utilities**, for more information on using this facility.

Chapter 9 Administration utilities

9.1 Using the ObjectStore documentation

This is the second of three chapters reproduced from the ObjectStore manuals with minor modifications: The others are chapter 8 **Database maintenance & administration**, and chapter 10 **User utilities**.

This section of the manual is included as an extended reference. Not all of the existing ObjectStore documentation is provided. Ignore spurious references to other ObjectStore documents.

ObjectStore manuals are written for C++ users; the C++ notation you see here does not apply to AllegroStore. If a Lisp equivalent exists, we have already introduced it in the AllegroStore manual; these chapters are provided so that you may read the relevant source documentation.

9.2 In this chapter

This chapter describes ObjectStore utilities used for administering, monitoring, and tuning the performance of ObjectStore directories and databases. These commands are prefixed with **os**.

The following table summarizes these utilities. Each utility is discussed in greater detail later in this chapter.

Utility	Description
osbackup	Provides on-line backup of ObjectStore databases
oschhost	Changes the host of a rawfs link.
oscmrf	Deletes cache files and commseg files that are not in use
oscmstat	Provides status information on the Cache Manager process
osrestore	Restores databases backed up with **osbackup**
ossvrchkpt	Forces all data to be propagated from the log to the database
ossvrclntkill	Kills a client thread on a Server
ossvrmtr	Reports information about resource utilization for a Server process
ossvrping	Reports whether a Server is running on a specified host
ossvrshtd	Shuts down a Server immediately
ossvrstat	Displays statistics on all clients currently connected to a Server, as well as information about the Server not specific to a particular client

Many user-level utilities for manipulating databases are also available. See the chapter on **Database User Utilities**.

9.3 Specifying pathnames

You specify the *pathname* argument to the utilities in this chapter according to the following rules:

File databases

You can specify an operating system pathname in this way:

```
/usr1/julie/my_file_db
```

You prefix the pathname with the host machine of the desired rawfs, in the form:

```
wally::/design/parts/my_dirman_db
```

Note: ObjectStore rawfs pathnames are always shown in lowercase form, although uppercase or lowercase can be used.

Here, the rawfs host prefix has the form:

```
hostname::
```

where *hostname* names the machine running the rawfs that manages the ObjectStore directory hierarchy containing the database.

9.4 Rawfs pathname wildcard processing

ObjectStore utilities that deal with rawfs directories and files, except **oscp**, can perform wildcard processing similar to shell wildcards (*, ?, {}, and []). For example, to list all the databases starting with charlie in directory sax, you type:

```
osls oscar::/sax/charlie\*
```

You must quote the wildcard with quotation marks (“”) or a backslash (\) to keep the shell from misinterpreting the asterisk as a shell wildcard.

9.5 Using the OS_DIRMAN_HOST variable

OS_DIRMAN_HOST

The recommended ways of specifying file and rawfs database pathnames are described in the previous section. You can also set the value of the variable OS_DIRMAN_HOST, which is provided in this release for compatibility with earlier releases. If you set this variable, pathname parsing is changed, so that pathnames lacking a rawfs host prefix are interpreted as naming rawfs databases managed by the specified host.

For example, if you have set OS_DIRMAN_HOST to wally, the pathname /usr1/julie/my_db is interpreted as a rawfs database on host wally.

```
::/usr1/julie/my_db
```

Filename restrictions

Note that if OS_DIRMAN_HOST were not set, the above pathname would be interpreted as a file database.

9.6 osbackup

osbackup [*options*] -f *backup-image-file pathname ...*

Generates a transaction-consistent image of the specified set of databases. This utility works with rawfs and file databases.

The *-f backup-file* argument specifies the output file for the backup image. The *pathname* argument can be a directory or database name. One or more pathnames may be specified, but all databases must reside on the same server. For rawfs databases, if a pathname is a directory, all databases in that directory are backed up, and, if the `-r` option is given, the backup descends recursively to include all subdirectories. For file databases, names must be specified with the *pathname* argument or in an import file, specified with the `-I` option.

On-line backup does not affect the concurrency of running applications. It does this by taking advantage of any operations already being performed by the server on behalf of various client applications, thereby limiting the cost of performing these operations on their own. On-line backup gives priority to databases that are already open at the time the backup starts, and within a database, to those sectors that are being actively used.

When backup starts up, it arranges to have all committed data for these databases propagated to the material database. It then determines which segments require backup and builds a map that describes this data and sets itself up to intercept read and write requests to and from these sectors. Anytime the server reads a sector of interest to the backup process which hasn't already been backed up, it allows the read to proceed and makes a copy of the data at that time. Similarly, write requests are intercepted and delayed long enough to retrieve the transaction consistent data first. Otherwise, the backup process operates in the background, retrieving data as efficiently as possible.

Supported tape drives include quarter inch cartridge and 8mm cartridge drives. Standard formats and sizes are:

Format	Capacity
QIC-11	60 Mb
QIC-24	60 Mb
QIC-150	150 Mb
EXB-8200	2200 Mb
EXB-8500	5000 Mb

Options to **osbackup** are:

-h server-host

Only useful with rawfs databases, where a single directory can contain databases that reside on different servers. Databases that do not reside on the specified *server-host* are not be backed up.

-i incremental-record-file

Specifies the *incremental-record-file,* a file that contains information about which databases have been backed up, and when they were backed up. This information is used to determine which segments within a database have been modified since the last backup at a lower level, and backs up only modified segments.

Performing a backup at any level for which no previous information exists is equivalent to doing a level 0 backup for that database.

This option can be used to override the default name for the file used to record this information. On UNIX systems the default name is `$OS_ROOTDIR/etc/backup_record.UID`, where `UID` is the effective user id of the user performing the backup.

`-I import-file`

Reads a list of databases to backup from `import-file`, where `import-file` contains a list of either file database or rawfs database pathnames, one per line. Leading and trailing white space is ignored. If the import file name is specified as "-," `osbackup` reads from standard input. If the `-I` option is used, additional pathnames can still be given on the command line.

`-l level`

Specifies the level of the backup. Backup is incremental at the segment level, meaning that a segment is only backed up if it has been modified since the last backup at a lower level. A level 0 backup (the default) backs up all of the segments in all of the specified databases. Backup levels of 0 through 9 are supported.

`-r`

Recursively descends into any directories specified on the command line, adding all databases found to the list of databases to be backed up. By default, only databases in the specified directory are backed up.

`-s size`

Sets the size in megabytes of the volume being dumped to. This option is mainly for use when backing up to a tape device, since end of media cannot be reliably detected on some systems. The user will be prompted to insert a new tape after every "size" megabytes of data are written.

On SunOS, use of the `-s` option is not required since the end of tape is reliably signaled to the application without any loss of data.

The pathname of the executable is `$OS_ROOTDIR/admin/osbackup`.

9.7 oschhost

oschhost [-fR] *newhost pathname ...*

oschhost [-fR] *oldhost newhost*

This utility operates on rawfs databases only.

Changes the host of an entry in the rawfs database, either by pathname (recursive, specified by the `-R` option, and wildcard allowed), or globally in a rawfs database. On UNIX, you must be the super-user to use this command.

The `-f` (force) argument makes the command proceed despite errors.

You can use **oschhost** to update the rawfs database after you restore an entire file system from one Server to another.

The pathname of the executable is `$OS_ROOTDIR/bin/oschhost`.

9.8 oscmrf

oscmrf *hostname*

Tells the Cache Manager process running on *hostname* to delete all the cache files and commseg files in its free pool. The value of *hostname* defaults to the local host. The Cache Manager deletes only files that are not in use by any client, so it is always safe to run this command.

After **oscmrf** runs, if an additional client appears, the Cache Manager must create new cache and commseg files, which is somewhat slower.

The pathname of the executable is `$OS_ROOTDIR/bin/oscmrf`.

9.9 oscmshtd

oscmshtd *hostname*

Shuts down the Cache Manager process running on *hostname*. *hostname* defaults to the local host.

The pathname of the executable is `$OS_ROOTDIR/bin/oscmshtd`.

9.10 oscmstat

oscmstat *hostname*

Prints out status information about *hostname*'s Cache Manager process for debugging the storage system. (*hostname* defaults to the local host.) It prints out one line for every Server to which the Cache Manager is connected. For each Server, it prints a line giving the name of the Server host, the client process ID of the client being processed, or `0` if none is being processed, and a string saying what the thread is doing or what it most recently did. It also prints out the names of all files known to the Cache Manager.

This is useful if you are trying to determine if files are in active use by ObjectStore, or are ObjectStore files no longer in use that can be deleted (with **oscmrf**). The second word of an ObjectStore file name is always the name of the host that created and owns or owned the file, so, for files named `ObjectStore_doolittle_commseg_8` and `ObjectStore_doolittle_cache_3`, the host name is `doolittle`. The command **oscmstat doolittle** prints out a list of all the files that the Cache Manager daemon on host `doolittle` currently knows about. If your file is *not* on the list, it is no longer in use, and can be removed with **oscmrf**.

Note that if **oscmstat** reports there is no Cache Manager running, it is also safe to delete the file, as long as you are certain that **oscmstat** did not fail due to temporary network failure or something similar.

Output typically looks like the following:

```
 AllegroStore 3.0 Cache Manager, Version 4.0.
 Process ID 22758, started Wed Nov 3 11:28:47 1993
 Soft Allocation Limit 0, Hard Allocation Limit 120000000.
 Allocated: free 32907264, used 8536064.
 Server host:      Client process   Status for this host:
                   ID:
 doolittle         0                Initializing: constructor finished
 higgins           0                Initializing: constructor finished
 pickering         0                Initializing: constructor finished
 pickering         0                Initializing: constructor finished

There is 1 client currently running on this host:
1637 101 (null) v3.0 0xf7660000

Free cache files:
  /tmp/AllegroStore_doolittle_cache_3 (8388608)
  /tmp/AllegroStore_doolittle_cache_5 (23994368)

In-use cache files:
  /tmp/AllegroStore_doolittle_cache_1 (8388608)

Free commseg files:
  /tmp/AllegroStore_doolittle_commseg_4 (147456)
  /tmp/AllegroStore_doolittle_commseg_6 (376832)

In-use commseg files:
  /tmp/AllegroStore_doolittle_commseg_2 (147456)
```

`Version` in the top line is an internal version number that has nothing to do with ObjectStore release numbers.

`Process ID` is the operating system process ID of the Cache Manager process. The allocation limit parameters are as described in the parameter file. The `Allocated` line gives the total sizes of the used pool and the free pool.

A list of lines follows, one for each Server connection to the Cache Manager. This information can be useful in debugging.

The last section consists of a list of lines, one for each client (ObjectStore application process) currently running on this host. For each client it gives the operating system process ID and user ID, the "name" of the client (assuming the client has called **`objectStore::set_client_name()`**), an internal version number that also has nothing to do with ObjectStore release numbers, and a virtual address within the Cache Manager that is useful in debugging the Cache Manager.

The pathname of the executable is `$OS_ROOTDIR/bin/oscmstat.`

9.11 osrestore

osrestore [*options*] -f *backup-file* [*pathname ...*]

Restores databases to disk that have been backed up with **osbackup**. The file or database specified by *pathname* is restored into the directory specified using the -d option, or the current working directory if -d is not provided. You can restore individual databases or directories by specifying them on the command line with the appropriate list of options. The required -f *backup-file* argument specifies a file or tape device that contains a backup image from which to restore databases.

By default all databases in the backup image will be restored into the current working directory. The -d option can be used to change the directory in which to restore. If one or more pathnames are given on the command line, only the named databases or directories will be restored. Unless the -n option is in effect, a directory name refers to the databases it contains and recursively to its subdirectories and the databases they contain.

See the examples later in this discussion for details on how database pathnames are reconstructed.

To restore databases, you always begin with a full level 0 backup image; **osrestore** prompts for any additional incremental backup images you may want to apply after this. Not all of the incremental backups necessarily need to be applied. To determine which incremental backups to apply, list the backup levels in chronological order, starting with the level 0 backup. For instance, if you made a level 0 backup on Monday, a level 5 on Tuesday and Wednesday, a level 3 on Thursday, and level 4 on Friday, the list would look like this: 0, 5, 5, 3, 4.

Now, scanning the list from right to left, find the lowest incremental backup level greater than 0, in this case the level 3 backup made on Thursday. All incrementally made between the level 0 backup and this level 3 backup need not be applied. In order to restore databases to their current state as of the backup on Friday you must apply the level 0 backup and the incremental backups made at level 3 and 4, in that order.

Options to **osrestore** are:

-d *restore-directory*

The directory in which to restore the databases. If not specified, the current working directory is used.

-n

Normally, if a directory is specified for restoring, all databases in the directory and its subdirectories are restored. Using the -n option limits the restore operation to databases contained in the named directory.

-t

Prints a list of databases contained in this backup image.

The following examples illustrate some uses of **osrestore**:

1. List the table of contents of the backup image in the file /test/backup-image

```
example> osrestore -t -f /test/backup-image
```

```
::example:/:     test/
::example:/test:     db1     db2     db3
```

This indicates that the backup image contains three file databases in the directory `/test` that were backed up on the host `example`.

2. Restore the entire backup image back into its original location

```
example> osrestore  -d /  -f /test/backup-image
restoring "::example:/test/db1" to "::example:/test/db1"
restoring "::example:/test/db2" to "::example:/test/db2"
restoring "::example:/test/db3" to "::example:/test/db3"

Do you wish to restore from any additional incremental
backups?
  Answer yes or no: no

restore completed for database "::example:/test/db1"
restore completed for database "::example:/test/db2"
restore completed for database "::example:/test/db3"
```

3. Restore the entire backup image into the directory `/examples`

```
example> osrestore -d /examples -f /test/backup-image
restoring "::example:/test/db1" to
"::example:/examples/test/db1"
restoring "::example:/test/db2" to
"::example:/examples/test/db2"
restoring "::example:/test/db3" to
"::example:/examples/test/db3"

Do you wish to restore from any additional incremental
backups?
Answer yes or no: no

restore completed for database
"::example:/examples/test/db1"
restore completed for database
"::example:/examples/test/db2"
restore completed for database
"::example:/examples/test/db3"
```

4. Restore the entire contents of the directory `/test` into the directory `/examples`

```
example> osrestore -d /examples -f /test/backup-image
/test
restoring "::example:/test/db1" to
"::example:/examples/db1"
restoring "::example:/test/db2" to
"::example:/examples/db2"
restoring "::example:/test/db3" to
"::example:/examples/db3"
```

```
Do you wish to restore from any additional incremental
backups?
  Answer yes or no: no

restore completed for database "::example:/examples/db1"
restore completed for database "::example:/examples/db2"
restore completed for database "::example:/examples/db3"
```

5. Restore only the database `/test/db1` into the current directory (`/examples`)

```
example> osrestore -f /test/backup-image /test/db1
restoring "::example:/test/db1" to
"::example:/examples/db1"

Do you wish to restore from any additional incremental
backups?
Answer yes or no: no

restore completed for database "::example:/examples/db1"
```

The pathname of the executable is `$OS_ROOTDIR/bin/osrestore`.

9.12 ossvrchkpt

ossvrchkpt *hostname*

Forces all data to be propagated from the log to the database.

The pathname of the executable is `$OS_ROOTDIR/bin/ossvrchkpt`.

9.13 ossvrclntkill

ossvrclntkill *hostname client-hostname client-pid*

ossvrclntkill *hostname [* -p *client-pid] [* -n *name]*
[-h *host] [* -a *]*

The first form kills a client thread on the Server running on *hostname*. You use **ossvrstat** to determine the *client-hostname* and *client-pid*.

In the second form, -p *client-pid* specifies the process id of the client process as an unsigned number.

Options are:

-p *client-pid*

In the second form, specifies the process id of the client process as an unsigned number.

-n *name*

Specifies the name (as set by objectstore::set_client_name()) of the client process to kill.

-h *host*

Specifies the client host name of the client process to kill.

-a

Specifies that all clients matching the specified criteria should be killed.

If you specify -a, you must specify one of -p, -n, or -h. To achieve the effect of -a by itself, use **ossvrshtd**.

If you do not specify -a, one of -p, -n, or -h must uniquely identify a client process on the Server running on *hostname*.

If the Server is run as root with authentication set to something other than NONE (authentication is SYS by default), the following rules apply.

1. Any user can kill clients they own. If the -a option is used (kill all clients matching the given search pattern), the user must own *all* matching processes, otherwise authentication fails and no clients are killed.
2. Any Release 1 client can be killed by **ossvrclntkill**, again with the exception of the rawfs.

Otherwise, no authentication is required.

The pathname of the executable is $OS_ROOTDIR/bin/ossvrclntkill.

9.19 ossvrmtr

ossvrmtr *hostname*

Provides information about resource utilization for the Server process running on the specified host, including system resource usage and internal metering. Metering information is summarized for total clients and for logs for these intervals: the last minute, the last 10 minutes, the last hour, and since startup.

You can use the **ossvrstat** command to see the Server parameters and per-client information.

For example, where `elvis` is a UNIX ObjectStore server:

```
paul@elvis% bin/ossvrmtr elvis

AllegroStore 3.0 Database Server
Client/Server protocol version 1.5
Compiled by paul at 93-11-05 18:53:58 in /elvis1/foo/bar/nserver
User time:                          1176.6 secs
System time:                        454.7 secs
Max. Res. Set Size:                 1874
Page Reclaims:                      127988
Page Faults:                        91648
Swaps:                              0
Block Input Operations:             46753
Block Output Operations:            7336
Signals Received:                   0
Voluntary Context Switches:         97645
Involuntary Context Switches:       222551

Log Sectors:                        36512

Server Meters:
Total since server start up:
      Client Meters:
      82591 messages received 13646 callback messages sent
      109104 callback sectors  103360 succeeded sectors
      260938 KB read        33972 KB written
      167 commits           9 readonly commits
      347 aborts            0 two phase transactions
      346 deadlocks         0 message buffer waits
      Log Meters:
      169 log records       6 record segment switches
      0 flush data          179 flush records
      0 KB data             38602 KB records
      22985 KB propagated   31 KB direct
      1479 propagations

Total over past 60 minute(s):
      Client Meters:
      51486 messages received 8547 callback messages sent
      68328 callback sectors 64768 succeeded sectors
      161156 KB read        21496 KB written
      105 commits           6 readonly commits
      206 aborts            0 two phase transactions
```

```
      206 deadlocks          0 message buffer waits
      Log Meters:
      105 log records        3 record segment switches
      0 flush data           113 flush records
      0 KB data              21953 KB records
      16840 KB propagated    0 KB direct
      1082 propagations

Total over past 10 minute(s):
      Client Meters:
      8482 messages received 1409 callback messages sent
      11264 callback sectors 10672 succeeded sectors
      26544 KB read          3916 KB written
      20 commits             2 readonly commits
      33 aborts              0 two phase transactions
      33 deadlocks           0 message buffer waits
      Log Meters:
      20 log records         0 record segment switches
      0 flush data           20 flush records
      0 KB data              3995 KB records
      2044 KB propagated     0 KB direct
      132 propagations

Total over past 1 minute(s):
      Client Meters:
      742 messages received  145 callback messages sent
      1160 callback sectors  1088 succeeded sectors
      2248 KB read           216 KB written
      1 commits              0 readonly commits
      3 aborts               0 two phase transactions
      3 deadlocks            0 message buffer waits
      Log Meters:
      1 log records          0 record segment switches
      0 flush data           1 flush records
       0 KB data             220 KB records
      192 KB propagated      0 KB direct
       11 propagations
```

The pathname of the executable is `$OS_ROOTDIR/bin/ossvrmtr`.

9.20 ossvrshtd

ossvrshtd [-f] *hostname*

Immediately shuts down the Server running on the specified host.

You must either be the super-user or be logged in as the `uid` of the owner of the **osserver** process to shut down the Server.

Without the `-f` (force) argument, it asks you to confirm that you want to shut down the Server.

On UNIX, if the Server is run as `root` with authentication set to something other than `NONE` (authorization is `SYS` by default), **ossvrshtd** must be run as `root`. Otherwise, **ossvrshtd** will succeed if it is run by the same user who started the Server, or by the `root` user.

The pathname of the executable is `$OS_ROOTDIR/bin/ossvrshtd`.

9.21 ossvrstat

ossvrstat *hostname*

Displays the settings of Server parameters, Server usage meters, and information for each client currently connected to the ObjectStore Server running on the specified host. For each client, Server resource information is given, followed by each client's state, grouped by state.

Each client is identified by *hostname*, and the program name is listed (if known) together with the process id on that host. (If the program name is not known, default_client_name is listed instead; the program name can be set with **ObjectStore::set_client_name()**.)

Sample output is shown in the following example:

```
elvis% ossvrstat elvis

AllegroStore 3.0 Database Server
Client/Server protocol version 1.5
Compiled by paul at 93-11-05 18:53:58 in /elvis/foo/bar/nserver

Parameter file:
Allow Shared Communications:           Yes
Authentication Required:               SYS
RAWFS DB Expiration Time:              300 seconds
Deadlock Strategy:                     Work
Direct To Segment Threshold:           128 sectors (64KB)
Log Path:                    /elvis/elvis_server_log
Log Data Segment Initial Size:         2048 sectors (1MB)
Log Data Segment Growth Increment:     2048 sectors (1MB)
Log Record Segment Buffer Size:        1024sectors (512KB)
Log Record Segment Initial Size:       1024 sectors (512KB)
Log Record Segment Growth Increment:   512 sectors (256KB)
Max Data Propagation Threshold:        8192 sectors (4MB)
Max Data Propagation Per Propagate:    512 sectors (256KB)
N Message Buffers:                     4
Notification Retry Time:               60 seconds
Propagation Sleep Time:                60 seconds
Propagation Buffer Size:               8192 sectors (4MB)
TCP Receive Buffer Size:               16384 bytes
TCP Send Buffer Size:                  16384 bytes
Log Sectors:                           36512

Server Meters:
Total since server start up:
      Client Meters:
```

```
87747 messages received     14450 callback messages sent
      115528 callback sectors 109480 succeeded sectors
      277146 KB read         35908 KB written
      177 commits            9 readonly commits
      371 aborts             0 two phase transactions
      370 deadlocks          0 message buffer waits
      Log Meters:
      179 log records        7 record segment switches
      1 flush data           189 flush records
      264 KB data            40310 KB records
      24521 KB propagated    31 KB direct
      1576 propagations

Total over past 60 minute(s):
      Client Meters:
      51145 messages received 8150 callback messages sent
      65152 callback sectors 61624 succeeded sectors
      160088 KB read         20836 KB written
      104 commits            5 readonly commits
      206 aborts             0 two phase transactions
       206 deadlocks         0 message buffer waits
      Log Meters:
      104 log records        3 record segment switches
      1 flush data           112 flush records
      264 KB data            21223 KB records
      15924 KB propagated    0 KB direct
      1024 propagations

Total over past 10 minute(s):
      Client Meters:
      8282 messages received 1382 callback messages sent
      11056 callback sectors 10504 succeeded sectors
      26012 KB read          3476 KB written
      18 commits             1 readonly commits
      33 aborts              0 two phase transactions
      33 deadlocks           0 message buffer waits
      Log Meters:
      18 log records         1 record segment switches
      1 flush data           18 flush records
      264 KB data            3279 KB records
      1792 KB propagated      0 KB direct
      115 propagations

Total over past 1 minute(s):
      Client Meters:
      923 messages received  57 callback messages sent
      456 callback sectors   416 succeeded sectors
      2888 KB read           352 KB written
       2 commits             0 readonly commits
      4 aborts               0 two phase transactions
       4 deadlocks           0 message buffer waits
      Log Meters:
      2 log records          1 record segment switches
       0 flush data          2 flush records
       0 KB data             360 KB records
       356 KB propagated     0 KB direct
```

```
       22 propagations

Client Meters: (9 active client(s))
client #159 (elvis/8462//ostore/sym/lib/osdir)
   9 messages received      9 callback messages sent
   0 callback sectors       0 succeeded sectors
   37 KB read               12 KB written
   2 commits                0 readonly commits
   0 aborts                 0 two phase transactions
   0 deadlocks              0 lock timeouts
client #163 (elvis/8470/swift $Revision: 1.1.1.2.1.1 $)
   12021 messages received  2588 callback messages sent
   20704 callback sectors   19704 succeeded sectors
   37717 KB read            5573 KB written
   27 commits               5 readonly commits
   42 aborts                0 two phase transactions
   42 deadlocks             0 lock timeouts
client #164 (elvis/8472/swift $Revision: 1.1.1.2.1.1 $)
   10461 messages received  859 callback messages sent
    6872 callback sectors   6528 succeeded sectors
   33088 KB read            4464 KB written
   22 commits               0 readonly commits
   47 aborts                0 two phase transactions
   47 deadlocks             0 lock timeouts
client #165 (elvis/8474/swift $Revision: 1.1.1.2.1.1 $)
   10968 messages received  2469 callback messages sent
   19752 callback sectors   18776 succeeded sectors
   34716 KB read            4184 KB written
    20 commits              0 readonly commits
   47 aborts                0 two phase transactions
   47 deadlocks             0 lock timeouts
client #166 (elvis/8484/swift $Revision: 1.1.1.2.1.1 $)
   10184 messages received  711 callback messages sent
   5688 callback sectors    5400 succeeded sectors
   32192 KB read            4268 KB written
   21 commits               0 readonly commits
   48 aborts                0 two phase transactions
   48 deadlocks             0 lock timeouts
client #167 (elvis/8482/swift $Revision: 1.1.1.2.1.1 $)
   10929 messages received  2583 callback messages sent
   20664 callback sectors   19344 succeeded sectors
   34568 KB read            4144 KB written
   20 commits               2 readonly commits
   49 aborts                0 two phase transactions
   49 deadlocks             0 lock timeouts
client #168 (elvis/8478/swift $Revision: 1.1.1.2.1.1 $)
   11366 messages received  2262 callback messages sent
   18096 callback sectors   17328 succeeded sectors
   35936 KB read            4584 KB written
   22 commits               1 readonly commits
   48 aborts                0 two phase transactions
   48 deadlocks             0 lock timeouts
client #169 (elvis/8480/swift $Revision: 1.1.1.2.1.1 $)
   11607 messages received  2662 callback messages sent
   21296 callback sectors    20096 succeeded sectors
   36672 KB read            4684 KB written
```

```
   23 commits               1 readonly commits
   45 aborts                0 two phase transactions
   45 deadlocks             0 lock timeouts
client #170 (elvis/8476/swift $Revision: 1.1.1.2.1.1 $)
   10012 messages received  307 callback messages sent
   2456 callback sectors    2304 succeeded sectors
   31676 KB read            3964 KB written
   19 commits               0 readonly commits
   44 aborts                0 two phase transactions
   44 deadlocks             0 lock timeouts

Client connections processing a client message:
      client #166 (elvis/8484/swift $Revision: 1.1.1.2.1.1. $) ->
get_blocks2 mapped

Client connections waiting for a lock:
      client #165 (elvis/8474/swift $Revision: 1.1.1.2.1.1 $) ->
      upgrade_locks2
      write locking database #16, segment #2, starting at sector
      16432 for 8 sectors
      client #167 (elvis/8482/swift $Revision: 1.1.1.2.1.1 $) ->
      upgrade_locks2
      write locking database #16, segment #2, starting at sector
      53184 for 8 sectors
      client #168 (elvis/8478/swift $Revision: 1.1.1.2.1.1 $) ->
      get_blocks2 mapped
      read locking database #16, segment #2, starting at sector
      36408 for 8 maximum sectors

Client connections awaiting a client message:
 client #159 (elvis/8462//odi/r3/ostore/sym/lib/osdir)
 client #163 (elvis/8470/swift $Revision: 1.1.1.2.1.1 $)
 client #164 (elvis/8472/swift $Revision: 1.1.1.2.1.1 $)
 client #169 (elvis/8480/swift $Revision: 1.1.1.2.1.1 $)
 client #170 (elvis/8476/swift $Revision: 1.1.1.2.1.1 $)
```

The pathname of the executable is `$OS_ROOTDIR/bin/ossvrstat`.

[This page intentionally left blank.]

Chapter 10 User utilities

10.1 Using the ObjectStore documentation

This is the last of three chapters reproduced from the ObjectStore manuals with minor modifications. The other two are chapter 8 **Database maintenance & administration** and chapter 9 **Administration utilities**.

This section of the manual is included as an extended reference. Not all of the existing ObjectStore documentation is provided. Ignore spurious references to other ObjectStore documents.

ObjectStore manuals are written for C++ users; the C++ notation you see here does not apply to AllegroStore. If a Lisp equivalent exists, we have already introduced it in the AllegroStore manual; these chapters are provided so that you may read the relevant source documentation.

10.2 In this chapter

This chapter describes user-level ObjectStore utilities used for creating and manipulating ObjectStore databases and the directories that contain them. (Utilities for administering databases are described in **Administration utilities**.)

The following table summarizes these utilities. Each utility is discussed in greater detail later in the chapter.

Utility	Description
oschangedbref	Changes database references
oschgrp	Changes the GID of directories or databases
oschmod	Changes permission modes of directories or databases
oschown	Changes the ownership of directories or databases
oscompact	Consolidates ObjectStore databases
oscp	Creates a copy of an ObjectStore database
osdf	Displays information on disk utilization for an ObjectStore file system
osglob	For ObjectStore files, performs filename expansion on the specified wordlist
oshostof	Tells you what host a database is on
osls	Lists the contents of an ObjectStore directory
osmkdir	Creates an ObjectStore directory
osmv	Renames an ObjectStore database or directory
osrm	Removes a database from its host Server and directory database
osrmdir	Removes an ObjectStore directory from the directory database
ossetasp	Finds or sets the application schema database for an executable
ossevol	Performs simple schema evolution
ossize	Reports the size of an ObjectStore database and its segments
ossvrping	Reports whether a Server is running on a specified host
ostest	Returns a value of true or false for a conditional test

Utility	Description
osverifydb	Verifies that pointers within a database are valid
osversion	Prints the version of ObjectStore you are using

Symbolic links to commands in /bin

By default, there are symbolic links to all commands located in `$OS_ROOTDIR/bin` through `/usr/bin`, and to commands located in `$OS_ROOTDIR/admin` through `/etc`, so that you need not add `OS_ROOTDIR` to your search path.

The user-level commands are prefixed with `os`, and most are analogous to shell commands; they include **oschgrp**, **oschmod**, **oschown**, **osls**, **osmkdir**, **osmv**, **osrm**, and **osrmdir**. In most cases, you can use ObjectStore utilities and their corresponding shell commands interchangeably on operating system files and directories containing ObjectStore file databases; any differences are noted in the documentation for a particular utility.

Where the input name is a file database name, the commands simply pass the name on to the corresponding system command (except **oscp**). Those commands that modify an individual database file in any way (**oschgrp**, **oschmod**, **oschown**, **osmv**, and **osrm**) attempt to verify that the file being operated on is in fact a database, by calling **os_database::lookup()** on the path before operating on it. This verification is only done, however, if *neither* `-f` (force) nor `-R` (recursive) is specified. If force or recursive flags are specified, the paths are simply passed on to the shell without further checking.

This means, for example, that

```
osrm /foo/bar
```

removes `/foo/bar` only if it is a database, while

```
osrm -f /foo/bar
```

removes `/foo/bar` regardless of what type of file it is.

10.3 oschangedbref

Changing database references

`oschangedbref` *`db`* `from to`

Changes the database references from a database. Both `from` and `to` are one of:

from or to argument option	Function
`name`	For the `from` argument, an absolute pathname that includes a server host prefix. For the `to` argument, a relative name.
`-n` *`name`*	For the `to` argument only, a relative name. This option must be used for names beginning with a hyphen.
`-i` *`db`*	The ID of the existing database *`db`*
`-I` *`id1 id2 id3`*	A specified ID (three unsigned decimal numbers)

The pathname of the executable is `$OS_ROOTDIR/bin/oschangedbref`.

10.4 oschgrp

oschgrp [-R][-f] *group pathname...*

Changing GID

Changes the group ID (GID) of the directories or databases given in *pathname...* to *group*. The group is a group name or number found in the GID file, /etc/group. You must belong to the specified group and be the owner of the database, or be the super-user.

-f and -R are identical to the shell **chgrp** command's force and recursive options, respectively.

oschgrp can perform wildcard processing similar to Unix shell wildcards (*, ?, {}, and []). You must quote the wildcard with quotation marks ("") or a backslash (\) to keep the shell from misinterpreting the asterisk as a shell wildcard.

oschgrp supports all the arguments of the shell command chgrp, and it accepts a combination of rawfs pathnames and file pathnames.

The pathname of the executable is $OS_ROOTDIR/bin/oschgrp.

10.5 oschmod

oschmod [-R][-f] *new-mode pathname...*

Changing permissions

Changes the permissions mode of the directories and databases given in *pathname* as arguments to *new-mode*. You must be the owner of the database or the super-user in order to change its mode. Execute permissions are relevant only to directories. The mode of each named file is changed according to *new-mode*, which may be absolute or symbolic, as follows.

Absolute Modes. An absolute mode is an octal number constructed from the OR of the following modes (note that *execute* is meaningful only for directories):

Octal number	Function
400	Read by owner.
200	Write by owner.
100	Execute (search in directory) by owner.
040	Read by group.
020	Write by group.
010	Execute (search) by group.
004	Read by others.
002	Write by others.
001	Execute (search) by others.

Symbolic Modes. A symbolic mode has the form:

```
[ who ] op permission [ op permission ] ...
```

who is a combination of:

who option	Type of permission it grants
u	User permissions
g	Group permissions
o	Others
a	All, or ugo

If *who* is omitted, the default is a, but the setting of the file creation mask (see **umask** in **sh**(1) or **csh**(1) for more information) is taken into account. When *who* is omitted, **oschmod** does not override the restrictions of your user mask.

op is one of:

op option	Function
+	To add the permission
-	To remove the permission

=	To assign the permission explicitly (all other bits for that category, owner, group, or others, are reset).

permission is any combination of:

permission option	Function
r	Read
w	Write
x	Execute

The letters u, g, or o indicate that *permission* is to be taken from the current mode for the user-class.

Omitting *permission* is useful only with =, to take away all permissions.

oschmod can perform wildcard processing similar to Unix shell wildcards (*, ?, {}, and []). You must quote the wildcard with quotation marks (“”) or a backslash (\) to keep the shell from misinterpreting the asterisk as a shell wildcard.

oschmod supports all the arguments of the shell command **chmod**. -f and -R are identical to the shell **chmod** command’s force and recursive options, respectively. Note that wildcards must be quoted with the backslash (\) character to get them past the shell, for example, *sax::/charlie**.

The pathname of the executable is $OS_ROOTDIR/bin/oschmod.

10.6 oschown

oschown [-R][-f] *owner[.group] pathname...*

Changing owner

Changes the ownership of the directories or databases given in *pathname* as arguments to *owner*. The *owner* is a user name found in the password file, /etc/passwd. Only the super-user may change the owner of a directory or database.

oschown can perform wildcard processing similar to Unix shell wildcards (*, ?, {}, and []). You must quote the wildcard with quotation marks (“”) or a backslash (\) to keep the shell from misinterpreting the asterisk as a shell wildcard.

oschown supports all the arguments of the shell command **chown**. -f and -R are identical to the shell **chown** command’s force and recursive options, respectively. It accepts a combination of rawfs pathnames and file pathnames.

oschown supports the arguments -f and -R, the force and recursive options, respectively.

The pathname of the executable is $OS_ROOTDIR/bin/oschown.

10.7 oscompact

```
oscompact  [-dbs_to_compact pathname+]
           [-segments_to_compact [pathname segment_number]+]
           [-db_references pathname+]
           [-segment_references [pathname segment_number]+]
           [-compaction_threshold percent_of_deleted_space]
```

The **oscompact** utility, running as an ObjectStore client process, compacts the databases whose pathnames are given after the `-dbs_to_compact` option flag, as well as those segments whose database pathname and segment number are given in the `-segments_to_compact` option. One of the `-dbs_to_compact` or `-segments_to_compact` arguments *must* be supplied; all other arguments are optional.

If supplied, the pathnames after the `-db_references` option name databases that are considered to contain pointers or ObjectStore references to the databases and/or segments to be compacted, as are the segments whose database pathname and segment number are given after the `-segment_references` option. The segment number used to identify the segment is the number obtained by a call to the API function **os_segment::get_number()**.

Finally, the `-compaction_threshold` option allows the caller to avoid the compaction of segments that have less than the `compaction_threshold` percentage of deleted space. If this option is not supplied, any segment that has internal deleted space will be compacted. The application programmer's interface for compaction is **objectstore::compact()**. See the *ObjectStore Reference Manual* for more information.

File systems

ObjectStoreupports two "file systems" for storing databases, and the compactor can run against segments in databases in either file system. In the first, and most common case, a single database is stored in a single host system file. The segments in such a database are made up of extents, all of which are allocated in the space provided by the host operating system for the single host file. When there are no free extents left in the host file, and growth of an ObjectStore segment is required, the ObjectStore Server extends the host file to provide the additional space. The compactor permits holes contained in segments to be compacted for return to the allocation pool for the host file, freeing that space for use by other segments in the same database. However, since operating systems provide no mechanism to free disk space allocated to regions internal to the host file, any such free space will remain inaccessible to other databases stored in other host files.

The ObjectStore raw file system, on the other hand, stores all databases in a single region, on either one or more host files or a "raw partition". When using the raw file system, any space freed by the compaction operation can be reused by any segment in any database stored in the raw file system.

The compactor compacts all C and C++ persistent data, including ObjectStore collections, indexes, and bound queries, and correctly relocates pointers and all forms of ObjectStore references to compacted data. ObjectStore `os_reference_local` references are relocated assuming they are relative to the database containing them. The compactor respects ObjectStore clusters, in that compaction ensures that objects allocated

in a particular cluster remain in the cluster, although the cluster itself may move as a result of compaction.

Limitations

The following data restrictions must be observed in using the compactor:

- *Unions requiring user discriminant functions*: Union discriminant functions require access to the representation to be compacted in order to run and therefore cannot be compacted.
- *Data types that cannot be compacted*: Some data structures become invalid as a result of compaction. A classic example is a hash table that hashes on the offset of an object within a segment. Because compaction modifies these offsets, there is no way such an implicit dependence on the segment offset can be accounted for by compaction. Therefore, the compacted hash table becomes invalid. Of course, ObjectStore collections and indexes are valid after compaction.
- ObjectStore versioned data cannot be compacted.
- Since the ObjectStore retain_persistent_addresses facility requires that persistent object locations within a segment remain invariant, no client application using this facility and referencing segments to be compacted can run concurrently with the ObjectStore compactor.
- Transient ObjectStore references into a compacted segment become invalid after compaction completes.

The pathname of the executable is `$OS_ROOTDIR/bin/oscompact.`

10.8 oscp

oscp *source-pathname destination-pathname*

Copying databases

Copies the ObjectStore database *source-pathname* to *destination-pathname*. If the destination database exists, it is deleted before the copy is performed.

Note: oscp takes only non-wildcard pathnames, neither of which may be a directory.

oscp contacts the Server to ensure that the database being copied is transaction-consistent and fully up to date, but **cp** does not. Therefore, you should use **cp** on a file database only if the Server handling access to the database has been shut down with **ossvrshtd**. Otherwise, use **oscp**.

Using **cp** can sometimes produce a database in an inconsistent state (if the database was copied during propagation of data from the log), or in a consistent but out-of-date state (if the effects of some transactions have not yet been propagated from the log to the database). Attempting to operate on an inconsistent copy will fail, signaling `err_inconsistent_db`.

Using **cp** also results in a copy with same database ID as the original, while **oscp** gives a new, unique ID to the copy. This is important only if you have applications that rely on the uniqueness of these IDs. If you use **cp**, you can give the copy a new, unique ID with **os_database::set_new_id()**.

oscp accepts a combination of rawfs pathnames and file pathnames. To move a database from a file to a rawfs or vice versa, you must use **oscp**; **cp** will not work correctly.

The pathname of the executable is `$OS_ROOTDIR/bin/oscp`.

10.9 osdf

osdf *hostname*

Showing disk space and usage

Shows disk space and utilization for the ObjectStore file system on the specified host. For example:

```
% osdf elvis
          Filesystemkbytesusedavail       capacity
          elvis95749533 95215       0%
```

The pathname of the executable is `$OS_ROOTDIR/bin/osdf`.

10.10 osglob

osglob *wordlist*

Performing filename expansion

Performs ObjectStore filename expansion on *wordlist*.

osglob can perform wildcard processing similar to Unix shell wildcards (`*`, `?`, `{}`, and `[]`). You must quote the wildcard with quotation marks (`""`) or a backslash (`\`) to keep the shell from misinterpreting the asterisk as a shell wildcard.

The pathname of the executable is `$OS_ROOTDIR/bin/osglob`.

10.11 oshostof

oshostof *pathname*

Takes one argument, a database pathname, determines what host the database is on, and prints the name of the host to standard output.

oshostof works for both file and rawfs databases. The normal pathname syntax is supported, including the OS_DIRMAN_HOST compatibility feature.

A typical use is as follows:

```
ossvrchkpt ‘oshostof a/b/c‘
```

The pathname of the executable is $OS_ROOTDIR/bin/oshostof.

10.12 osls

osls [-dRlsu] [-Rlu] *pathname* **...**

Listing directory contents

If *pathname* is a directory, **osls** lists the contents of the directory.

The host on which each database resides is identified with its canonical name, even if an alternative name was specified to create it.

osls can perform wildcard processing similar to Unix shell wildcards (*, ?, {}, and []). You must quote the wildcard with quotation marks ("") or a backslash (\) to keep the shell from misinterpreting the asterisk as a shell wildcard.

osls ignores trailing and multiple slashes in pathnames. It accepts a combination of rawfs pathnames and file pathnames.

Options for **osls** are:

Option	Function
-d	Lists the information about the directory itself, rather than the contents.
-R	Corresponds to the shell **ls** command's recursive option.
-u	Lists the uids of the contained databases.
-l	Displays information about directory contents in long format, including the size in bytes.
-s	Causes the size to be displayed in 1 KByte blocks.

The pathname of the executable is $OS_ROOTDIR/bin/osls.

10.13 osmkdir

osmkdir [-p] *directory*

Creating a directory

Creates an ObjectStore directory. The `-p` option recursively creates missing directories to make the supplied directory path exist. **osmkdir** supports all the arguments of the shell command **mkdir**. It accepts a combination of rawfs pathnames and file pathnames.

The pathname of the executable is `$OS_ROOTDIR/bin/osmkdir`.

10.14 osmv

osmv [-fi] *directory1 directory2*

osmv [-fi] *database1 database2*

osmv [-fi] *database directory*

Renaming directories and databases

Renames directories or databases in a file system. The first and second forms rename a directory and a database, respectively. The third form moves the named database to the named directory, retaining the same name in the new directory.

osmv can perform wildcard processing similar to Unix shell wildcards (*, ?, {}, and []). You must quote the wildcard with quotation marks ("") or a backslash (\) to keep the shell from misinterpreting the asterisk as a shell wildcard.

osmv supports all the arguments of the shell command **mv**. It does not accept a combination of rawfs pathnames and file pathnames; pathnames must represent either all rawfs databases and directories or all file databases and operating system directories.

The -f option overrides the -i option as well as mode restrictions.

The -i option enables interactive mode, as for the **mv** command.

Note: using either **mv** or **osmv** can result in an inconsistent database, if the Server crashes immediately following invocation of the command. When the Sever is restarted, it might not be able to locate the database to perform the recovery operations necessary to restore the database to a consistent state. In such a case, attempting to operate on an inconsistent copy will fail, signaling err_inconsistent_db.

The pathname of the executable is $OS_ROOTDIR/bin/osmv.

10.15 osrm

osrm [-firR][-iR] *database...*

osrm -u [-f] *Server-host uid0 uid1 uid2*

Removes ObjectStore databases from the database's host Server, as well as from the directory database. To remove a database, you must have write permission in its directory, but you don't need write access to the database itself.

The first form is used for removing databases from the rawfs and the Server. This is the form normally used by users.

The second form is used for removing databases from the ObjectStore Server only and is normally used only by the system administrator for cleaning up "dangling references" in the Server directory of uids which were located by **osrverf**. To use this form, you must have the same uid as the user who started the Server on host *Server-host. uid0*, *uid1,* and *uid2* are the three parts of the database uid, as obtained by **osls -u** (see **osls**).

osrm accepts these options as command arguments:

Option	Function
-f	Suppresses an error message if the specified database is not found.
-i	Asks whether you want to delete each specified database.
-r	Recursively deletes all databases in the directory database, starting from the selected pathname.
-R	Removes an entry from the rawfs only, without actually deleting the database from the Server. This can be useful if you have dangling references (that is, an entry is in the rawfs, but there is no database on the Server), or if you have moved the entry from one rawfs to another.

osrm can perform wildcard processing similar to Unix shell wildcards (*, ?, {}, and []). You must quote the wildcard with quotation marks ("") or a backslash (\) to keep the shell from misinterpreting the asterisk as a shell wildcard.

osrm supports all the arguments of the shell command **rm**. It accepts a combination of rawfs pathnames and file pathnames.

The pathname of the executable is $OS_ROOTDIR/bin/osrm.

10.16 osrmdir

osrmdir *directory...*

Removing directories

Removes ObjectStore directories from the directory database. To remove a directory, the directory must be empty, and you must have write permission in its parent (but you don't need write access to the directory itself).

osrmdir can perform wildcard processing similar to Unix shell wildcards (`*`, `?`, `{}`, and `[]`). You must quote the wildcard with quotation marks (`""`) or a backslash (`\`) to keep the shell from misinterpreting the asterisk as a shell wildcard.

osrmdir supports all the arguments of the shell command **rmdir**. It accepts a combination of rawfs pathnames and file pathnames.

The pathname of the executable is `$OS_ROOTDIR/bin/osrmdir`.

10.17 ossetasp

ossetasp *executable-pathname database-pathname*

ossetasp -p *executable-pathname*

Application schema

The first form patches the specified executable to look for its application schema in the specified database. The second form (using `-p`) shows the pathname of the specified executable's application schema database.

The pathname of the executable is `$OS_ROOTDIR/bin/ossetasp`.

10.18 ossevol

ossevol workdb schemadb evolvedb+ [*keyword-options*]

Used instead of a call to **os_schema_evolution::evolve()** for evolutions with no user-defined transformer functions, reclassifiers, or illegal pointer handlers.

The arguments are:

Argument	Function
workdb	the work database used during schema evolution
schemadb	the database containing the schema to which to evolve
evolvedb+	the database(s) to be evolved as a unit

The *keyword-options* are:

Keyword option	Function
-task_list *filename*	Specifies the name of the file to which the task list will be written. Use "-" for stdout.
-classes_to_be_removed *class-name(s)*	Specifies the names of the classes to be removed.
-classes_to_be_recycled *class-name(s)*	Specifies the names of the classes to be recycled; by default, the storage associated with all classes is recycled.
-local_references_are_db_relative yes | no	Assumes that all local references are relative to the database, as specified by one of the arguments *yes* or *no* (the default).
-resolve_ambiguous_void_pointers yes | no	Resolves an ambiguous void pointer to the outermost enclosing colocated object as specified by one of the arguments *yes* or *no*. *No* is the default.
-workspace *workspace-name(s)*	Identifies the workspace for versioned databases to be evolved.

Keyword option	Function
`-explanation_level` *n*	A number from *1* to *3*; primarily an internal debugging aid.

The pathname of the executable is `$OS_ROOTDIR/bin/ossevol`.

10.19 ossize

Reporting database and segment sizes

`ossize` *`[options] pathname`*

Reports the size of the specified database and the size of its segments. Can also list the types used by the database and the number of stored instances of each type.

Persistently allocated pointers (that is, pointer to pointers, such as `new(db) thing*` or `new(db) thing*[100]`) are not distinguished as separate types, but are displayed together.

Options are:

Option	Function
`-a`	Prints out the total length of the info segment immediately after the length of the data segment.
`-c`	Prints the type contents for each segment.
`-C`	Prints the type contents for the entire database.
`-D` *`pathname`*	Prints information about a directory database instead of an individual database. *`pathname`* is interpreted as the name of a Server host on which to look for the directory database.
`-f`	Prints information about the location of all free blocks of storage in a segment.
`-n` *`segment-number`*	Prints information only about the segment specified as *`segment-number`*, rather than information about every segment in the database. *`segment-number`* is a "data segment" number, such as those printed by the `-a` option. `-n` is particularly useful in conjunction with `-o` and `-c`, since it reduces the amount of output.
`-o`	Prints a complete table of every *object* in the segment, showing its offset and size. This table contains an enormous amount of data, which can be useful in debugging. Do not confuse this with the `-0` option, described below.
`-ss`	Prints the type summaries by the *space* used by the instances of each type (this is the default).
`-sn`	Prints the type summaries by the *number* of instances of each type.
`-st`	Prints the type summaries alphabetically by *`type-name`*.

Option	Function
`-w` *`workspace-name`*	Runs **`ossize`** with the current workspace set to *`workspace-name`*, which must be the name of a workspace stored in this database. This allows you to examine the size (and contents, with `-c` and `f`) of a particular version of the database. If you don't provide a `-w` argument, the "transient workspace" is used as the current workspace (that is, the usual default). If there is a segment that isn't known by the current workspace, **`ossize`** prints `Error: there is no version of this segment in this workspace`.
`-W`	Prints a list of all the named workspaces that are stored in the specified database. When specified without other arguments, `-W` prints only workspace names, with no information about database size.
`-0`	Causes **`ossize`** to include the internal segment 0 in type summaries. This implies `-c` if neither `-c` nor `-C` is set.

`ossize` prints out the comment for each segment that has a (non–zero-length) comment.

The pathname of the executable is `$OS_ROOTDIR/bin/ossize.`

10.20 ossvrping

ossvrping [-v] [*hostname*]

Reports whether the Server running on the specified host is responding to ping-messages. For example:

```
% ossvrping elvis
     elvis is alive
```

ossvrping defaults first to OS_SERVER_HOST, then to the local host. The -v argument provides more information when it fails to contact the Server.

The pathname of the executable is $OS_ROOTDIR/bin/ossvrping.

10.21 ostest

ostest -dfprsw *pathname*

This command works only on rawfs databases.

Takes one of the following options and an ObjectStore pathname, and returns with an exit code of zero (true) or non-zero (false).

The option is one of:

Option	Function
-d	*pathname* is a directory
-f	*pathname* is a database
-r	requestor has read access to *pathname*
-s	*pathname* is not a directory and has a non-zero size
-w	requestor has write access to *pathname*
-p	*pathname* is a file pathname

The pathname of the executable is $OS_ROOTDIR/bin/ostest.

10.22 osverifydb

osverifydb [-o] [-v] [-m][-w *workspace-name*][-limit *number*] *pathname*

Verifying pointers

Verifies all pointers contained within a database identified by *pathname*. Verification in this case implies:

- there are no transient pointers, and
- persistent pointers point to valid (not deleted) storage, and the declared type for a pointer as determined from the schema matches the actual type of the pointed-to object.

Options are:

Option	Function
-o	Prints out every object in the database using the metaobject value protocol.
-v	Tells **osverifydb** to print every pointer value.
-m	Verifies metaobjects. With this option, any metaobjects encountered when using the -o option are printed.
-w *workspace-name*	Verifies pointers for versioned databases in the specified workspace.
-limit *number*	Limits to *number* the error messages reported for any segment in the database.

When **osverifydb** detects an invalid pointer, it indicates the location and the value of the pointer. Whenever possible, it prints out a symbolic path to the bad pointer, starting with the outermost enclosing object. For example, the following output is the result of running **osverifydb** on a database that contains an object of type c1, with the bad pointers identified by the error messages.

```
beethoven% osverifydb /camper/van
Verifying database beethoven::/camper/van
Verifying segment 2 Size: 8192 bytes

Pointer to non-persistent storage.
 Pointer Location: 0x6010000. Contents: 0x1.
 Lvalue expression for pointer: c1::m1

Pointer type mismatch; the declared type is incompatible with the
```

```
actual type of the object
 Pointer Location: 0x6010004. Contents: 0x601003c.
 Declared type c2*. Actual type: c3*.
 Lvalue expression for pointer: c1::m2

Pointer to deleted storage
 Pointer Location: 0x6010008. Contents: 0x6010040.
 Declared type c2*.
 Lvalue expression for pointer: c1::m3


Pointer type mismatch; the declared type is incompatible with the
actual type of the object
 Pointer Location: 0x601000c. Contents: 0x6010028.
 Declared type c2*. Actual type: c1*.
 Lvalue expression for pointer: c1::m4
 Lvalue expression for pointed to object: c1::ma[5]

Pointer type mismatch; the declared type is incompatible with the
actual type of the object
 Pointer Location: 0x6010010. Contents: 0x6010044.
 Declared type c2*. Actual type: char*.
 Lvalue expression for pointer: c1::m5
 Lvalue expression for pointed to object: char[0]

Pointer to non-persistent storage.
 Pointer Location: 0x6010014. Contents: 0x1.
 Lvalue expression for pointer: c1::ma[0]

Pointer type mismatch; the declared type is incompatible with the
actual type of the object
 Pointer Location: 0x6010028. Contents: 0x601003c.
 Declared type c2*. Actual type: c3*.
 Lvalue expression for pointer: c1::ma[5]

Pointer to non-persistent storage.
 Pointer Location: 0x6010068. Contents: 0x1.
 Lvalue expression for pointer: void*[5]
 Verified 5 objects in segment

Verified 5 objects in database
beethoven%
```

The pathname of the executable is `$OS_ROOTDIR/bin/osverifydb`.

10.23 osversion

osversion

Prints the version of ObjectStore being used on your machine's architecture. Here is the output on a SPARCstation:

```
immerglück% osversion
ObjectStore Release 5.0 Service Pack 3 for Solaris
2.x (SunOS 5.0) SPARC/SunPro
```

The pathname of the executable is `$OS_ROOTDIR/bin/osversion`.

[This page intentionally left blank.]

Index

A

B

C

D

E

F

G

H

I

L

M

N

O

P

Q

R

S

T

U

V

W