Persistent LISP Objects

PLOB!

User's Guide

28. May 2005

Heiko Kirschke

See section **WWW and email addresses** [*97*] for contact addresses to the author.

Part number PLOB–UG–2.11. This document covers PLOB! version 2.11 released at 28. May 2005.

Contents

List of Figures

Chapter 1

Introduction

This chapter gives an overview on persistency and persistent objects in common. It explains the fundamental properties for persistent objects from a LISP perspective, and presents an overview on the the approach used in PLOB! (*P*ersistent *L*isp *OB*jects) for implementing persistent objects. For technical data like supported operating systems, supported LISP systems etc. see section 'Technical data' on page 67.

1.1 Ways to persistency

Focusing on application programming, persistency is an issue in many applications. There are different ways to achieve persistency, each having its benefits and drawbacks:

File oriented persistency: Save and load objects states to and from a file.

File oriented In file-oriented persistency, the state of each object to make persistent is written into a file and read in again later. The only benefit of this approach is that it is simple to implement. One of its drawbacks is that it is data- and system-dependent. Also, relocation and establishing references between objects is hard to achieve. Support for changes on an object's structure is hard to implement. Database-like features, like selecting single objects out of the set of all objects, are also hard to implement.

In the sense of LISP, constructs using methods specialized to generic function **make-load-form** [CLtLII 1990, p. 659] belong into this category of persistency.

Database oriented persistency: Save and load objects states to and from a (relational) database.

Database oriented In database-oriented persistency, the objects are stored in a (relational) database. In relational databases, there is a different paradigm of viewing 'objects' (that means, tuples) as values[1] having no identity independent from their state. Because of the different programming paradigms, the interfacing between a relational database and a programming language can get quite complex and obscure (see for example ESQL for a binding between SQL and C); this is often called 'impedance mismatch'. Because there are certain means of establishing a mapping between object-oriented and database concepts (figure 1.1), this approach can make sometimes usage of database features, like complex queries, transactions etc. Prerequisite is that this mapping from object-oriented into database concepts is simple

[1]Values will be defined in section 'Objects and classes' on page 4.

Object-oriented	**Relational Database**
Class	Relation
Instance	Tuple
Slot	Attribute

Figure 1.1: Mapping between o-o and relational database concepts

and straight forward enough so that the database manipulation functions can work on the mapped data at all. For example, to administrate data efficiently, attributes must be typed statically in most relational databases; so, this approach results either in static typed slots of objects (which is inappropiate for a dynamic type language as LISP is) or this mismatch requires rather expensive mapping operations. In object-oriented systems, there is a notion of objects having an identity, making each one unique among the set of all objects considered; a (single directed) reference between two objects is established by using an appropiate representation of an object's identity, for example by using its memory address. In a relational database, this has to be mimiced by using (bi directional) expensive joins. In databases, aggregate types are represented implicit by relations; especially in LISP, an explicit type representation by class metaobjects is preferred.

Because LISP has no standardized relational database interface, only single examples can be quoted, as the SQL interface offerred by LISPWORKS and ALLEGRO Common LISP.

Object-oriented persistency: Persistency as an orthogonal feature of each object.

Object-oriented In object-oriented persistency, persistent objects are viewed as usual in an object-oriented sense, and to be more specific in this context, as LISP or CLOS objects. This has some consequences on the design of an object-oriented database. For example, each LISP object has a (more or less) explicit represented class, and since objects are typed dynamically, each object knows of its type. Also, LISP has a notion of identity of objects for establishing references between them, and this identity should be represented for persistent objects, too.

Besides the system described here, ALLEGROSTORE and Itasca can be mentioned as examples for object-oriented databases.

There is no common calculus for object-oriented databases.

Relational databases are now quite mature.

The object-oriented approach has been favored in PLOB, although there are some theoretical and some practical drawbacks on this approach. The theoretical drawback is that there is no calculus on object-oriented persistency as the relational calculus for relational databases. Relying on the relational calculus, some mathematical attractive properties can be shown, like the existence of normal forms to avoid redundancies and anomalies; this does not hold for object-oriented persistency. From a very stringent point of view, object-oriented persistency can even be seen as a step back towards hierarchical databases, which have been wide-spread in the sixties and have been the motivation for Codd to develop the concept of relational databases [Codd 1979]. The practical drawbacks have more or less to do with the fact that usable relational databases have now been on the market for at least 15 years, so they are now quite mature and have sophisticated query optimization techniques, good transaction behavior etc.[2] These are also features which should be found in an object-oriented database, but naturally it takes some time to code and test these features.

[2]It took quite long for the market since Codd's original publication on relational databases in 1969 to provide a 'real' relational database.

1.2 The approach taken by PLOB!

Start working on PLOB, the aim was and is to implement a database with certain features, some more external features from viewing towards PLOB's API, some more internal features for the overall design. Some of them are borrowed from *The Object-Oriented Database System Manifesto* [Atkinson et al. 1992].

Persistency should be orthogonal and transparent.

At first, persistency should be an orthogonal property w.r.t. LISP's type system; in practice, this means that each LISP and CLOS object should have the possibility of becoming persistent, independent of its actual type. Persistency should be as transparent as possible, meaning that users should not need to learn a new programming language for handling persistent objects, but rather integrate persistency into their applications with minimal or no changes, covering both new and legacy systems. Transparency in a wider sense also lead to integrate some of the transient LISP environment into PLOB, like persistent packages and persistent symbols, and implementing persistency by reachability. Other important features are *database features*, like transactions, locking, queries (at least to some extent), *efficiency* and *performance*. All of these features have been taken into consideration or have been implemented in PLOB.

Features of LISP should be integrated; database features should be added.

Looking towards the internal features, it was decided to represent types and classes of persistent objects explicit, similar the way LISP does it; this simplifies the usage of persistent objects, since there is no necessity for a mapping between different paradigms as shown in figure 1.1. Since LISP offers a lot of useful and efficiently represented numeric and non-numeric types, it was decided to represent all of those types in PLOB, too. PLOB! distinguishes internally between representations for instances of immediates[3], built in classes, structure classes and 'true' CLOS classes (figure 1.2), all

Explicite representation of types and classes as objects in PLOB.

Demand for type completeness.

Efficient and appropiate representation of persistent objects.

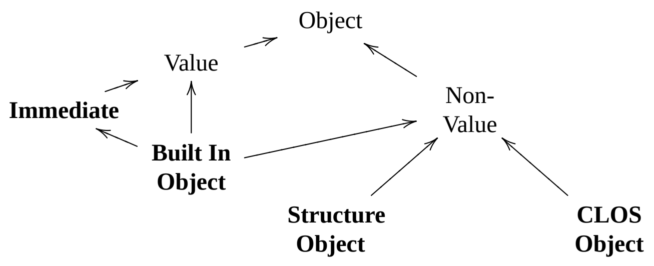


Figure 1.2: Taxonomy of object representations

of them except the immediates extended by some database-specific slots. The principle of self-describing structures was obeyed, which may be characterized by two axioms:

1. Each object has a class.
2. Classes are represented by objects.

Exactly as in LISP, structure classes have some restrictions compared to CLOS classes, the most important one that schema evolution[4] is supported for instances of structure classes only if either the number of slots did not increase or a structure class' instance undergoing a schema evolution is allowed to change its persistent identity. For instances of persistent CLOS classes, full schema evolution without loss of identity is supported.

The experience with integrated systems in the past showed that integration is most of the time useful when doing everyday work. When it comes to certain 'special tasks' (for example, for performance or efficiency reasons), it is sometimes necessary to aban-

Working 'directly' on persistent objects.

[3]Immediates will be defined in section 'Objects and classes' on page 4.

[4]Schema evolution will be defined in section 'Schema evolution' on page 7.

don integration and to do it on her or his own. When using a system which prevents bypassing integration, these tasks are hard to solve. A consequence for PLOB! is that the transparent interface should and can be used, but it is also possible to work directly on low-level representations of persistent objects. This is important for LISP objects which are out of control for PLOB! and are modified by side effects, for example, lists which are modified destructively.

1.3 Related documentation

This document describes how to use PLOB's API to make objects persistent, and how the system itself is administrated. The more low-level documentation accompaning this user's guide can be found in [API Manual 1997], which describes in detail each constant, variable, macro, class, function, generic function and method implemented in PLOB! and exported from package `PLOB`. [Internal Manual 1997] describes the internal program constructs not exported from package `PLOB`. [Kirschke 1995] is an abstract giving a course overview on PLOB! and is a supplement to this introduction.

The fundamental principles of PLOB! are explained in my (german-written) computer science master's thesis [Kirschke 1994]. If you are interested in making a reference manual by extracting documentation from LISP source files see [LispDoc Manual 1994].

More background information on (relational) databases can be found in [O'Neil 1994]. A vast amount of transaction techniques for databases is explained in [Gray & Reuter 1993].

1.4 Notion used

The margin notes summarize a section of text or give some other background information.

`Typewriter type` is used for programming examples and LISP code. Code which is important in the current context is emphasized by `underlining`. Using a listener prompt `CL-USER 1 >` in front of a code example means that the example can be typed in directly into a listener. This is a <*non-terminal*>, meant to be replaced by some concrete object. Repetitions are indicated by curly braces, so `{` <*repeat me*> `}` means 0 to *n* occurencies of the non-terminal <*repeat me*>.

References A paragraph labeled **References** gives references to other sections or documents.

1.5 Terms used

This section defines some of the terms used within this document, some having to do with object-oriented programming in common, others having to do with databases.

1.5.1 Objects and classes

Object

An *object* is the abstraction of a real-world entity; it owns an *identity* and has a *state* and a *behavior*.

Identity

Identity is a property which makes an object unique among the set of all objects in consideration; by its identity it can be distinguished from all other objects.

Value

A value is an object whose identity can be completely defined by its state, that means, two values are identical if and only if they have the same state. In this context, a value is further restricted to be an instance of a LISP's built in class, like a numbers or a character. With 'real' (non-value) objects, their identity is independent from their current state, that means, two non-value objects having the same state are equal, but not necessarily identical. With a *representation of identity* an object can be referenced.

Immediate

An Immediate is a value object whose state itself is represented directly in a representation of identity, because its state needs less memory than an identity representation. It differs from a value only by its special form of representation. Candidate classes for an immediate representation are for example fixnums, that means, integer numbers with their absolute value $< 2^n$, with $n <$ the number of bits used for an identity representation.

Class

More than `this`

— Roxy Music

The structure of an object's state and its behavior is defined by its *class*. The structure of an object's state is an aggregation of *slots*; a *slot state* is the partial state of an object represented in a single slot. To be accordant with CLOS' terminology, a slot state is called also a *slot value*. The behavior of an object is made up by its *methods*; each single method defines a part of the object's behavior. Classes are represented by class metaobjects, slots by slot metaobjects, and methods are represented by method metaobjects.

1.5.2 Database terms

Reachability

Persistency by reachability means that all objects which are reachable from a designated persistent root object will remain persistent. Objects not being reachable from this root object will be deleted with the next garbage collection. This closely resembles the concept of *indefinite extent* as defined in [CLtLII 1990, p. 43]. Further details will be given in section 'Making objects reachable' on page 48.

Orthogonal and transparent persistency

Persistency is orthogonal if it is a base property of an object which is independent from other object properties, like inheriting from a certain class. Persistency is transparent if there are no or few changes necessary to link persistency into an application's code.

Transaction

A transaction is a database state changing operation which fulfills the following ACID properties [Gray & Reuter 1993, p. 166]:

Atomicity A transaction's changes to the state are atomic: either all happen or none happen. These changes include database changes, messages, and actions on transducers.

Consistency A transaction is a correct transformation of state. The actions taken as a group do not violate any of the integrity constraints associated with the state. This requires the transaction be a correct program.

Isolation Even though transactions execute concurrently, it appears to each transaction, T, that others executed either before T or after T, but not both.

Durability Once a transaction completes successfully (commits), its changes to the state survive failures.

Figure 1.3 shows what happens during a transaction: At t_n, a transaction is started

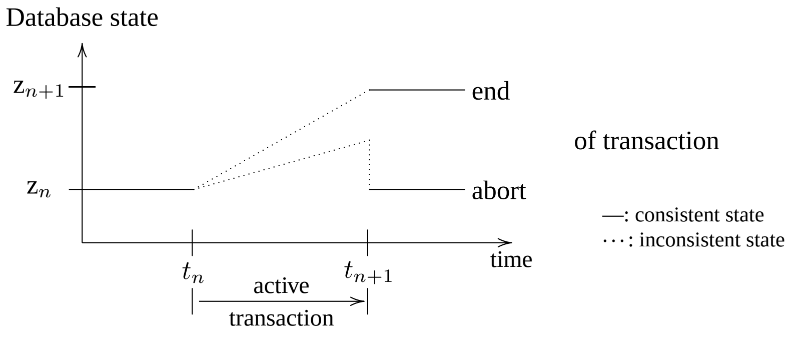


Figure 1.3: State change at ending or aborting a transaction

which lasts until t_{n+1}. At t_{n+1}, the transaction is either ended (comitted) or aborted (for example, if an error occurs during the transaction being active). The database states z_n and z_{n+1} are defined as being consistent from the application's point of view before t_n or after t_{n+1}. In the interval $[t_n, t_{n+1}]$, the database may become temporary inconsistent, which is shielded to other concurrent transactions by isolation.

Index and query

An *index* is an associative memory mapping a key to a value. In a database, an index is usually bound to one or more slots of a class, with imposing a 1-dimensional key ordering when using more than one slot. It maps a slot value (as an index key) to the object having that slot value (as the value associated to the index' key). This association is maintained more or less automatically by the database system itself. In general, the keys of the index are ordered, for better performance on selecting objects falling into a range of slot values. As will be shown in 'Indexes and reachability' on page 26, an index will also ensure reachability for a persistent object.

A *query* selects objects out of the set of all objects of a class or aggregates information on a [sub]set of these objects; the condition used for discrimination is usually some specific slot value or a range of slot values. So, an index is used to speed up a query.

For example, let us assume that a class **person** maintains an index on its slot `soc-sec-#`. This index will maintain the mapping of `soc-sec-#`s to the **person** instance bound to that `soc-sec-#`:

```
① CL-USER 2 > (make-instance 'person :soc-sec-# 2000)
  #<instance person soc-sec-#=2000 short-objid=50326730>
② CL-USER 3 > (make-instance 'person :soc-sec-# 3000)
  #<instance person soc-sec-#=3000 short-objid=50326732>
③ CL-USER 4 > (p-select 'person :where 'soc-sec-# := 2000)
  #<instance person soc-sec-#=2000 short-objid=50326730>
```

The mappings are established at ① and ②, and the index is accessed with the query at ③.

Schema evolution

A schema evolution means changing the definition ('schema' in database terminology) of a persistent class stored in a database and what should happen to the instances whose class has been redefined. In general, there are two ways to handle this, either a *destructive* or *non-destructive* proceeding. In the destructive approach, a mapping from the instance' structure as given by the 'old' definition to the instance' structure as given by the 'new' definition is established, and the instance slots are rearranged permanently to match the 'new' definition. With the non-destructive approach, the mapping is done too, but the instance is not written back into the database, but used only to pretend the 'new' structure to the upper application levels requesting the object's state.

Schema evolution can be done destructive or non-destructive.

Chapter 2

Quick tour through PLOB!

This chapter shows in a few example statements how to work with PLOB! to make objects persistent. Each of the concepts shown here is explained in detail in one of the following chapters of the manual; references to these chapters are given at the end of each section. The examples assume that PLOB! has been installed as described in [Installation Guide 1997]. The example code itself is in `plob-2.11/src/example/plob-quick-tour.lisp`

2.1 Starting LISP and loading PLOB!

At first, ALLEGRO Common LISP is started and the system itself is loaded:

```
/home/kirschke>cd plob-2.11/src/allegro
/home/kirschke/plob-2.11/src/allegro>cl
Allegro CL 4.3 [Linux/X86; R1] (8/5/97 16:30)
;;...
CL-USER 5 > (load "defsystem-plob")
; Fast loading ./defsystem-plob.fasl
T
CL-USER 6 > (load-plob)
; Loading system: "PLOB!".
;   Loading product for module: "PLOB:SOURCE;plob-defpackage".
;;...
; Fast loading PLOB:SOURCE;plob-sexpr.fasl
;    (/home/kirschke/plob-2.11/src/allegro/plob-sexpr.fasl)
T
```

Omissions of unimportant system messages are marked by ‘;; . . . ’.

References Section ‘Starting on the client side’ on page 69; section **bootstrap ...** [API Manual 1997, p. 6].

2.2 Opening the database

Next, the database session to PLOB! is opened explicitly by a call to function **open-my-session** [API Manual 1997, p. 45]; this step could also be omitted, PLOB! will open the database when it is needed the first time. The next example code assumes that the lisp root is already formatted; this is done automatically the very first time a database is opened.

```
    CL-USER 7 > (open-my-session)
①   ;;;; Bootstrap   : Opening tcp://nthost/database
    ;;; Info from server at executing client:SH_short_read_root:
    ;;; ====================================================
    ;;; PLOB! daemon version 2.11 database version 2.11 - WIN32
    ;;; ====================================================
    ;;; Copyright (C) 1994-2005 Heiko Kirschke Heiko.Kirschke@acm.org
    ;;; Part of server code (C) University of St. Andrews
    ;;; PLOB! comes with ABSOLUTELY NO WARRANTY; for details, look into the
    ;;; user's guide 'plob/ps/userg.ps' provided within the distribution.
    ;;;; Bootstrap(1): Loaded PLOB::STRUCTURE-DESCRIPTION, 11/13 slots.
    ;;...
    ;;;; Bootstrap(4): Loaded PLOB::METHOD-DESCRIPTION, 5/6 slots.
    ;;;; LISP root formatted at 1997/11/21 16:56 by ALLEGRO
②   #<heap kirschke@nthost 'Initial Lisp Listener' short-objid=50327650>
```

This bootstrap is always done the first time a database is opened; the bootstrap loads some metaobjects from the persistent storage into the LISP image. The first line of output ① shows the database opened; it uses TCP/IP as transmission protocol (`tcp:`), the server host is `nthost` (a remote Windows/NT PC) and the database opened is located in subdirectory `database` relative to the server's installation directory. The call returned an object representing the session to the database ②; this object is stored in variable ***default-persistent-heap*** [API Manual 1997, p. 30] and will be used as default session for all database operations.

References Section 'Session' on page 46

2.3 Showing other sessions active on the database

Now let's look who is on the database:

```
    CL-USER 8 > (show-sessions)
      Sessions on tcp://nthost/plob
①   * #<heap kirschke@nthost 'Initial Lisp Listener'
             transaction=514 short-objid=50327650>
②   . #<heap kirschke@nthost 'Metaheap' short-objid=50329172>
```

There are 2 sessions started from the current LISP process; the first one ① is the actual session (marked by a '`*`'), the other one ② is a session used by PLOB! internally for storing and loading of metaobjects.

References Section 'Session' on page 46

2.4 Defining a persistent class

The most simple way to make CLOS classes and their instances persistent is by adding the class option `(:metaclass persistent-metaclass)`:

```
CL-USER 9 > (defclass person ()
              ((name :initarg :name :initform nil :accessor person-name)
               (soc-sec-# :initarg :soc-sec-#
                          :accessor person-soc-sec-#
                          :index (btree :test equal)))
```

```
            (:metaclass persistent-metaclass))
#<PERSISTENT-METACLASS PERSON>
```

This example uses the tight-binding interface of PLOB! for persistent objects. As will be shown later, persistency is available too for classes whose definition cannot or should not be changed by adding the class option `:metaclass persistent-metaclass`. Those classes use the loose binding interface of PLOB!, which differs from the tight-binding interface by the lack of some functionality.

The class itself is not yet persistent; this will be done the first time an instance of the class is created and stored.

2.5 Binding to a persistent symbol

```
CL-USER 10 > (setf #!*cleese*                                              ①
            (make-instance 'person :name "Cleese" :soc-sec-# 2000))
;;;;; Stored #<instance class-description person 1.00                    ②
;;;;;                   slots=2/3 short-objid=50327268>.
#<instance person soc-sec-#=2000 short-objid=50326603>                     ③
```

The Common LISP macro reader construct `#!`<*symbol*> at ① introduces a persistent symbol with the same name and package like the transient <*symbol*>. Persistent symbols are used as 'entry points' into the database, to reference persistent objects by a name. An object bound to a persistent symbol is always reachable. Therefore, the symbol's value is persistent, too, and will never be removed by a garbage collection. The persistent symbol is a symbol on its own; the only relation to its transient companion is that they share the same name and package name.

The output ② shows that a class description for class **person** is stored in the database; the name of the class is `person`, its version number is `1.00`; it has 3 slots, 2 are persistent slots and 1 is a transient slot (the last one is added silently by PLOB! at establishing the class metaobject representing class **person**). For the slot named `soc-sec-#`, an index will be maintained automatically which maps a value to the instance having that slot value.

The call shown here ③ returned the now persistent object which will be bound to the persistent symbol `*cleese*`. For slots with a declared index, the slot name and its value will be shown in the print representation.

References Section 'Tight binding' on page 19; section 'Declaring an index on a slot' on page 24; section 'Persistent symbols' on page 49.

2.6 Making persistent instances

Now, some more persistent instances are created:

```
CL-USER 11 > (make-instance 'person :name "Palin" :soc-sec-# 3000)
#<instance person soc-sec-#=3000 short-objid=50326605>
CL-USER 12 > (make-instance 'person :name "Gilliam" :soc-sec-# 4000)
#<instance person soc-sec-#=4000 short-objid=50326607>
```

Besides using persistent symbols, this example shows the second way to ensure reachability for a persistent object. Indexes are defined as being reachable, so the index declared on slot `soc-sec-#` will ensure reachability for all instances of class **person**, although not every individual **person** instance may be bound to a persistent symbol.

References Section 'Indexes and reachability' on page 26.

2.7 Doing simple queries

Get the instance of class **person** whose slot value of slot `soc-sec-#` equals to 2000:

```
CL-USER 13 > (p-select 'person :where 'soc-sec-# := 2000)
#<instance person soc-sec-#=2000 short-objid=50326603>
CL-USER 14 > (person-name *)
"Cleese"
```

This is the person named `"Cleese"` which has been stored above as the very first instance of class **person**. The query accessed the index declared for slot `soc-sec-#`.

References Section 'Querying an index' on page 25.

2.8 Defining a print method

Since the *objid* is rather poor for identifying the person instance, a print method will be defined:

```
CL-USER 15 > (defmethod print-object ((obj person) stream)
               (print-unreadable-object (obj stream :type t :identity t)
                 (format stream "~A (short-objid ~A)"
                         (if (slot-boundp obj 'name) (person-name obj))
                         (persistent-object-objid obj))))
```

Now, the output of the simple query will look somewhat better:

```
CL-USER 16 > (p-select 'person :where 'soc-sec-# := 2000)
#<PERSON Cleese (short-objid=50326603) @ #xa064f02>
```

Unfortunately, the method itself cannot be stored in the database, since PLOB! does not support storing of binary function code; section 'Binary function code' on page 36 explaines the workarounds which are implemented in PLOB! to cope with function code.

2.9 Leaving LISP

When leaving LISP, all open sessions to the database are closed automatically. A close of all sessions can be forced by calling generic function **close-heap** [API Manual 1997, p. 28].

```
CL-USER 17 > :exit
cplobheap.c(196): fnDeinitializeHeapModule:
  PLOB! is trying to close 2 pending session(s), please wait ... done!
```

References Section 'Finishing on the client side' on page 69

2.10 Restarting the system; reload of classes

After a restart of the LISP system and a reload of PLOB! (as described in section 'Starting LISP and loading PLOB' on page 9), loading the value of a persistent symbol will re-establish the class of the object found in the value cell of the persistent symbol into the current LISP image:

```
/home/kirschke/plob-2.11/src/allegro>cl
;; ...
CL-USER 1 > (load "defsystem-plob")
;; ...
CL-USER 2 > (load-plob)
;; ...
CL-USER 3 > #!*cleese*
;;;; Bootstrap   : Opening tcp://nthost/database                   ①
;; ...
;;;; LISP root formatted at 1997/11/21 16:56 by ALLEGRO
;;;;; Compiling #<instance class-description person 1.00             ②
;;;;;                     slots=2/3 short-objid=50327268>
#<instance person soc-sec-#=2000 short-objid=50326603>              ③
```

The output ① shows that the database is opened implicit at loading the persistent symbol, and that a class description has been compiled into the current LISP image ②. This compilation will re-establish the class as it has been specified within its original definition as given with the `defclass` statement in section 'Defining a persistent class' on page 10; all superclasses, slots, slot initargs, slot initforms, slot accessors etc. will be re-established, too. The return value ③ is the value of the persistent symbol `*cleese*`:

```
CL-USER 4 > (person-name *)
"Cleese"
```

2.11 Doing a schema evolution

Adding, removing or changing a slot of a class definition will result in a modification of the persisten class description:

```
CL-USER 5 > (defclass person ()
              ((name :initarg :name :initform nil :accessor person-name)
               (soc-sec-# :initarg :soc-sec-#
                          :accessor person-soc-sec-#
                          :index (btree :test equal))
               ;; Adding slot age:
               (age :initarg :age :initform nil :accessor person-age))
              (:metaclass persistent-metaclass))
#<PERSISTENT-METACLASS PERSON>
```

Next time an instance of class **person** is loaded or stored, the new class description will be stored within the database, and the instance loaded or stored will be updated to this new class description. In the following example, a reload of the top-level state of the object bound to the persistent symbol `*cleese*` is done to enforce the new class description being stored to the database:

```
CL-USER 6 > #!(*cleese* :flat)                                      ①
Error: The persistent definition for class
PERSON does not match its transient counterpart.
       Reason: total number of slots; tr.: 4 / pe.: 3               ②
  [condition type: SIMPLE-ERROR]

Restart actions (select using :continue):
 0: Store the transient definition to the persistent store.
 1: Replace the transient definition by its persistent counterpart.
 2: Ignore definition mismatch (might result in LISP runtime errors).
 3: Show the transient class definition.
 4: Show the persistent class definition.
[1] CL-USER 7 > :cont
```

```
;;;;; Updated #<instance class-description person 1.01
;;;;;                   slots=3/4 short-objid=50327096>.
#<instance person soc-sec-#=2000 short-objid=50326603>
```

The evaluation ① forces a reload of the top-level state of the object bound to the persistent symbol (this is done by the `:flat` argument given to the persistent symbol reader). An error is raised ② which shows the reason for the schema evolution (the number of slots has been increased to 4 slots).

References Section 'Schema evolution' on page 62.

2.12 Using transactions

Each read and write access to the database is guarded by a transaction (and locking). Transactions are used for ensuring consistent changes to the database: Either all or none of the changes done to objects states during a transaction are stored in the database. They can be used explicit by embedding the code which should do such a consistent change into macro **with-transaction** [API Manual 1997, p. 135]:

```
CL-USER 8 > (with-transaction ()
              (setf (person-age #!*cleese*) 52)
              ;; <some other code in between>
              (setf (person-soc-sec-# #!*cleese*) 2001))
```

This call will ensure that either both slots are changed or none of them: Executing the whole `(with-transaction ...)` block will result in the changes written to the database. Each jump out of the `(with-transaction ...)` block will result in a transaction's abort, which will reset the slot states to their value before the block was entered. This jump out-of-the-block happens for example when *<some other code in between>* raises a runtime error. [1]

References Section 'Transaction' on page 46.

2.13 Defining and using btrees

Persistent btrees are used internally for many purposes (for example, for maintaining the persistent package and symbol tables, and for indexes), so it was decided to make them available to the user API. Btrees are similar to hash tables, with the difference that their keys are sorted. Similar to hash tables, they come with two (not three) test modes `eq` and `equal`. A test mode of `eq` means that the btree's keys are sorted by identity, whereas a test mode of `equal` will sort the keys accroding to their state.

```
CL-USER 9 > (setf #!*btree* (make-btree :test 'equal))
#<btree equal 0/0 short-objid=50327317>
CL-USER 10 > (setf (getbtree "Cleese" #!*btree*) #!*cleese*)
#<instance person soc-sec-#=2000 short-objid=50326603>
CL-USER 11 > (setf (getbtree "myself" #!*btree*) "Heiko Kirschke")
"Heiko Kirschke"
CL-USER 12 > (p-apropos-btree #!*btree*)
"Clesse", data: #<instance person short-objid=50326603>
"myself", data: #<simple-string 'Heiko Kirschke' short-objid=50327186>
```

[1] For now, a transaction's abort will only affect the persistent representation of the object; its transient representation remains unchanged and may therefore contain some new state.

References Section ‘Persistent btrees’ on page 52.

2.14 Server log file

The server reports important actions, errors and information messages to a log file containing plain text. The name of the file is `messages.log`. This file is located on the server host in the database directory directly below the database root directory specified at PLOB’s installation. So, when encountering any problems with running PLOB!, a look into the server’s log file might help. A typical excerpt looks like this:

```
1998/01/07 10:41:18 kirschke@nthost                                      ①
       splob.c(462): fnServerInitializePlob:
       Started ./plobd                                                   ②
1998/01/07 10:41:26 #<heap kirschke@nthost ‘Metaheap’
                                           short-objid=50328853>
       splob.c(1926): fnSHopen (#’c-sh-open):
       Client kirschke@nthost, now 1 client(s):                          ③
       Opening stable heap ‘plob’ succeeded.
       Reason: Administrator login
1998/01/07 10:41:46 #<heap kirschke@nthost ‘Metaheap’
                                           short-objid=50328853>
       splob.c(1190): fnSHclose (#’c-sh-close):
       Closed stable heap ‘plob’,                                        ④
       now 0 client(s).
       Garbage collection resulted in 4317 - 1524 = 2793 objects.
```

At ①, each message shows the date and time it was generated and the current client which was responsible for the raised message. The source code location and C function which raised the message is shown, too, and the message itself ②. All physical database openings are reported ③; when the last client disconnects, a garbage collection is triggered, which writes the number of deleted objecs into the log file ④.

Chapter 3

Using PLOB!

A rose is a rose is a rose is a rose.

— Gertrude Stein

```
(defun is-a (the-object the-class-object)
  (when (eq (class-of the-object) the-class-object)
    the-class-object))
(is-a (is-a (is-a (find-class 'standard-class)
                  (find-class 'standard-class))
            (find-class 'standard-class))
      (find-class 'standard-class))
```

— CLOS

This chapter focuses on PLOB's API, starting from the mostly transparent high-level CLOS interface going towards the LISP objects interface and the direct persistent object interface. Persistency is supported with three kinds of operation:

Tight binding The tight binding is available for instances of CLOS classes intended to use persistency explicitely (section 'Tight binding' on page 19). This interface is the most transparent one.

Loose binding The loose binding gives the possibility of making any LISP object persistent where either the class definition is out of scope or not available at all (section 'Loose Binding' on page 26). It can also be used for instances of CLOS classes where persistency is not a so important feature to pay the (performance) price of the tight binding.

Direct representation It is also possible to work on persistent objects directly (section 'Direct Interface' on page 28). Since PLOB's typing is adapted to LISP, this interface looks similar to the interface defined in [CLtLII 1990]. Also, instances of database specific types can be manipulated directly, for example persistent btrees (section 'Persistent btrees' on page 52).

All these interfaces can be mixed; a decision for using one of those interfaces at a certain place does not exclude usage of another (but lower-level) interface at another place.

Many of the examples given in the following text correspond to the code found in module `plob-2.11/src/example/plob-example`, so it is a completement to the more low-level example module documentation of the example code in chapter **Example Module** [API Manual 1997, p. 139].

3.1 Objects and identity

An object's represented identity makes it possible to refer it.

A reference to an object can be achieved by thinking of an object's *identity*: The identity of an object is something unique to it, making it distingushable from all other objects in a set. So, a representation of identity can be used for establishing an unambigous reference to an object. In transient LISP systems, the memory address of a non-immediate object is used for representing its identity (for ALLEGRO Common LISP, the hexadecimal number behind the '@' character in the output of the standard print-method). In a LISP system, these identity representations are almost never visible at top level, since references on top level are handled by symbols; anyway, LISP has a defined notion of identity for objects.

3.1.1 Identification of persistent objects by their *objid*

We are using tokens or tokens of tokens as long as we do not have access to the things themselves.

— Umberto Eco: The Name of the Rose

In PLOB, the representation of a persistent object's identity is not hidden in the internals, but visible at the top-level API, for the purpose of working directly on the persistent object's representation. This representation is called *objid* (*obj*ect *id*entifier), so an *objid* can be used to reference a persistent object. As explained later in section 'Persistent symbols' on page 49), the LISP way-of-working with persistent symbols referencing persistent objects is possible, too.

In PLOB, objids are represented by fixnums.

An *objid* itself is represented by a fixnum; sometimes it is necessary to distinguish an *objid* from a fixnum. This is accomplished by wrapping the fixnum into an instance of class **persistent-object** [API Manual 1997, p. 106] for non-immediate persistent objects and, if necessary, into an instance of class **persistent-immediate-object** [Internal Manual 1997, p. 172] for immediate objects. The correct class is choosen by function **make-persistent-object** [API Manual 1997, p. 43].

References Function **make-persistent-object** [API Manual 1997, p. 43]

3.1.2 Identity, persistent and transient objects

Objects are represented twice, in persistent and transient memory.

Object access in PLOB! is done by *swizzling*: A persistent object's state is transferred from the server's objectspace into a transient representation of the client's objectspace, and the state manipulating functions of the client work on such a transient counterpart. At appropiate moments, both representations are equalized (for example, when calling **store-object** [API Manual 1997, p. 132] or **load-object** [API Manual 1997, p. 38], see section 'Loose Binding' on page 26). For doing such an equalization, the reference of a transient representation to its persistent object must be known, to transfer the transient state into the 'correct' persistent object, that means, the persistent object the transient representation was filled from. This reference itself is represented by an *objid.* For transient representations generated using the tight binding (section 'Tight binding' on page 19), this *objid* is stored within each transient representation and can be requested when necessary. For CLOS instances not using the tight binding and instances of all other LISP classes, a table is maintained mapping a transient representation onto its *objid.* The *objid* of a transient representation can be requested by a call to generic function **persistent-object-objid** [API Manual 1997, p. 106].

References Generic function **persistent-object-objid** [API Manual 1997, p. 106].

3.1.3 Identity and caching

Entia non sunt multiplicanda praeter necessitatem.
[Entities must not be copied unnecessarily.]

— William of Occam

Since this table references *objids* and transient representations, it serves as an object cache, too.At the moment, there is the unsatisfying situation that either the cache has to be cleared sometimes explicitely (section 'Cache littering LISP memory' on page 91) or is cleared by PLOB! itself (section 'Aborting a transaction' on page 47). This clearing will break the binding between all transient representations contained in this table to their persistent counterparts.

For strings and numbers, parameter ***cache-strings*** [Internal Manual 1997, p. 17] and parameter ***cache-numbers*** [Internal Manual 1997, p. 17] decide if strings and numbers should be cached at all. Especially numbers often have a more 'immediate' character, and so it is desirable not to maintain the persistent identity of each of such instances, since this would fill the cache with many small objects.

References Function **clear-cache** [API Manual 1997, p. 28]; parameter ***cache-strings*** [Internal Manual 1997, p. 17], parameter ***cache-numbers*** [Internal Manual 1997, p. 17].

3.2 Making CLOS objects persistent: Tight binding

Within the tight binding, PLOB! takes control over CLOS instances.

A tight binding means that PLOB! takes control over the creation and state of a persistent CLOS object. All creation, state requesting and state changing calls to a persistent object with a tight bounded class are bypassed to PLOB. The tight binding itself is achieved with the help of the Meta Object Protocol (MOP), an integral part of CLOS and is described in detail in [AMOP 1991]. Because of technical restrictions, the tight binding can only be offered for CLOS classes. The binding is obtained by adding the class option `:metaclass persistent-metaclass` to a class declaration:

```
(defclass <class name> ({<superclasses>})
  ({<slots>})
  (:metaclass persistent-metaclass))
```

Using this class option, additional slot and class options are enabled, which can be used to steer persistency for the class declared. Section 'Top level declarations' on page 38 contains further details how persistency can be influenced by top level declarations.

3.2.1 Additional slot options

The tight binding adds the slot options `:extent`, `:index` and `:location`.

Within the tight binding, additional slot options can be used to steer persistency on a per-slot level. Here is an overview of the additional slot options:

```
(defclass <class name> ({<superclasses>})
  ((<slot name>
    :extent <keyword>
    :index <index definition> ;; Explained in section 3.3, p. 24
    :location <fixnum constant>
    { <other slot options> } )
  { <more slots> } )
 (:metaclass persistent-metaclass))
```

The slot option `:allocation` is obeyed by PLOB! as defined in [CLtLII 1990, p. 776, 779]; in other words, shared slots among instances of a class are made persistent, too, and keep their property of being shared slots.

Slot option `:extent`

The slot option `:extent` tells how a slot is represented.

Declare the extent of the slot. Normally, each slot of a persistent object is represented twice, in persistent and transient memory (the second representation being for performance reasons). The `:extent` slot option influences when and how the slot's state is transferred between the transient and persistent memory (figures 3.1 and 3.2).

Value of slot option `:extent`	**Represent. Tr.**	**Represent. Pe.**	**Explanation**
`:transient`	Yes	No	The slot will only be represented in transient memory but will not be represented in persistent memory. This slot is 'ignored' by PLOB! w.r.t. persistency.
`:cached`	Yes	Yes	The slot will be represented both in transient and persistent memory. Slot reading is done from the transient slot's state. Slot writing only affects the transient slot's state; no changes of the transient representation will be propagated automatically to persistent memory. The slot's state will only be promoted to persistent memory by a call to generic function **store-object** [API Manual 1997, p. 132] on the CLOS instance the slot belongs to.
`:cached-write-through`	Yes	Yes	The slot will be represented in transient and persistent memory. Slot reading is done from the transient slot's state. Changes of the transient representation will be propagated automatically and immediately to persistent memory.
`:persistent`	No	Yes	The slot will only be represented in persistent memory but will not be represented in transient memory. Reading is done directly from the persistent memory. For writing, this slot extent implies that changes to the slot's state will be propagated automatically to persistent memory. Use of this extent is depreciated, since it is very resource consuming.
`:object`	Yes	Yes	This extent indicates that a direct persistent representation should be used for the slot's state: The slot will not contain a transient representation of a persistent object, but a forward object referencing the persistent object.
`:objid`	Yes	Yes	This extent indicates that a direct persistent representation should be used for the slot's state: The slot will not contain a transient representation of a persistent object, but a numeric *objid* referencing the persistent object.

Figure 3.1: Slot extent, transient and persistent object representation

Figure 3.2 shows how the slot extent affects the slot representation and promoting of the slot's state between the transient and persistent representation of an object; the structure and class definitions of this example are:

```
(defstruct persistent-example-structure slot-1 slot-2)
(setf (class-extent (find-class 'persistent-example-structure))
      :cached-demand-load)
```

```
(defclass persistent-example-class ()
  ((slot-3  :extent :transient)
   (slot-4  :extent :transient)
   (slot-5  :extent :cached)
   (slot-6  :extent :cached)
   (slot-7  :extent :cached-write-through)
   (slot-8  :extent :cached-write-through)
   (slot-9  :extent :persistent)
   (slot-10 :extent :persistent))
  (:metaclass persistent-metaclass))
```

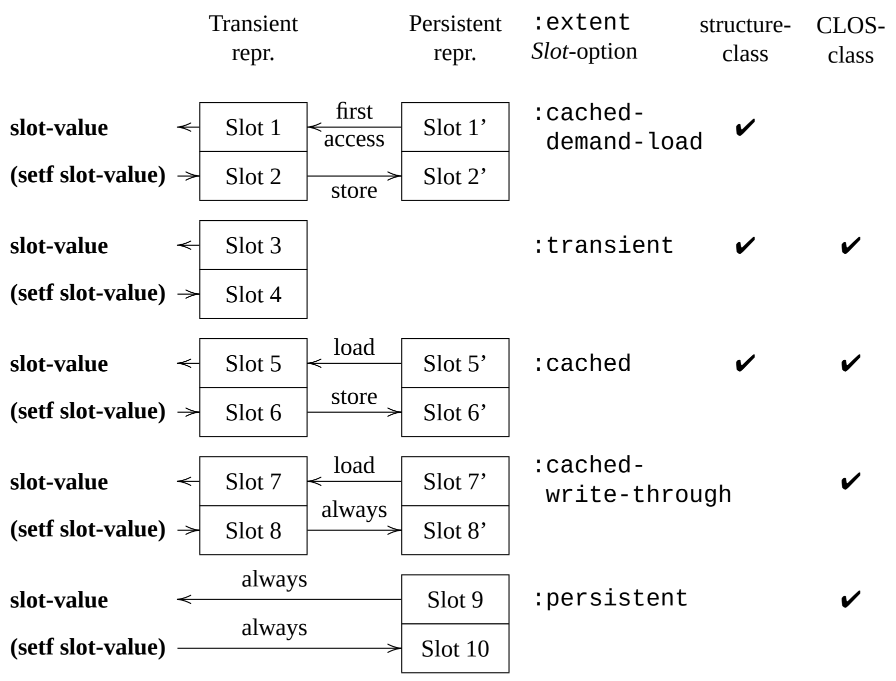


Figure 3.2: Transient object, persistent object and slot extent

If a slot has no slot option `:extent`, the value of class option `:extent` will be used by default.

Slot option `:location`

The slot option `:location` is used to enforce a user-defined ordering for the slots of a class. This is important if persistent instances are to be used as keys for an index or a btree, since the slot ordering represents the significance of the values contained in the object's slots.

The slot option `:location` *tells where a slot is represented.*

Consider for example the following class declaration:

```
CL-USER 1 > (defclass ordered-instance ({<superclasses>})
              ((most-specific-slot :location 10
                                   {<other slot options>})
               (medium-specific-slot :location 20
                                     {<other slot options>})
               (least-specific-slot :location 30
                                    {<other slot options>})
               {<more slots>})
              (:metaclass persistent-metaclass))
```

By default, the slot locations are computed after the slots have been ordered alphabetically by name. Specifying each slot with a `:location` option will enforce that the slots of instances of **ordered-instance** will be arranged in the order as given by the numeric argument passed to the `:location` option. The numbers passed with the `:location` option are used only for changing the default ordering, they are not directly used as slot locations in persistent objects representations. Now, when instances of **ordered-instance** are used as keys for btrees with an `equal` test mode, the instances will be inserted into the btree according to the specified slot ordering, that means, the value of slot **most-specific-slot** will be the primary key, the value of slot **medium-specific-slot** will be the secondary key, and the value of slot **lowest-specific-slot** will be used as ternary key.

References for all additional slot options Class **persistent-metaclass** [API Manual 1997, p. 103], generic function **(setf slot-extent)** [API Manual 1997, p. 126].

3.2.2 Additional class options

The tight binding adds the class options `:constructor`, `:dependent`, `:extent` and `:schema-evolution`.

Within the tight binding, additional class options can be used to steer persistency. Here is an overview of the additional class options:

```
(defclass <class name> ( { <superclasses> } )
  ( { <slots> } )
  (:metaclass persistent-metaclass)
  (:constructor <funcallable object>)   ;; usage deprecated
  (:dependent <boolean>)
  (:extent <keyword>)
  (:schema-evolution <keyword>)
  { <other class options> } )
```

Class option `:constructor` *(usage deprecated)*

The class option `:constructor` declares a constructor for transient representations of a persistent class.

Set the constructor function for instances of the class to the option's argument, which is a function being declared as `(defun` *<constructor>* `(p-objid depth p-heap) ...)`. The constructor is called for creating transient instances of the class being loaded from the persistent heap into transient memory. For persistent CLOS objects, the constructor is called with 3 arguments:

p-objid The *objid* of the object to load from the persistent heap (section 'Identification of objects' on page 18).

depth The depth the object should be loaded to.

p-heap The persistent heap the object should be loaded from.

The constructor is responsible for creating the transient instance and filling the transient instance from its persistent representation. If this class option is not passed, PLOB! will create the transient representation automatically. This class option is meant for instances which can not be created the standard way PLOB! does it; for establishing side effects on storing or loading of objects, use the mechanisms described in section 'Hooking into object storing or loading' on page 37. To make things clear, *usage of this class option is strongly deprecated.*

Class option `:dependent`

The class option `:dependent` tells if instances of a persistent class should be transferred from server to client among with objects referencing those instances.

Declaring a class as being `:dependent` with an option argument of non-`nil` will result in instances of that class to be transferred from the server to the client together with the object referencing such instances. Use this option if you have classes whose instances are more or less 'embedded' into the referencing instance, and are only referenced from this 'embedding' instance. This is the case when it is known in advance that when the client loads the referencing instance, the referenced instance will be loaded in the same transaction in almost all cases, too. For example, consider a class for geometric lines ended by two points.

```
CL-USER 2 > (defclass point ()
              ((x :type float :initarg :x :initform 0.0)
               (y :type float :initarg :y :initform 0.0))
              (:metaclass persistent-metaclass)
              ;; Transfer a point from server to client
              ;; when its referencing object is to be transferred.
              (:dependent t))
```

```
CL-USER 3 > (defclass line ()
              (;; Instances of class point will almost ever be referenced
               ;; by exactly one instance of class line.
               (point-1 :type point :initform nil)
               (point-2 :type point :initform nil))
              (:metaclass persistent-metaclass))
```

If it is known in advance that the end points are always needed when a **line** instance is loaded into transient memory, it would make sense to mark the class **point** as being `:dependent`. A rule of thumb is if the slot of a class (**line** in the example above) is typed by a `:type` slot option (like slots **point-1** and **point-2**), a `:dependent` class option should be given with the class the slot is typed to (**point** in this example).

The `:dependent` flag is placed onto the referenced object, and not on the referencing object. Clever usage of this option will result in much faster access to instances of `:dependent` marked classes being referenced by other instances, since communication overhead is dramatically reduced for such instances. Exhaustive use of this option may result in big, maybe redundant data packages transferred from the server to the client.

A `:dependent` flag can be placed on individual objects, too (see function **(setf p-dependent)** [API Manual 1997, p. 63]). Consider for example a cons cell which should be loaded from persistent memory. If it is known that the object referenced by the cdr of the cons cell should be loaded too, marking the cdr object as being `:dependend` will transfer the cdr object together with the cons cell.

Class option `:extent`

This option's argument will be used as default for the slot extent of each slot of the class. If a slot has an `:extent` slot option specified, the slot option's value takes precendence over the class option's value. The default class extent for classes with tight binding (that means, with a `:metaclass` of `persistent-metaclass`) is specified by the value of parameter ***default-plob-slot-extent*** [Internal Manual 1997, p. 32].

The class option `:extent` declares a default slot extent for a class.

Class option `:schema-evolution`

The value of the `:schema-evolution` class option decides on how transient class redefintions should be promoted to the instances of the class in persistent memory (section 'Schema evolution' on page 62).

The class option `:schema-evolution` tells how a change of a transient class definition should be promoted to its persistent class description-object class.

References for all additional class options Class **persistent-metaclass** [API Manual 1997, p. 103], generic function **(setf class-constructor)** [API Manual 1997, p. 13], generic function **(setf class-dependent)** [API Manual 1997, p. 15], generic function

(setf class-extent) [API Manual 1997, p. 26], generic function **(setf class-dependent)** [API Manual 1997, p. 15], function **(setf p-dependent)** [API Manual 1997, p. 63], generic function **(setf schema-evolution)** [API Manual 1997, p. 113].

3.3 Declaring an index on a slot

The slot otion `:index` *declares an index to be maintained on the slot values.*

For the tight binding CLOS interface, an index can be declared on a slot: The additional slot option `:index` declares an index to be maintained for the slot's values. An index maps the slot value of the slot it is declared for of all instances to the instance having the corresponding slot value. Currently, indexes are represented by persistent btrees; those will be explained in detail in section 'Persistent btrees' on page 52. For the moment, it is sufficient to assume that a btree is like a hash table having its key values arranged always sorted. An index is always a 1:1 mapping from a slot value to the instance having that slot value. For example, this class will maintain an index on its slot named `soc-sec-#`:

```
CL-USER 4 > (defclass person ({<superclasses>})
              ((soc-sec-# :initarg :soc-sec-#
                          :accessor person-soc-sec-#
                          :extent :cached-write-through
                          :index (btree :test equal))
               {<more slots>})
              (:metaclass persistent-metaclass))
```

Since an index has to be unique in its key, either no `:initform` at all should be declared for the slot or the `:initform` must not return the same value for two non-identical instances; it must always return a different value. Indexes are inherited to subclasses.

3.3.1 Inserting objects into an index

Setting a slot's value of an object will insert the object into the index.

Objects will be inserted into the index as soon as the slot for which an index has been declared is set to a value, either by a call to function **(setf slot-value)** [CLtLII 1990, p. 857] or by a call to a writer method defined on the slot. Since PLOB! cares for calling function **(setf slot-value)** [CLtLII 1990, p. 857] when initializing an instance, it is possible to pass the slot value to be maintained in an index as an `:init-arg` to generic function **make-instance** [CLtLII 1990, p. 848].

```
CL-USER 5 > (setf *ps-1* (make-instance 'person :soc-sec-# 1))
;;;;; Stored #<instance class-description person 1.00
;;;;;                  slots=1/2 short-objid=50327387>.
;;;;; Stored #<structure structure-description persistent-btree
;;;;;                   1.00 slots=0/2 short-objid=50326860>.
#<instance person soc-sec-#=1 short-objid=50326730>
CL-USER 6 > (setf *ps-2* (make-instance 'person :soc-sec-# 2))
#<instance person soc-sec-#=2 short-objid=50326732>
```

This example abstracts for now from the problem of referencing the created persistent instance in a later LISP session, where the binding of the transient symbol `*ps-1*` to the persistent object does no longer exist; this problem and its solution is handled in section 'Persistent symbols' on page 49.

Index keys have to be unique.

The requirement of an index being unique in its key will result in an error message if the passed `:soc-sec-#` is not unique in all of the index' current keys.

```
CL-USER 7 > (setf *ps-b* (make-instance 'person :soc-sec-# 2))
Error: Trying to set duplicate slot value 2 for index
```

```
        #<btree equal 2/678 short-objid=50327130> on slot
        #<instance effective-slot-description soc-sec-#
                   extent=cached-write-through
                   short-objid=50327645>
        of object #<instance person soc-sec-#=#<unbound-marker=0x104>
                             short-objid=50326734>;
        object which captures the value already is
        #<instance person soc-sec-#=2 short-objid=50326732>.
```

Changing the slot's value of an instance contained in an index will remove the association from the slot's old value to the instance and will set up an association from the slot's new value to the instance.

```
CL-USER 8 > (setf (slot-value *ps-2* 'soc-sec-#) 3)
3
```

Now, the index key of 2 is not bound any more and can be used for another instance:

```
CL-USER 9 > (setf *ps-b* (make-instance 'person :soc-sec-# 2))
#<instance person soc-sec-#=2 short-objid=50326736>
```

As explained in section 'Persistent btrees' on page 52, it is also possible for an index represented by a btree to hold persistent vectors as keys; this makes it possible to work with primary, secondary, ..., *n*ary keys on slots, where *n* is the (*n* - *1*) th componenent of the persistent vector. Also, this can be used to enforce uniqueness for btree keys by using a vector with an additional, unique-guaranteeing component compared to the non-unique, non-vector key.

3.3.2 Removing objects from an index

Don:	I didn't know you had a cousin Penelope, Bill! Was she pretty?
W. C.:	Well, her face was so wrinkled it looked like seven miles of bad road. She had so many gold teeth, Don, she used to have to sleep with her head in a safe. She died in Bolivia.
Don:	Oh Bill, it must be hard to lose a relative.
W. C.:	It's almost impossible.

— W. C. Fields

An instance will be removed from an index by making the slot's value unbound.

Unbinding the slot of an object will remove the object from the index.

```
CL-USER 10 > (slot-makunbound *ps-1* 'soc-sec-#)
#<instance person soc-sec-#=#<unbound-marker=0x104> short-objid=50326730>
```

Unbounding the slot's value of an instance contained in an index will remove the association from the slot's value to the instance. Except that the slot's state is set to be unbound, the instance itself is not changed or destroyed in any way. Unless there are other references to the instance, it now may disappear with the next garbage collection.

3.3.3 Querying an index

The function **p-select** [API Manual 1997, p. 83] will query for all instances having specific values in a slot with a declared index. It is possible to ask for a single value or for a range of values. For example:

Function **p-select** *[API Manual 1997, p. 83] is used to query an index.*

```
CL-USER 11 > (p-select 'person :where 'soc-sec-# := 2)
#<instance person soc-sec-#=2 short-objid=50326732>
CL-USER 12 > (soc-sec-# (p-select 'person :where 'soc-sec-# := 2))
2
CL-USER 13 > (p-select 'person :where 'soc-sec-# :< 3)
(#<instance person soc-sec-#=2 short-objid=50326732>)
```

An index can also be used to retrieve all objects of a class:

```
CL-USER 14 > (p-select 'person :where 'soc-sec-#)
(#<instance person soc-sec-#=2 short-objid=50326732>
 #<instance person soc-sec-#=3 short-objid=50326734>)
```

At the moment, no SQL- or OQL-like interface is available.

3.3.4 Indexes and reachability

Objects in an index are always reachable and therefore persistent.

An index-representing object has references to all objects it contains. PLOB! implements persistency by reachability: As long as a persistent object is reachable from a persistent root object, it will remain persistent. All persistent classes (represented by class description-objects) known to PLOB! are implicit reachable. Since an index-representing object is bound to the persistent slot description-object holding the slot's description, and this slot description-object is bound to the persistent class description-object, all instances contained in an index-representing object are reachable and therefore persistent.[1] Here is the complete reference chain for the example class **person** given above:

> Persistent root $\longrightarrow$ class table $\longrightarrow$ class description-object of class **person** $\longrightarrow$ slot description-object of slot **soc-sec-#** $\longrightarrow$ index-representing object (a btree) $\overset{*}{\longrightarrow}$ instance of class **person**

So, the only possibility to get rid of an object of a class with a declared index is to remove the object from the index.

References Section 'Persistent btrees' on page 52, function **p-select** [API Manual 1997, p. 83], constant **+plob-slot-write-through-extents+** [Internal Manual 1997, p. 180], function **slot-makunbound** [CLtLII 1990, p. 855].

3.4 Making objects persistent: Loose binding

'Loose binding' means explicite equalization of an object's state between its transient and persistent representation.

The loose binding is established by a store/load interface made up by the two functions **store-object** [API Manual 1997, p. 132] and **load-object** [API Manual 1997, p. 38]. Their purpose is to destructively equalize between the state of an object in persistent memory and the state of its counterpart in transient memory. It lacks some of the features of the tight binding, for example no automatic index administration is possible for instances handled this way. Some other functionality is brought back by the top-level declarations which can be used to influence persistency for instances of loose-bound classes, too, see section 'Top level declarations' on page 38. The loose binding is done by storing an instance by a call to generic function **store-object** [API Manual 1997, p. 132]:

```
CL-USER 15 > (setf *objid* (store-object <object>))
;; To be more specific, for a call like
;; CL-USER 16 > (store-object "A string.")
;; #<simple-string 'A string' short-objid=50327649>
;; is returned.
```

Any LISP object can be stored this way, *<object>* is not limited to be a CLOS instance. The term 'any' is to be understood literally here, the only exception is binary function code (see section 'Binary function code' on page 36). The reason for offering a loose binding for general LISP and CLOS objects is that getting control over the access

[1] In some persistent systems, such an object referencing all objects of a class is called an *allset*.

to the state of non-CLOS and general CLOS instances would mean to overwrite many system-supplied functions working on those objects, with unforeseeable side effects, like system instability, loss of overall system performance etc.

The example call above returned an *objid* which acts as a proxy for the persistent representation of the stored transient object. This proxy can be used in turn to load the object:

```
CL-USER 17 > (load-object *objid*) ;; returns <object>
;; For the string stored above,
;; "A string."
;; is returned.
```

Besides loading the whole persistent representation into a transient object, the *objid* (proxy) can be used to work directly on the persistent representation; this is explained in section 'Direct Interface' on page 28.

This call will return the object currently stored. As mentioned already in an example above, this one abstracts too from the problem of referencing the created persistent instance in a later LISP session, where the binding of the transient symbol `*objid*` to the *objid* does no longer exist; this problem and its solution is handled in section 'Persistent symbols' on page 49.

Though it is also possible to pass the numeric *objid* as shown in the output behind the **store-object** call directly to function **load-object** [API Manual 1997, p. 38] for loading the object, this is no good way to reference a persistent object. This numeric *objid* is temporary, since it will change its value with the next garbage collection on the persistent heap; using persistent symbols (section 'Persistent symbols' on page 49) for addressing is safe.

References Generic function **store-object** [API Manual 1997, p. 132], function **load-object** [API Manual 1997, p. 38].

3.4.1 Setting the depth of storing or loading an object

Both functions **store-object** [API Manual 1997, p. 132] and **load-object** [API Manual 1997, p. 38] have a *depth* argument telling how 'deep' the object's state should be transferred between transient and persistent memory. The *depth* argument is an optional argument with its default being the value of parameter ***default-depth*** [Internal Manual 1997, p. 31].

Specifying a *depth* of `:cached`

A depth of `:cached` will transfer an object's state and all objects referenced in its slots if and only if the objects are not found in the cache. Some calls do an object storing and implicit caching. For example, if an instance of a tight bounded persistent CLOS class is created by a call to generic function **make-instance** [CLtLII 1990, p. 848] and at least one slot of the class has an `:initform`, the created object will be stored and put into the cache. Depending on the system configuration, strings and numbers might not be cached at all (section 'Identity and caching' on page 19) and so will always be stored with a depth of `:cached`.

Specifying a *depth* of `:flat`

A depth of `:flat` will transfer always the top-level of an object's state. The objects referenced in its slots will be transferred if and only if they are not found in the cache.

Specifying a *depth* of `:deep`

A depth of `:deep` will transfer always the top-level of an object's state and all the objects referenced in its slots.

Specifying a *depth* of `:objid` or `:object`

This will result in a do-nothing in both functions, since this depth indicates that the caller is working directly on the persistent representation of the object passed, so no transfer is necessary.

3.4.2 Tight vs. loose binding: Benefits and drawbacks

The benefits of using a tight binding is a very high degree of transparency for persistency: Besides knowing of the above mentioned additional slot and class options, the user of the tight-binding interface can use persistent objects with few knowledge of what is happening back-stage in PLOB. Also, the tight binding offers more functionality than the loose binding interface, for example the automatic administration of indexes. The drawback is a possible loss in overall performance on instances of classes with tight binding to persistency, since the tight binding is very generic. Also, many LISP systems (both LISPWORKS Common LISP and ALLEGRO Common LISP belonging to them) do heavily optimizations to slot accesses of instances whose class is a direct instance of the class metaobject class **standard-class**; this advantage is lost for exactly the classes using the tight binding.

A further benefit on using the loose binding besides keeping the optimized slot access is that instances whose class declaration is unknown or out of scope can be stored and loaded, too. A drawback is that some functionality is lost w.r.t. the tight binding interface.

3.5 Working directly on persistent objects

This section explains how to work directly on the persistent representations of persistent objects. This way of working is sometimes needed for efficiently establishing side effects.[2] For example, when working destructively on lists which should be kept persistent, there are at least two methods how to do this:

1. Do destructive changes to the transient representation of the list and store the list anew (for example, by a call to generic function **store-object** [API Manual 1997, p. 132]) after all modifications have been done.

2. Do destructive changes directly to the persistent list and, if necessary, load the list anew after all modifications have been done (for example, by a call to function **load-object** [API Manual 1997, p. 38]) .

Solution 1 has the benefit of being simple to code, but will store not only the list itself anew, but eventually all objects referenced by the list. PLOB! does not distinguish on a per-class basis between objects to be stored and objects not to be stored during a store operation. Instead, all objects referenced by the object to store are written to the persistent heap.

By working directly on persistent objects, solution 2 does not suffer from this problem, but another application interface has to be used. At a first glance, this solution sounds similar to the first one the other way round. This worked in practice when building up a database in an offline phase (*not* [re]loading the [modified] persistent objects)

[2]This kind of working would be completely unnecessary when designing a persistent LISP system from scratch on, but this was beyond the scope of this work.

and using it (more or less read-only) in an online phase. The lists have been referenced by slots of persistent CLOS instances representing nodes of a graph. In that application, the lists have not been loaded explicit by calls to function **load-object** [API Manual 1997, p. 38], but when a persistent CLOS instance was loaded on-demand and its slot value containing the list has been accessed.

3.5.1 Immediates

An immediate is an object whose representation of state captures fewer bits than an *objid* identifying the object, so it is much more efficient to represent the immediate object's state directly in a (type marked) *objid*. Table 3.3 shows all types whose in-

Type	Width [bits]	See also
character	8	
bitmask	24+5	reserved, but not yet implemented
short-float	28	
fixnum	30	
marker	29	section 'Markers' on page 51

Figure 3.3: Immediate represented types and their data width in bits

stances are represented as immediates in PLOB. Therefore, instances of these types have no functions for direct access. True **short-float**s are only represented in LISPWORKS Common LISP, in ALLEGRO Common LISP they are always coerced to type **single-float**.

3.5.2 Establishing a context for direct working

For establishing a context for direct working, there are the following two posibilities.

Macro with-direct-representation

Macro **with-direct-representation** [API Manual 1997, p. 134] establishes a context on its *forms* argument where direct representation is used for all following direct interface functions. Since all what macro **with-direct-representation** [API Manual 1997, p. 134] does is binding a special variable to a new value, the scope of this context is dymnamic and not lexical. Consider for example:

```
CL-USER 18 > (p-cons "The CAR" "The CDR")                                    ①
("The CAR" . "The CDR")
CL-USER 19 > (with-direct-representation () (p-cons "The CAR" "The CDR"))②
#<cons (#<simple-string 'The CAR' short-objid=50327142> .
        #<simple-string 'The CDR' short-objid=50327141>)
        short-objid=50327160>
```

The first example call ① returns a transient representation for the created persistent cons cell, whereas the second call ② returns an object referencing the persistent cons cell. In ②, no transient cons cell is created at all.

Macro **with-transient-representation** [API Manual 1997, p. 136] can be used to switch to a context where transient representations are generated for persistent objects; the transient representation is the default representation selected by PLOB.

Overwrite optional *depth* argument

Many of the direct interface functions have an optional *depth* argument. When this optional argument is set to `:object`, the function will work directly on a persistent representation. Consider for example:

②
```
CL-USER 20 > (p-cons "The CAR" "The CDR" :object)
#<cons (#<simple-string 'The CAR' short-objid=50327145> .
        #<simple-string 'The CDR' short-objid=50327144>)
        short-objid=50327160>
```

[See the explanations given with the example code before.]

3.5.3 Common issues for the following sections

In the following sections, many classes have a function **p-allocate-**<*class name*>. These functions allocate an instance of the class named <*class name*> in the persistent heap and initialize it to some specific state (normally, all slots are set to be unbound). In the following sections, they are marked by an **Allocator** label.

The functions named **p-make-**<*class name*> allocate an instance of the class named <*class name*> in the persistent heap and write the state of its *t-from* argument into the fresh allocated persistent object. In the following sections, they are marked by a **Creator** label.

The functions named **p-**<*class name*>**p** or **p-**<*class name*>**-p** are predicates taking an *objid* as argument. They check if the persistent object referenced by the *objid* is an instance of the class named <*class name*>. In the following sections, those functions are maked by a **Type Predicate** label.

With the direct interface, it is possible to work either on single slots of a persistent object or on the state of a persistent object as a whole.

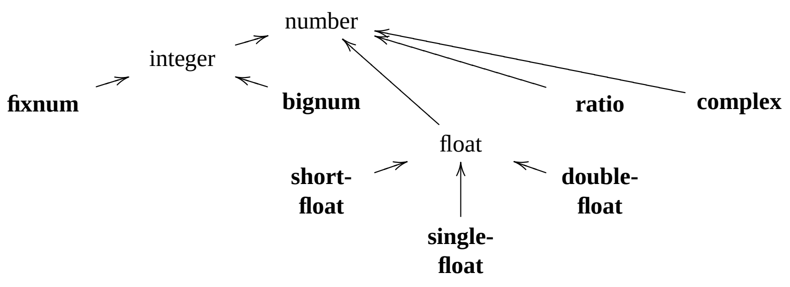
The functions working on single slots are named similar to the ones defined in [CLtLII 1990] with a prefix of **p-**, for example, function **p-car** [API Manual 1997, p. 57] works on the car of a persistent cons cell. In the following sections, those functions are marked marked by a **Slot Accessor** label in general or by a **Slot Reader** or **Slot Writer** label if appropiate.

All functions whose name is matching **(setf p-**<*class name*>**)** store a transient instance of the class named <*class name*> to the persistent heap as a whole. A persistent instance is allocated, if appropiate, and its state is set to the state of the transient object passed as the *new-value* function argument. In the following sections, those functions are marked by an **Object Store** label.

All functions whose name is matching **p-**<*class name*> load a persistent instance of the class named <*class name*> from the persistent heap. A transient instance is allocated, if appropiate, and its state is set to the state of the persistent object referenced by the passed *objid* function argument. In the following sections, those functions are marked by an **Object Load** label.

Some persistent classes have functions returning some kind of information about its instances. In the following sections, those functions are marked by an **Information** label.

References All functions named **p-allocate-**<*class name*>; all functions named **p-**<*class name*>**p** or **p-**<*class name*>**-p**; all **p-**<***fn***> functions with <*fn*> being one function defined in [CLtLII 1990]; all functions named **(setf p-**<*class name*>**)** and **p-**<*class name*>.


number
integer
fixnum
bignum
ratio
complex
float
short-float
single-float
double-float

3.5.4 Working directly on persistent numbers

Figure 3.4 shows all numeric classes implemented as built in types in PLOB. Not all of these classes might be available in the actual LISP system used, for example, none of the ALLEGRO Common LISP versions and only pre-4.0.1 versions of LISPWORKS Common LISP do support short floats. LISPWORKS Common LISP 4.0.1 does not support single floats, too; all floats are represented by double floats. Class fixnum is the class of all 30 bit wide persistent fixnums; transient integers of the actual LISP system used will be transformed into persistent 30 bit fixnums, if possible.

References for type bignum

Allocator	Function **p-allocate-bignum** [API Manual 1997, p. 46]
Creator	Function **p-make-bignum** [API Manual 1997, p. 75]
Type Predicate	Function **p-bignum-p** [API Manual 1997, p. 55]
Object Store	Function **(setf p-bignum)** [API Manual 1997, p. 55]
Object Load	Function **p-bignum** [API Manual 1997, p. 55]

References for type single float

Allocator	Function **p-allocate-single-float** [API Manual 1997, p. 48]
Creator	Function **p-make-single-float** [API Manual 1997, p. 77]
Type Predicate	Function **p-single-float-p** [API Manual 1997, p. 85]
Object Store	Function **(setf p-single-float)** [API Manual 1997, p. 85]
Object Load	Function **p-single-float** [API Manual 1997, p. 85]

References for type double float

Allocator	Function **p-allocate-double-float** [API Manual 1997, p. 47]
Creator	Function **p-make-double-float** [API Manual 1997, p. 75]
Type Predicate	Function **p-double-float-p** [API Manual 1997, p. 65]
Object Store	Function **(setf p-double-float)** [API Manual 1997, p. 64]
Object Load	Function **p-double-float** [API Manual 1997, p. 64]

References for type ratio

Allocator	Function **p-allocate-ratio** [API Manual 1997, p. 48]
Type Predicate	Function **p-ratiop** [API Manual 1997, p. 81]
Object Store	Function **(setf p-ratio)** [API Manual 1997, p. 80]
Object Load	Function **p-ratio** [API Manual 1997, p. 80]
Slot Reader	Function **p-numerator** [API Manual 1997, p. 77], function **p-denominator** [API Manual 1997, p. 63]

References for type complex

Allocator	Function **p-allocate-complex** [API Manual 1997, p. 46]
Type Predicate	Function **p-complexp** [API Manual 1997, p. 61]
Object Store	Function **(setf p-complex)** [API Manual 1997, p. 60]
Object Load	Function **p-complex** [API Manual 1997, p. 60]
Slot Reader	Function **p-realpart** [API Manual 1997, p. 82], function **p-imagpart** [API Manual 1997, p. 71]

3.5.5 Working directly on persistent lists

Lists are handled as sequences, that means, a load or store operation will be done for the cdr of each cons cell, too.

References for type cons

Allocator	Function **p-allocate-cons** [API Manual 1997, p. 47]
Creator	Function **p-cons** [API Manual 1997, p. 61]
Type Predicate	Function **p-consp** [API Manual 1997, p. 61]
Object Store	Function **(setf p-list)** [API Manual 1997, p. 74]
Object Load	Function **p-list** [API Manual 1997, p. 73]
Slot Reader	Function **p-car** [API Manual 1997, p. 57], function **p-cdr** [API Manual 1997, p. 58]
Slot Writer	Function **(setf p-car)** [API Manual 1997, p. 57], function **(setf p-cdr)** [API Manual 1997, p. 58]

Addtionally, function **p-null** [API Manual 1997, p. 77] is a predicate which checks for its argument being either a transient or persistent `nil`.

3.5.6 Working directly on persistent arrays

If possible, specialized arrays, that means, arrays with an `:element-type` $\neq$ **t**, are stored with the element type they are specialized to:

```
CL-USER 21 > (store-object (make-array '(2 3)
                                       :element-type '(unsigned-byte 8)
                                       :initial-element 0))
#<array (unsigned-byte 8) (2 3) short-objid=50327320>
```

This is only done if and only if LISPWORKS Common LISP or ALLEGRO Common LISP do not downcast the element type to type **t**; no downcast to type **t** is induced by PLOB! itself on array elements:

```
CL-USER 22 > (array-element-type (make-array '(2 3)
                                             :element-type 'float
                                             :initial-element 0.0))
T
```

In this case, a downcast to type **t** is done by ALLEGRO Common LISP, so the array elements will also be stored as non-specialized instances.

References for type array

Allocator	Function **p-allocate-array** [API Manual 1997, p. 46]
Type Predicate	Function **p-arrayp** [API Manual 1997, p. 54]
Object Store	Function **(setf p-array)** [API Manual 1997, p. 52]
Object Load	Function **p-array** [API Manual 1997, p. 52]
Slot Accessor	Function **p-array-data-vector** [API Manual 1997, p. 53] (direct usage deprecated)
Information	Function **(setf p-array-fill-pointer)** [API Manual 1997, p. 54], function **p-array-fill-pointer** [API Manual 1997, p. 54], function **p-array-displaced-offset** [API Manual 1997, p. 53], function **p-array-adjustable** [API Manual 1997, p. 52], function **p-array-rank** [API Manual 1997, p. 54], function **p-array-dimensions** [API Manual 1997, p. 53], function **p-array-element-type** [API Manual 1997, p. 53]

For the moment, there is no **p-aref** function for direct access to persistent array elements. The workaround is that a persistent array must be stored or loaded as a whole for accessing its elements.

3.5.7 Working directly on persistent vectors

The following functions work on simple vectors, that means, vectors without a fill pointer and their element type specialized to type **t**. Non simple vectors are handled as 1-dimensional arrays.

References for type vector

Allocator	Function	**p-allocate-vector** [API Manual 1997, p. 50]
Type Predicate	Function	**p-vectorp** [API Manual 1997, p. 101]
Object Store	Function	**(setf p-vector)** [API Manual 1997, p. 100]
Object Load	Function	**p-vector** [API Manual 1997, p. 100]
Slot Reader	Function	**p-svref** [API Manual 1997, p. 96]
Slot Writer	Function	**(setf p-svref)** [API Manual 1997, p. 96]
Information	Function	**p-vector-length** [API Manual 1997, p. 101]

3.5.8 Working directly on persistent bit vectors

References for type bit-vector

Allocator	Function	**p-allocate-bit-vector** [API Manual 1997, p. 46]
Type Predicate	Function	**p-bit-vector-p** [API Manual 1997, p. 57]
Object Store	Function	**(setf p-bit-vector)** [API Manual 1997, p. 56]
Object Load	Function	**p-bit-vector** [API Manual 1997, p. 55]
Information	Function	**p-bit-vector-length** [API Manual 1997, p. 56], function **p-bit-vector-max-length** [API Manual 1997, p. 56]

Because a bit vector has single bit elements and all allocations are done to word boundaries, there are `(- (p-bit-vector-max-length <`*`bit-vector`*`>) (p-bit-vector-length <`*`bit-vector`*`>))` overhead bits allocated with each bit vector.

For the moment, there is no **p-aref** function for direct access to persistent bit vector elements. The workaround is that a persistent bit vector must be stored or loaded as a whole for accessing its elements. In the future, persistent bit vectors with their number of elements ≤ 24 may be represented as immediates.

3.5.9 Working directly on persistent strings

References for type string

Allocator	Function	**p-allocate-string** [API Manual 1997, p. 49]
Type Predicate	Function	**p-stringp** [API Manual 1997, p. 89]
Object Store	Function	**(setf p-string)** [API Manual 1997, p. 88]
Object Load	Function	**p-string** [API Manual 1997, p. 87]
Information	Function	**p-string-length** [API Manual 1997, p. 88], function **p-string-max-length** [API Manual 1997, p. 88]

Because a string has single byte elements and all allocations are done to word boundaries, there are `(- (p-string-max-length <`*`string`*`>) (p-string-length <`*`string`*`>))` overhead bytes allocated with each string.

Meta bits and the like are not supported for characters contained in a persistent string.

In the future, persistent strings with their number of characters ≤ 3 may be represented as immediates.

3.5.10 Working directly on persistent functions

References for type function

Allocator	Function **p-allocate-function** [API Manual 1997, p. 47]
Type Predicate	Function **p-functionp** [API Manual 1997, p. 70]
Object Store	Function **(setf p-function)** [API Manual 1997, p. 68]
Object Load	Function **p-function** [API Manual 1997, p. 67]
Slot Reader	Function **p-function-code** [API Manual 1997, p. 68], function **p-function-language** [API Manual 1997, p. 69], function **p-function-name** [API Manual 1997, p. 69]
Slot Writer	Function **(setf p-function-code)** [API Manual 1997, p. 68], function **(setf p-function-language)** [API Manual 1997, p. 69], function **(setf p-function-name)** [API Manual 1997, p. 69]

Direct storing and loading of binary function code is not supported, see section ‘Binary function code’ on page 36 for details and the implemented solution.

3.5.11 Working directly on persistent hash tables

In many LISP systems, hash tables are stored internally as instances of a `defstruct` class, so PLOB! would be able to store hash table instances without any special intervention. Since the names choosen for the class whose instances represent hash tables and its slots vary among the LISP system implementators, and for efficiency reasons, hash tables are explicit represented in persistent memory by instances of class **persistent-hash-table** [Internal Manual 1997, p. 171].

References for type hash-table

Object Store	Function **(setf p-hash-table)** [API Manual 1997, p. 70]
Object Load	Function **p-hash-table** [API Manual 1997, p. 70]

Using a persistent btree could be more suitable than using a persistent hash table (section ‘Persistent btrees’ on page 52, and section ‘Consequences for data modeling’ on page 41).

3.5.12 Working directly on persistent CLOS instances

The direct way of working on persistent structures and CLOS instances is business as usual, this can be done by calling the functions **(setf slot-value)** [CLtLII 1990, p. 857], **slot-value** [CLtLII 1990, p. 857], **slot-boundp** [CLtLII 1990, p. 855] and **slot-makunbound** [CLtLII 1990, p. 855]. In ALLEGRO Common LISP, there is a problem with those functions working directly on persistent objects: They expect their *object* argument to be an instance of a CLOS class effectively having the slot whose *slot-name* was passed; this is not the case for the direct access interface, since this interface works on *references* to persistent objects but not on true CLOS objects themselves. As a workaround, when using the direct interface on CLOS instances in ALLEGRO Common LISP, the functions **(setf p-slot-value)** [API Manual 1997, p. 87], **p-slot-value** [API Manual 1997, p. 86], **p-slot-boundp** [API Manual 1997, p. 86] and **p-slot-makunbound** [API Manual 1997, p. 86] must be used. *This workaround has only to be used under these described circumstances.*

slot-value *[CLtLII 1990, p. 857] and companions work on both structure and CLOS objects.*

Allocator	Function **p-allocate-instance** [Internal Manual 1997, p. 92]. Direct usage is deprecated; instead, the proceeding described at **Creator** should be used for creating objects.

Creator	Generic function **make-instance** [CLtLII 1990, p. 848] should be called, and, if necessary, the instance should be stored by a call to generic function **store-object** [API Manual 1997, p. 132].
Type Predicate	Function **p-instancep** [Internal Manual 1997, p. 117]
Slot Reader	For LISPWORKS Common LISP: function **slot-value** [CLtLII 1990, p. 857]. For ALLEGRO Common LISP: function **p-slot-value** [API Manual 1997, p. 86]
Slot Writer	For LISPWORKS Common LISP: Function **(setf slot-value)** [CLtLII 1990, p. 857], function **slot-makunbound** [CLtLII 1990, p. 855]. For ALLEGRO Common LISP: Function **(setf p-slot-value)** [API Manual 1997, p. 87], function **p-slot-makunbound** [API Manual 1997, p. 86]
Information	For LISPWORKS Common LISP: Function **slot-boundp** [CLtLII 1990, p. 855]. For ALLEGRO Common LISP: Function **p-slot-boundp** [API Manual 1997, p. 86]

3.5.13 Working directly on persistent structure instances

Allocator	Function **p-allocate-structure** [API Manual 1997, p. 49]. Direct usage is deprecated; instead, the proceeding described at **Creator** should be used for creating objects.
Creator	The system generated structure constructor should be called and the object should be stored by a call to generic function **store-object** [API Manual 1997, p. 132].
Type Predicate	Function **p-structurep** [API Manual 1997, p. 95]
Slot Reader	For LISPWORKS Common LISP: function **slot-value** [CLtLII 1990, p. 857]. For ALLEGRO Common LISP: function **p-slot-value** [API Manual 1997, p. 86]
Slot Writer	For LISPWORKS Common LISP: Function **(setf slot-value)** [CLtLII 1990, p. 857], function **slot-makunbound** [CLtLII 1990, p. 855]. For ALLEGRO Common LISP: Function **(setf p-slot-value)** [API Manual 1997, p. 87], function **p-slot-makunbound** [API Manual 1997, p. 86]
Information	For LISPWORKS Common LISP: Function **slot-boundp** [CLtLII 1990, p. 855]. For ALLEGRO Common LISP: Function **p-slot-boundp** [API Manual 1997, p. 86]

References Method **(setf slot-value-using-class) (t structure-class persistent-object symbol)** [API Manual 1997, p. 131], item **slot-value-using-class (structure-class persistent-object t)** [??], method **slot-boundp-using-class (structure-class persistent-object t)** [API Manual 1997, p. 117], method **slot-makunbound-using-class (structure-class persistent-object symbol)** [API Manual 1997, p. 129].

3.6 Binary function code

Binary function code can not be stored directly.

Binary function code is the only exception to the orthogonal persistency principle obeyed elsewhere in PLOB. Because of the expected relocation problems and highly system dependent memory layout without a lacking norm between different LISP systems, storing and loading of binary function code is not supported. Because no binary function code can be stored, no methods are stored too. Future versions of PLOB!

may support direct storing of binary function code. Currently implemented are two workarounds for this situation.

References Class **persistent-s-expression** [API Manual 1997, p. 107].

3.6.1 Storing functions by name

Functions are stored by name, if the name can be retrieved.

By the standard set in PLOB!, 'storing' of a function code object is done by trying to retrieve the function's symbolic name and marking the persistent symbol as being fbound'ed; 'loading' means just to look for this symbolic name in the current LISP image and return the function object bound to it.

```
CL-USER 23 > (store-object #'car)
#<function CAR short-objid=50326928>
CL-USER 24 > (setf #!*persistent-car* *)
#<function CAR short-objid=50326928>
CL-USER 25 > #!*persistent-car*
#<Function CAR>
CL-USER 26 > (funcall #!*persistent-car* '(1 2))
1
```

If the symbol found at load time has no function binding, a dummy function object is generated which signals an error when being called. This way functions found in the LISP image can be 'stored', and, for standard functions as defined in [CLtLII 1990], even in a system-independent manner. Persistent function objects are explicitely represented in PLOB.

3.6.2 Persistent s-expressions

Given access to a function's source code, it can be stored as a persistent s-expression.

PLOB! has a class **persistent-s-expression** [API Manual 1997, p. 107], which copes with persistent s-expressions. As a special case, when the s-expression has a `car` of `defun`, the s-expression is handled as a function definition. Usage of s-expressions requires access to a function's λ expression's source code. Considering functions and λ expressions generated at application runtime, this surely is not a very strong restriction. Since class **persistent-s-expression** [API Manual 1997, p. 107] is a straight-forward defined persistent CLOS class, persistent s-expressions are represented explicitely, too.

```
CL-USER 27 > (setf #!*code*
               (make-instance 'persistent-s-expression
                              :source '(lambda (x)
                                         (format t "X is ~A~%" x)
                                         x)))
#<instance persistent-s-expression short-objid=50326545>
CL-USER 28 > (funcall (sexpr-code #!*code*) 1)
X is 1
1
```

The expression given with the `:source` initarg has to be a λ list and *not* a function-quoted expression (`#'(...)`), since function quotes might be compiled into binary function code; this binary function code cannot be stored by PLOB.

3.7 Hooking into object storing or loading

For objects of some classes, it might be necessary to have side effects coupled to store and load operations on the object. This section describes how objects are notified if they are about to be stored or if they have been loaded from a persistent heap.

3.7.1 Object storing

Application interface is still insufficient, for now needs methods specialized to generic function **t-object-to-p-objid** [Internal Manual 1997, p. 253]. Introduce generic function `passivate`?

3.7.2 Object loading

Application interface is still insufficient, for now needs methods specialized to generic function **initialize-instance** [CLtLII 1990, p. 847] and looking for `&key` argument `:objid`. Introduce generic function `activate`? Problem: Mangled handling of creating and initializing a CLOS instance in generic function **make-instance** [CLtLII 1990, p. 848].

3.8 Top level declarations

Top level declarations as a substitute for missing slot and class options.

Instances of built in classes, structure classes and CLOS classes with their definition being not available at all can not use the tight binding described in section 'Tight binding' on page 19. For built in and structure classes, there is no principal way of achieving a tight binding. For CLOS classes, this would require a change to the original class definition (that means, adding a class option `:metaclass persistent-metaclass`); this is possible in principle, but doing so to system defined classes would be a rather dangerous attempt, since a loss in overall system performance or stability could be the result. So, no additional class and slot options can be given to those classes within their definition. To bypass this situation, top level declarations can be used to give PLOB! the hints necessary for handling persistency for those classes. When PLOB! encounters a class not being made persistent up to that moment, the top level declarations for the class are evaluated and stored within the persistent class description-object. This means that the top level declarations for a class must be given *before* the first instance of the class is stored, since this will store the class description-object, too. As with the additional slot (section 'Additional slot options' on page 19) and class options (section 'Additional class options' on page 22), these declarations determine a class' or a slot's extent (section 'Slot option `:extent`' on page 20, and section 'Class option `:extent`' on page 23, its constructor (section 'Class option `:constructor`' on page 22), dependent flag (section 'Class option `:dependent`' on page 22), and its schema evolution (section 'Class option `:schema-evolution`' on page 23). For other additional slot and class options not mentioned here there are no corresponding top level declarations, since their functionality can not be achieved for non tight bound classes.

3.8.1 Declaring a slot's extent

The extent declaration is the most important one and is evaluated on following levels, going from the least to the most significant level. If a declaration is found on a more siginificant level, it will overwrite a declaration from the less significant level.

Default value If none of the following levels package, class or slot specifies an extent, a class specific default value is used. For structure classes, this is the value of parameter ***default-structure-slot-extent*** [Internal Manual 1997, p. 32]. For CLOS classes *not* using the tight binding, this is the value of parameter ***default-clos-slot-extent*** [Internal Manual 1997, p. 31]. For CLOS classes using the tight binding, this is the value of parameter ***default-plob-slot-extent*** [Internal Manual 1997, p. 32].

Package level An extent can be declared on package level by calling generic function **(setf package-extent)** [API Manual 1997, p. 101], for example:

```
(setf (package-extent (find-package <package name>)) :transient)
```

This extent is used for all classes whose naming symbol is in the package the declaration was made for; in the above example, the slots of all classes defined in package <*package name*> belonging to that package would become transient by default.

Class level An extent can be declared on class level by calling generic function **(setf class-extent)** [API Manual 1997, p. 26]. For example, to make all slots of the class named <*class name*> transient:

```
(setf (class-extent (find-class <class name>)) :transient)
```

Slot level An extent can be declared for a slot in a class by calling generic function **(setf slot-extent)** [API Manual 1997, p. 126]. For example, to make the slot named <*slot name*> of the class named <*class name*> transient:

```
(setf (slot-extent <slot name> (find-class <class name>)) :transient)
```

References Section 'Slot option `:extent`' on page 20; parameter ***default-structure-slot-extent*** [Internal Manual 1997, p. 32], parameter ***default-clos-slot-extent*** [Internal Manual 1997, p. 31], parameter ***default-plob-slot-extent*** [Internal Manual 1997, p. 32]; generic function **(setf package-extent)** [API Manual 1997, p. 101], generic function **(setf class-extent)** [API Manual 1997, p. 26], generic function **(setf slot-extent)** [API Manual 1997, p. 126].

3.8.2 Declaring a class' constructor

A class' constructor can be declared for a class by calling generic function **(setf class-constructor)** [API Manual 1997, p. 13], for example:

```
(setf (class-constructor (find-class <class name>)) <funcallable object>)
```

References Section 'Class option `:constructor`' on page 22; generic function **(setf class-constructor)** [API Manual 1997, p. 13].

3.8.3 Declaring a class' dependent flag

A class' dependent flag can be declared for a class by calling generic function **(setf class-dependent)** [API Manual 1997, p. 15], for example:

```
(setf (class-dependent (find-class <class name>)) <boolean>)
```

References Section 'Class option `:dependent`' on page 22; generic function **(setf class-dependent)** [API Manual 1997, p. 15].

3.8.4 Declaring a class' schema evolution

A class' schema evolution can be declared for a class by calling generic function **(setf schema-evolution)** [API Manual 1997, p. 113], for example:

```
(setf (schema-evolution (find-class <class name>)) <keyword>)
```

References Section 'Class option `:schema-evolution`' on page 23; generic function **(setf schema-evolution)** [API Manual 1997, p. 113].

3.8.5 Making objects read-only

Marking objects as read-only will raise an error on requesting a write lock onto the object or onto one of its slots:

```
   CL-USER 29 > (setf #!*precious* "A very precious string.")
   "A very precious string."
①  CL-USER 30 > (setf (p-read-only (p-intern '*precious*)) t)
   T
②  CL-USER 31 > (setf #!*precious* "A not-so-precious string.")
   ;;; Info from server at executing client:fnSHwriteIndex:
   ;;; Waiting for lock level vector, mode wi on object
   ;;; #<symbol common-lisp-user::*precious* short-objid=50328919> by
   ;;; #<heap kirschke@localhost 'Initial Lisp Listener'
   ;;;       transaction=2466 short-objid=50327650>.
   Error: Locking read-only object #<symbol common-lisp-user::*precious*
          short-objid=50328919> by #<heap kirschke@localhost 'Initial
          Lisp Listener' transaction=2466 short-objid=50327650> failed.
     [condition type: SIMPLE-ERROR]
   Restart actions (select using :continue):
    0: Remove all other locks brute-force from object & retry locking.
    1: Retry to lock the object.
③  [1] CL-USER 32 > :pop
④  CL-USER 33 > (setf (p-read-only (p-intern '*precious*)) nil)
   NIL
⑤  CL-USER 34 > (setf #!*precious* "A not-so-precious string.")
   "A not-so-precious string."
```

The persistent symbol is marked as read-only ①. The attempt to set the persistent symbol to a new value raises a continuable error ②. Here, the error is aborted ③; choosing continue instead of aborting would remove the read-only lock and would set the symbol to the new value. After the read-ony lock has been removed ④, the symbol can be set to a new value ⑤.

References Function **(setf p-read-only)** [API Manual 1997, p. 81], function **p-read-only** [API Manual 1997, p. 81].

3.9 Programming with persistent data

Time is not the main point. Time is the one and only point.

— Miles Davis

This section contains some hints on how to program with using persistent objects. Persistency in PLOB! is rather transparent, and this is paid with a certain performance penalty compared to a pure transient system. By doing careful data modeling and coding this penalty can be kept to a minimum.

3.9.1 Overhead associated with an object transfer

Because the memory overhead with an object's transfer is rather low, it is neglected here; considered is only the time needed for an object's state transfer between client and server.

The total overhead $O_{total}(slots, mode)$ associated with an object transfer can be splitted into the communication overhead $O_{comm}(slots)$ needed by TCP/IP for trans-

TCP/IP is the transfer protocol used by the operating system for transfer of data.

ferring an object's state between client and server and the overhead $O_{server}(slots, mode)$ needed by the server for working on an object.

$$\begin{array}{rcl} O_{total}(slots, mode) & = & O_{comm}(slots) + O_{server}(slots, mode) \\ slots & \equiv & \text{number of object slots} \\ mode & \in & \{\texttt{read}, \texttt{write}\} \end{array}$$

The server overhead can be splitted into O_{lock} needed for locking the object and $O_{trans}(slots, mode)$ for saving the object's lock and its state into the transaction log file.

The term slots is valid here also for objects of built in classes; for example, a cons cell has 2 slots, a symbol 5 slots etc.

$$O_{server}(slots, mode) \quad = \quad O_{lock} + O_{trans}(slots, mode)$$

These overheads can be modeled by:

$$\begin{array}{rcl} O_{comm}(slots) & \sim & \lceil \frac{slots}{blocksize} \rceil \\ blocksize & \equiv & \text{TCP/IP block size of appr. 1500 bytes = 375 } slots \\ O_{lock} & = & constant \\ O_{trans}(slots, mode) & \sim & \begin{cases} constant & \text{if } mode = \texttt{read} \\ slots & \text{if } mode = \texttt{write} \end{cases} \end{array}$$

For the most common cases, that means, for objects with fewer than 375 slots, the following inequality holds:

$$O_{lock} \ll O_{trans} \ll O_{comm}$$

The communication overhead O_{comm} will contribute at most to the total overhead O_{total}, and so should be kept to a minimum.

3.9.2 Consequences for data modeling

Because $O_{comm}(slots)$ is proportional to the ceiling function on the number of slots, the overhead up to a total of around 375 slots will always be the same. A consequence which can be drawn from this is that sequences represented by vectors should be used instead of general lists whereever possible.

Also, when needing associative access to data, it should be thought of using a btree instead of a hash table, since the access to btrees can be done on a per-element basis, whereas persistent hash tables are always transferred as a whole.

3.9.3 Consequences for transaction handling

The transaction handling incures much more overhead on the server than on the client's side. The server will write at least the lock state of each object into a transaction log file; if a write lock is placed onto an object, the object's state will be written to the transaction log file, too. So, the consequences for transaction handling in the client's code are not so stringent as the consequences with the data modeling: Transactions should not last too long, since this might overflow the server's disk file space.

3.9.4 Storing or loading lots of data

PLOB! has implemented fine-granularity locking and transactions, to ensure a high level of isolation on database operations. For this feature, a performance price is paid, which does not shine up very stringent in doing online work with the database, but when lots of data should be written to or read from the database. One of these cases is the initial populating of a database, that means when a database should be filled with objects drawn from other, external sources. The proposal is to do these operations offline from the application's point of view, in an extra process reserving the whole database for the operation as a whole. This is no performance trick to bypass the resource-intensive locking; the proposal's intent is to lock the whole database on the coarsest level instead of locking each object seperately, which will gain a performance improvement of about a magnitude.[3]

To cope with circular references, PLOB! uses the object-level locking during a transaction for store or load operations to check if an object has already been stored or loaded within that transaction; so, when abonding object-level locking in favor of locking the whole store, PLOB! must be instructed to use another algorithm to look for already stored or loaded objects. This can be done by telling PLOB! not to store or load objects found in its internal cache. To be more concrete, here are two code examples.

Populating a database

Here, the term 'populating' means reading objects from an external source and write them into a database.

```
(defun populate-database ()
  (let (;; Make sure storing ends at the first object found in the cache:
        (plob::*default-depth* :cached))
    (clear-cache) ;; make sure the cache is empty
    ;; All locking has to be done in a transaction, so start one now:
    (with-transaction ()
      ;; Write-lock the whole store; the store lock will extend to the
      ;; end of the transaction opened in the (with-transaction ...) block:
      (write-lock-store)
      ;; The 'effective' populating function:
      (read-external-objects-and-write-them-to-database))))
```

References Parameter ***default-depth*** [Internal Manual 1997, p. 31], function **clear-cache** [API Manual 1997, p. 28], macro **with-transaction** [API Manual 1997, p. 135], function **write-lock-store** [API Manual 1997, p. 136].

Dumping a database

Here, the term 'dumping' means reading objects from a database and write them to an external sink.

```
(defun dump-database ()
  (let (;; Make sure loading ends at the first object found in the cache:
        (plob::*default-depth* :cached))
    (clear-cache) ;; make sure the cache is empty
    ;; All locking has to be done in a transaction, so start one now:
    (with-transaction ()
      ;; Read-lock the whole store; the store lock will extend to the
      ;; end of the transaction opened in the (with-transaction ...) block:
      (read-lock-store)
      ;; The 'effective' dump function:
      (read-database-objects-and-write-them-to-external-sink))))
```

[3]The performance improvement at storing objects is reached because PLOB's client C code checks if the client process holds a write lock on store level. When such a lock is found, the server is not asked at all to lock fresh created objects. This reduces the communication overhead dramaticaly. Instead, the objects are created into the C code client's cache, filled by the LISP level, and flushed to the server when they have been filled completely.

References Parameter ***default-depth*** [Internal Manual 1997, p. 31], function **clear-cache** [API Manual 1997, p. 28], macro **with-transaction** [API Manual 1997, p. 135], function **read-lock-store** [API Manual 1997, p. 110].

3.10 Using PLOB! with existing applications

It is possible to use PLOB! for making instances of existing applications persistent. This proceeding has been used already in PLOB! internally for integrating some LISP built in types as persistent classes.

3.10.1 Example: Persistent logical pathnames

For example, following steps have been done to create a persistent class for the built in class **logical-pathname** of LISPWORKS Common LISP:

1. A constructor function **make-logical-pathname-by-plob** [Internal Manual 1997, p. 76] is defined, which creates a transient instance of class **logical-pathname**:

   ```
   #+LispWorks
   (defun make-logical-pathname-by-plob
        (&key host directory name type version)
     "Create a LispWorks logical pathname which is loaded by PLOB"
     (system::make-logical-pathname-from-components
      host directory name type version))
   ```

   Since instances of class **logical-pathname** in LISPWORKS Common LISP are represented as structure instances, the arguments of this constructor resemble the `&key` arguments given to a system generated structure constructor function for a ‘normal’ `defstruct` class.

2. This instance constructor is registered as constructor function for instances of class **logical-pathname** into PLOB:

   ```
   #+LispWorks
   (setf (class-constructor (find-class 'logical-pathname))
         'make-logical-pathname-by-plob)
   ```

3. The class extent of class **logical-pathname** is set to `:cached`:

   ```
   #+LispWorks
   (setf (class-extent (find-class 'logical-pathname)) :cached)
   ```

4. The slot **system::device** of class **logical-pathname** is made transient, since it is not needed in the persistent representation:

   ```
   #+LispWorks
   (setf (slot-extent 'system::device (find-class 'logical-pathname))
         :transient)
   ```

Now, instances of the LISPWORKS Common LISP built in class **logical-pathname** are handled as persistent structure instances. The approach for ALLEGRO Common LISP would be similar as shown here.

3.10.2 General proceeding for legacy systems

Give me all your lupines, please!

— Monty Python

Using PLOB! with existing applications depends of course very much on the application in consideration, so more general rules and hints are given here. One of the following two proposals or a combination of them should be feasible to make instances of existing applications persistent.

Using the loose binding

The loose binding (section 'Loose Binding' on page 26) could be used for example by 'hooking' into the application at appropiate places, calling functions **store-object** [API Manual 1997, p. 132] and **load-object** [API Manual 1997, p. 38] on the objects to make persistent. A load could be done at application startup and a store at application shutdown. It should be checked if the application has generic functions for doing this.

Top-level declarations (section 'Top level declarations' on page 38) can be used to accomplish or optimize this binding. It is important to declare those slots as being `:transient` which reference pure-transient data, because the transitive closure of such data could become very large.

Using the tight binding

The tight binding (section 'Tight binding' on page 19) could be used by trying at first to retrieve the original class definitions of the classes whose instance should be made persistent, for example with the help of the LISP inspector. After having obtained the original class definition, there are two possibilities to continue, either to redefine the 'old' class definition or to create a new one.

Redefine old definition This is accomplished by adding the class option `(:metaclass persistent-metaclass)` to the original class definition, and the modified definition is evaluated. This would add a true tight binding, but might slow down the application considerably or might cause other problems, like application instability.

Create a new class definition This is accomplished by creating a new class with another name and a class option `(:metaclass persistent-metaclass)`, having more or less the same slots as the original class definition. This would create a new, persistent class tree besides the existing, transient one. The way-of-working is similar to the solution proposed by employing the loose binding: There must be some equalization between instances of both trees. Or, `(change-class)` could be used on those (single) transient instances which should be held persistent.

Chapter 4

Database functionality

This chapter describes concepts, functions and classes of PLOB! whose transient counterparts are either not found in [CLtLII 1990] at all because they represent certain database features not needed by a transient LISP system or whose meaning in PLOB! has been adopted for the needs in a persistent system.

4.1 Databases, sessions, transactions and locks

This section describes some terms and design issues of PLOB! concerning databases, sessions, transactions and locks from a programmer's point of view, and the relations between them.

4.1.1 Database

A database contains persistent objects. At the moment (1998/08/27), a PLOB! client can have only one database open at a time.[1] It is a global resource shared among all LISP lightweight processes running in a LISP image. A database is opened the very first time when the first 'session' is established.

Multiple Databases

Working with multiple databases would mean to open the first database, do something with it, close the first database, open the second database, etc. To be more concrete, for true working with multiple databases, the processing would look like:

```
(progn

  ;; Open the session; since the database is currently closed,
  ;; this implies opening the database referenced by the passed URL:
  (open-my-session "tcp://localhost/database1")
  ;; Do something:
  (do-something-with-database-1)
  ;; This closes all sessions and the database:
  (close-heap)
```

[1]A single PLOB! server can handle any (reasonable) number of clients.

```
    ;; Open the session; since the database is currently closed,
    ;; this implies opening the database referenced by the passed URL:
    (open-my-session "tcp://localhost/database2")
    ;; Do something:
    (do-something-with-database-2)
    ;; This closes all sessions and the database:
    (close-heap))
```

For historical reasons (because of its global character), the representation of a database is a bit unclear/smeared in PLOB.

4.1.2 Session

A session is used as an 'access organizer' or 'access helper' to a database. A session organizes access to the persistent objects contained in the database by ensuring that each access is within an active transaction, that a read access to a persistent object in the database is done within a read lock and a write access within a write lock. By default, each LISP lightweight process (thread) gets its own session when the thread does its first access to a persistent object. All sessions of a running LISP image are bound to the same underlying database.

The session associated to the current thread is contained in variable ***default-persistent-heap*** [API Manual 1997, p. 30]. There is one global session which is used at the bootstrap and for storing metaobject information; this one is contained in variable ***root-persistent-heap*** [Internal Manual 1997, p. 191].

Sessions are represented as persistent objects of class **persistent-heap** [API Manual 1997, p. 102].

References Function **open-my-session** [API Manual 1997, p. 45] and parameter ***database-url*** [API Manual 1997, p. 30], parameter ***default-database-url*** [Internal Manual 1997, p. 31]; function **close-my-session** [API Manual 1997, p. 29], function **show-sessions** [API Manual 1997, p. 116], function **p-sessions** [API Manual 1997, p. 84], macro **with-session** [API Manual 1997, p. 135].

4.1.3 Transaction

A transaction is some state-changing operation to the underlying database. The transactions implemented in PLOB! are two phased transactions. This transaction model ensures that each single transaction has isolated access to persistent objects; essentially, it means that a transaction is always divided into two phases, the first phase where locks are established and the second phase which starts with the first lock being released. From the user's side, the second phase is never visible, since it is embedded into the transaction's commit processing.

Figure 4.1 shows the possible states of a transaction; only in the active state, a transaction is allowed to change the database's state. According to the isolation table given in [Gray & Reuter 1993, p. 399], PLOB! implements an isolation of 3°. Since starting and stopping a transaction might have side effects in the highest application layer, transactions are always started and ended by the highest application layer.

A transaction could also be seen as a mean to ensure that the database is in a consistent state before the transaction has been started and after it has been finished. Here, consistency is defined in terms purely of the application programmer's point of view, and involves no other constraints.

For now, a transaction is bound exactly to one session. An active transaction in a session is for example shown in its print output; with no active transaction on the

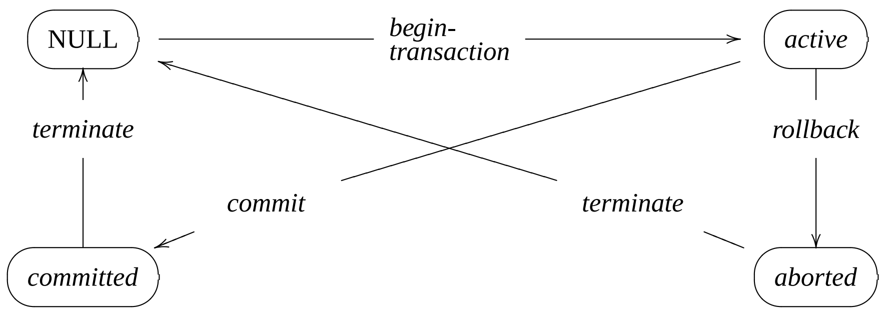


Figure 4.1: S/T diagram of a transaction

default session, the value of variable ***default-persistent-heap*** [API Manual 1997, p. 30] prints like:

```
CL-USER 35 > *default-persistent-heap*
#<heap kirschke@localhost ‘Initial Lisp Listener’ short-objid=133987850>
```

Within an active transaction, it prints like:

```
CL-USER 36 > (begin-transaction *default-persistent-heap*)
179048
CL-USER 37 > *default-persistent-heap*
#<heap kirschke@localhost ‘Initial Lisp Listener’ transaction=179048
                                                  short-objid=133987850>
CL-USER 38 > (in-transaction-p *default-persistent-heap*)
179048
CL-USER 39 > (end-transaction *default-persistent-heap*)
179048
CL-USER 40 > *default-persistent-heap*
#<heap kirschke@localhost ‘Initial Lisp Listener’ short-objid=133987850>
CL-USER 41 > (in-transaction-p *default-persistent-heap*)
NIL
```

Since transactions are bound to exactly one session, a transaction is not represented explicit but implicit by a session containing a transaction id.

References Macro **with-transaction** [API Manual 1997, p. 135], function **begin-transaction** [API Manual 1997, p. 5], function **end-transaction** [API Manual 1997, p. 31], function **cancel-transaction** [API Manual 1997, p. 12].

Aborting a transaction

At the moment, aborting a transaction will clear the cache to enforce a reload of the persistent objects into transient representations, with all of its consequences (section ‘Identity and caching’ on page 19). Future versions of PLOB! will support a more elaborate handling of transaction aborts.

4.1.4 Lock

A lock ensures that there is no conflicting access to a persistent object. In conjunction with transactions, locking is an implementation of the concept of isolation. The locking mechanism implemented is hierarchical or tree locking, a specialized version of predicate locking [Gray & Reuter 1993, p. 406]. Locking is possible on the levels **store** (lock all persistent objects), **vector** (lock one persistent object) and **element** (lock one slot of a persistent object) with the lock modes **read-only**, **read** or **write**. The locking protocol is a pessimistic one: An object must be locked before its state can be accessed. All locking is done implicit during accessing the state of a persistent object.

All locks must be set within an active transaction; they are bound to the transaction. When ending the transaction (either committing or aborting), all locks set in the scope of the transaction are removed. Before any access can be done to a persistent object, a read or write lock must be granted to the modifying transaction. If a lock cannot be granted, a lock conflict occurs. A lock conflict is given by the following rules:

1. Conflicts between locks set by different transactions are given by the conflict matrix shown in table 4.2. In essence, this matrix says that read locks are shared among a locked object, whereas a write lock requires no other lock set onto the object.

2. A lock which should be set by a transaction never conflicts with other locks already set by the same transaction. For example, if a transaction requested a write lock for a persistent object and the same transaction requests later a read lock, these locks do not conflict.

Already granted→ Requested↓	**RO-Intent**	**Read-Only**	**Read-Intent**	**Read**	**Write-Intent**	**Write**
Read-Only-Intent	✔	✔	✔	✔	✔	✔
Read-Only	✔	✔	✔	✔	✔	✔
Read-Intent	✔	✔	✔	✔	✔	✘
Read	✔	✔	✔	✔	✘	✘
Write-Intent	✔	✔	✔	✘	✔	✘
Write	✔	✘	✘	✘	✘	✘

✔ ≡ Compatible locks ✘ ≡ Conflicting locks

Table 4.2: Conflictmatrix for locks

A conflict is 'resolved' by the client requesting the conflicting lock to wait until either the other lock involved in the conflict is removed (for example, because another transaction holding this lock has terminated) or a timeout is encountered (this indicates a maybe not solvable conflict, like a deadlock, and will raise an error).

The functions **write-lock-store** [API Manual 1997, p. 136] and **read-lock-store** [API Manual 1997, p. 110] can be used to place a write or read lock onto the whole database. This is usefull when the whole database should be written or read, see section 'Storing or loading lots of data' on page 41 for an example.

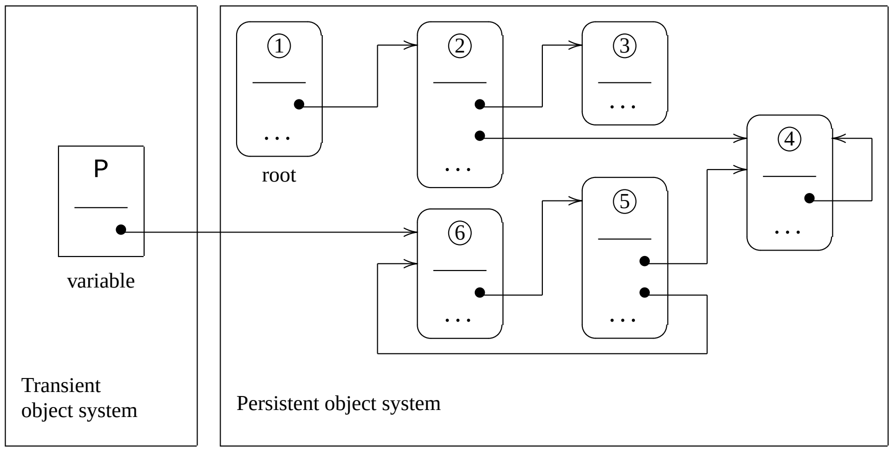
Locks are represented as persistent objects. They will almost never be seen directly by the database programmer, since most of the locking is done automatically.

References Function **write-lock-store** [API Manual 1997, p. 136], function **read-lock-store** [API Manual 1997, p. 110]; section **locking ...** [API Manual 1997, p. 38].

4.2 Making objects reachable

References from transient variables to persistent objects are not regarded by PLOB.

In figure 4.3, objects ① – ④ are reachable from the designated persistent root object and will not be garbage collected. Objects ⑤ and ⑥ are only reachable from a transient variable and are not reachable from the persistent root, since the persistent system does not know anything about references from transient objects to persistent objects. In consequence, objects ⑤ and ⑥ would be garbage-collected. The problem with references from (transient) variables to persistent objects is solved two-fold in PLOB:



①: Root object
②, ③, ④: Reachable objects ⑤, ⑥: Reachable objects by transient variable P

Figure 4.3: Reachable objects

ffline garbage collection.

1. A garbage collection is triggered if and only if there are no references from transient to persistent objects. This is the case when there is no (transient) client connected to the persistent object system. To be more specific, a garbage collection is triggered in the server after the last client has disconnected itself.

Persistent symbols with persistent values.

2. PLOB! offers an interface to persistent symbols, which are defined as being reachable, both the symbol itself and its value. Another advantage is that an object bound to a persistent symbol can be located in the persistent object system by the symbol's name.

Additionally, declaring an index on a persistent class using the tight binding will ensure reachability for the class' instances (section 'Indexes and reachability' on page 26).

4.3 Persistent symbols

As with transient symbols, persistent symbols serve for naming persistent objects.[2] Persistent symbols always have a global scope, there is no mean to establish a local scope for a persistent symbol. For convenience, the construct `#!`<*symbol*> is established by PLOB! to refer to the persistent <*symbol*> having the same name and package as the transient <*symbol*>.

```
CL-USER 42 > (setf #!*foo* 3)                                    ①
3
CL-USER 43 > #!*foo*
3
CL-USER 44 > *foo*                                               ②
Error: Attempt to take the value of the unbound variable '*FOO*'.
  [condition type: UNBOUND-VARIABLE]
;; ...
CL-USER 45 > #!*bar*                                             ③
Error: Cannot locate a persistent symbol named *BAR*.
```

[2]Named objects are found in ODMG [ODMG-2 1997, p. 242, **bind()** method], too, but ODMG does not cope with symbols, only with simple names.

```
  [condition type: SIMPLE-ERROR]
;; ...
```
④
```
CL-USER 46 > (p-intern '*bar*)
50327204         ;; This is the numeric objid of the created symbol
NIL              ;; NIL means it is a new created persistent symbol
```
⑤
```
CL-USER 47 > #!*bar*
#<unbound-marker=0x104>
```

The value of a persistent symbol can be set with special form **setf** [CLtLII 1990, p. 124–125] ①. The value of the transient symbol will neither be set nor be changed ②; only the name and package of the transient symbol are used to reference the persistent symbol's name and package. Opposed to transient symbols, a read access to a non-existing persistent symbol will not result in the persistent symbol being automatically inserted into the database's symbol table, but will raise an error ③. This is done to prevent the database from getting littered with a lot of automatic inserted and maybe otherwise unused symbols. Creating a new, unbound persistent symbol can be done by calling function **p-intern** [API Manual 1997, p. 71] ④. Referencing an unbound persistent symbol will not raise an error, but will return a special marker instance representing an unbound persistent object ⑤.

The complete reference chain to a persistent symbol is:

Persistent root $\rightarrow$ package table $\overset{*}{\rightarrow}$ package $\rightarrow$ list of internal symbols $\overset{*}{\rightarrow}$ symbol

References for persistent symbols

Allocator	Function **p-allocate-symbol** [API Manual 1997, p. 50]
Creator	Macro **p-setq** [API Manual 1997, p. 84]
Type Predicate	Function **p-symbolp** [API Manual 1997, p. 99]
Object Store	Function **(setf p-symbol)** [API Manual 1997, p. 96]
Object Load	Function **p-symbol** [API Manual 1997, p. 96]
Slot Reader	Function **p-symbol-function** [API Manual 1997, p. 97], function **p-symbol-name** [API Manual 1997, p. 97], function **p-symbol-package** [API Manual 1997, p. 98], function **p-symbol-plist** [API Manual 1997, p. 98], function **p-symbol-value** [API Manual 1997, p. 98]
Slot Writer	Function **(setf p-symbol-function)** [API Manual 1997, p. 97], function **(setf p-symbol-plist)** [API Manual 1997, p. 98], function **(setf p-symbol-value)** [API Manual 1997, p. 99], function **p-fmakunbound** [API Manual 1997, p. 67], function **p-makunbound** [API Manual 1997, p. 77]. Once established, the name of a persistent symbol can not be changed, so there is no writer for a symbol's name.
Information	Function **p-boundp** [API Manual 1997, p. 57], function **p-fboundp** [API Manual 1997, p. 65]
Administration	Function **p-find-symbol** [API Manual 1997, p. 66], function **p-intern** [API Manual 1997, p. 71], function **p-unintern** [API Manual 1997, p. 100]

4.4 Persistent packages

Persistent packages serve as containers for persistent symbols; all persistent packages are reachable and are therefore never garbage collected. At the moment, all persistent symbols belonging to a persistent package are internal persistent symbols. If the same database is used by different LISP systems (for example, by LISPWORKS Common LISP and ALLEGRO Common LISP at the same time), there is a need for exporting symbols and a persistent use-package list, see function **p-find-class** [API Manual 1997, p. 65] and section ‘Missing ‘use-package’ feature’ on page 92.

References Function **p-find-package** [API Manual 1997, p. 66], function **p-delete-package** [API Manual 1997, p. 63], function **p-package-internals** [API Manual 1997, p. 79], function **p-apropos-packages** [API Manual 1997, p. 51], function **p-package** [API Manual 1997, p. 77], function **p-package-name** [API Manual 1997, p. 79], function **p-package-externals** [API Manual 1997, p. 78].

4.5 Markers

PLOB! has an addtional class **marker**. Its nature and semantic shows some resemblance to Common LISP’s condition classes, with the difference that it does not have an associated error handling protocol. Markers are used for representing certain conditions, which are not necessarily error conditions.

Unbound slot The marker object contained in constant **+plob-unbound-marker+** [API Manual 1997, p. 109] is used for slots which are not bound to a value: A slot which references such a marker is considered as being unbound.

Unstorable object If PLOB! encounteres objects it can not store into the database, the instance in constant **+plob-unstorable-object-marker+** [API Manual 1997, p. 109] is stored instead of the unstorable object itself. Since PLOB! is almost type-complete, this marker is used very rare, mainly when attempting to store binary function code.

Extremal value The marker object contained in constant **+plob-min-marker+** [API Manual 1997, p. 109]/constant **+plob-max-marker+** [API Manual 1997, p. 109] represent a generic object with its state being equal to the state of the minimum/maximum object in a set. For a set of persistent numeric values, they serve as a representation for the minimum/maximum number contained in the set. For a set with persistent strings, they represent the first/last string in this [ASCII-]ordered set.

For example, let’s take the `*c*` btree from ‘Hierarchical keys’ on page 53. The data associated to the minimum/maximum key stored in btree `*c*` can be retrieved by:

```
CL-USER 48 > (getbtree +plob-min-marker+ *c*)
"string (1 1)."
CL-USER 49 > (getbtree +plob-max-marker+ *c*)
"string (4d0 3)."
```

The corresponding key values themselves can be requested by a call to function **btree-minkey** [API Manual 1997, p. 10]/function **btree-maxkey** [API Manual 1997, p. 10]:

```
CL-USER 50 > (btree-minkey *c*)
#(1 1)
CL-USER 51 > (btree-maxkey *c*)
```

```
#(4d0 3)
```

Often, those conditions are represented by LISP implementators by using singular objects, like a string or a symbol for each marker instance. The concept of a marker class represents those conditions explicit by instances of a disjunctive class. Markers are represented as immediates, and so do not take up much space.

References Constant **+plob-min-marker+** [API Manual 1997, p. 109], constant **+plob-max-marker+** [API Manual 1997, p. 109], constant **+plob-unbound-marker+** [API Manual 1997, p. 109], constant **+plob-unstorable-object-marker+** [API Manual 1997, p. 109].

4.6 Persistent btrees

A btree is similar to the well-known hash table: It associates a value to a key. The difference to a hash table is that the keys are 1-dimensional sorted. Two kinds of sortings can be used:

By identity This will sort the keys according to their *objids*. Since each persistent object has an *objid*, all persistent objects regardless of their class can be used as key objects in a single btree.

By state This will sort the keys according to their state. This implies for key objects to be inserted, that they must be able to compare their state with the state of all keys which are already stored in the btree. For the most important persistent classes used with btrees, compare methods have been implemented.

The kind of sorting is selected with the `:test` argument passed to function **make-btree** [API Manual 1997, p. 41]. A `:test` argument of `eq` will sort the keys by identity, and a `:test` argument of `equal` will sort the keys by state.

References Class **persistent-btree** [Internal Manual 1997, p. 167] and function **make-btree** [API Manual 1997, p. 41], function **getbtree** [API Manual 1997, p. 34], generic function **clrbtree** [API Manual 1997, p. 29], generic function **rembtree** [API Manual 1997, p. 111], function **btree-count** [API Manual 1997, p. 8], function **btree-size** [API Manual 1997, p. 11], generic function **btree-test** [API Manual 1997, p. 11], function **p-apropos-btree** [API Manual 1997, p. 51].

4.6.1 Btrees sorted by state

For an `equal` btree, the type(s) of keys it can contain is fixed with the first object inserted into the btree.

```
    CL-USER 52 > (setf *b* (make-btree :test 'equal))
    #<btree equal 0/0=0*678 short-objid=50327519>
①   CL-USER 53 > (setf (getbtree 1 *b*) "string 1.")
    "string 1."
②   CL-USER 54 > (setf (getbtree 2.0 *b*) "string 2.0.")
    "string 2.0."
③   CL-USER 55 > (setf (getbtree "three" *b*) "string three.")
    Error: From server at executing client:SH_btree_insert_by_string:
           splobbtree.c(1153): fnKeyCmp:
           Illegal search key "three" for
           object #<btree equal 2/678=1*678 short-objid=50327519>:
```

```
          compare failed with
          object 1.
     [condition type: POSTORE-ERROR]
```

For example, if the first object inserted into an equal btree is a number ①, only numbers can be inserted afterwards ②; trying to insert for example a string will raise an error ③.

Hierarchical keys

Using a non-atomic object as a key in an equal btree has a special meaning: PLOB! will try to insert the key according to an element-wise comparision to the slots of the keys already found in the btree. This way, a non-atomic key object is a compound key, with its first slot being the primary key, the second slot being the secondary key and so on:

```
CL-USER 56 > (setf *c* (make-btree :test 'equal))
#<btree equal 0/0=0*678 short-objid=50327520>
CL-USER 57 > (setf (getbtree #(1 1) *c*) "string (1 1).")
"string (1 1)."
CL-USER 58 > (setf (getbtree #(1 2) *c*) "string (1 2).")
"string (1 2)."
CL-USER 59 > (setf (getbtree #(1 3.0) *c*) "string (1 3.0).")
"string (1 3.0)."
CL-USER 60 > (setf (getbtree #(4d0 3) *c*) "string (4d0 3).")
"string (4d0 3)."
CL-USER 61 > (p-apropos-btree *c*)
#(1 1), data: #<simple-string 'string (1 1).' short-objid=50327507>
#(1 2), data: #<simple-string 'string (1 2).' short-objid=50327504>
#(1 3.0), data: #<simple-string 'string (1 3.0).' short-objid=50327502>
#(4.0d0 3), data: #<simple-string 'string (4d0 3).' short-objid=50327369>
```

4.6.2 Mapping btrees

Generic function **mapbtree** [API Manual 1997, p. 43] can be used to map all elements of a btree.

```
CL-USER 62 > (mapbtree #'(lambda (key data)                              ①
                            (format t "key ~A, data ~A~%" key data) t)    ②
                        *b*)
key 1, data string 1.
key 2.0, data string 2.0.
2                                                                         ③
CL-USER 63 > (mapbtree #'(lambda (key data)
                            (format t "key ~A, data ~A~%" key data) t)
                        *b* :descending t)                                ④
key 2.0, data string 2.0.
key 1, data string 1.
2                                                                         ③
CL-USER 64 > (mapbtree #'(lambda (key data)                              ⑤
                            (format t "key ~A, data ~A~%" key data) t)
                        *b* :>= 2)
key 2.0, data string 2.0.
1                                                                         ③
```

Since the keys are ordered, the map function is called by default with the keys in ascending order ①. It is important that the map function returns a non-nil value if the mapping should continue with the next element ②. The call to generic function

mapbtree [API Manual 1997, p. 43] returns the number of mapped btree elements ③. If the `::descending` argument is passed with a non-`nil` value, the key interval is iterated in descending order ④. It is possible to restrict the mapping to an interval of key values ⑤. The lower key to map can be passed with a `::>` or `::>=` argument, the upper key to map with a `::<` or `::<=` argument.

It is allowed to delete the key currently passed to the map function from the btree.

References Generic function **mapbtree** [API Manual 1997, p. 43]

4.6.3 Cursors on btrees

An access interface similar to cursors in SQL is available for btrees. Instances of class **persistent-btree-mapper** [Internal Manual 1997, p. 167] can be used for accessing a btree by positioning. For example, create a btree with some elements in it.

```
CL-USER 65 > (setf *btree* (make-btree :test 'equal))
#<btree equal 0/0=0*678 short-objid=50327518>
CL-USER 66 > (dotimes (i 1000) (setf (getbtree i *btree*)
                                     (format nil "String ~A" i)))
```

Now, the btree is accessed by an instance of class **persistent-btree-mapper** [Internal Manual 1997, p. 167].

```
①    CL-USER 67 > (setf *mapper* (make-btree-mapper *btree*))
②    CL-USER 68 > (btree-mapper-seek *mapper* 1)
     1
     0
     "String 0"
②    CL-USER 69 > (btree-mapper-seek *mapper* 1)
     1
     1
     "String 1"
③    CL-USER 70 > (btree-mapper-seek *mapper* -1 :end)
     1
     998
     "String 998"
```

A mapper is established by calling the constructor of the mapper with the btree as its argument which should be iterated ①; further optional arguments can specify a search interval for the mapper. With a call to function **btree-mapper-seek** [API Manual 1997, p. 9], the next or previous element is returned. Positioning can be done relative to the start or end of the current search interval ③.

References Function **make-btree-mapper** [API Manual 1997, p. 42], function **btree-mapper-search** [API Manual 1997, p. 8], function **btree-mapper-seek** [API Manual 1997, p. 9], function **(setf btree-mapper-seek)** [API Manual 1997, p. 9]

4.6.4 Setting the page size

The internal used page size of a btree can be set either by passing the key argument `:pagesize` along with the number of objects each btree page should hold, or by calling function **(setf btree-pagesize)** [API Manual 1997, p. 10]. For very small or very big btrees, a smaller or larger page size than the default page size should be used. To be more exact, the page size' optimum is fulfilled for

$$\begin{aligned} n^n &\approx o \\ n &\equiv \text{page size} \\ o &\equiv \text{total number of objects in btree} \end{aligned}$$

References Function **make-btree** [API Manual 1997, p. 41], function **(setf btree-pagesize)** [API Manual 1997, p. 10], function **btree-pagesize** [API Manual 1997, p. 10].

4.6.5 Requesting extremal values

The data object associated to the minimum or maximum key object contained in a btree can be requested by passing the special marker objects constant **+plob-min-marker+** [API Manual 1997, p. 109] and **+plob-max-marker+** [API Manual 1997, p. 109] to function **getbtree** [API Manual 1997, p. 34]. This way, it is also possible to modify the data objects associated to the minimum or maximum key by passing the marker object to **(setf getbtree)** [API Manual 1997, p. 35]. The minimum or maximum key object itself is returned by a call to function **btree-minkey** [API Manual 1997, p. 10] or function **btree-maxkey** [API Manual 1997, p. 10].

References Constant **+plob-min-marker+** [API Manual 1997, p. 109], constant **+plob-max-marker+** [API Manual 1997, p. 109]; Function **btree-minkey** [API Manual 1997, p. 10], function **btree-maxkey** [API Manual 1997, p. 10].

4.6.6 Inspecting btrees

In LISPWORKS Common LISP, a btree's contents can be viewed with the built in LISPWORKS inspector. Simply start an inspector on a btree object to look at its contents. To see the structure of the btree correctly, in the inspector's 'View' menu the entry 'Unsorted' should be selected.

Example: Inspecting the index associated to a slot

For example, store some thousand persons by evaluating `(store-n-random-people 2000)` (found in `plob-2.11/src/example/plob-example.lisp`. Evaluate `(p-find-class 'person)`, this will return as first value the class description-object of class **person**. Inspect this first value. Next, inspect the vector labelled `PLOB::P-EFFECTIVE-SLOTS`. Next, inspect the instance marked as being the effective-slot-description for slot **soc-sec-#**. In this slot description-object, the index defined for slot **soc-sec-#** of class **person** can be found within the line labelled `PLOB::P-INDEX`. Inspect this instance; this will show the btree which is used for representing the index declared on slot **soc-sec-#** within class **person**.

4.7 Regular expressions

Regular expressions are builtin objects in PLOB! and can be used as a filter in iterating on btrees.

```
CL-USER 71 > (setf *o* (p-make-regex "Heik[eo]"))
#<regex 'Heik[eo]' short-objid=50328180>
CL-USER 72 > (setf *c* (make-btree :test 'equal))
#<btree equal 0/0=0*678 short-objid=50328176>
CL-USER 73 > (setf (getbtree "Heiko" *c*) "Kirschke")
"Kirschke"
CL-USER 74 > (setf (getbtree "Heike" *c*) "Pflugradt")
"Pflugradt"
CL-USER 75 > (setf (getbtree "Nicola" *c*) "Kirschke")
"Kirschke"
CL-USER 76 > (mapbtree #'(lambda (key data)
                          (format t "key ~A, data ~A~%" key data) t)
                    *c* :filter *o*)
key Heike, data Pflugradt
key Heiko, data Kirschke
2
```

The regular expressions can only be used in iterating queries, not in searching queries. The reason is that there is no direct search criteria implemented for regular expression on btress, instead a brute-force search is done, applying the regular expression as a filter on all keys read from the btree. For this reason, setting the minimum and maximum search keys is important, since this could reduce search time.

PLOB! uses the regex code of Henry Spencer as e.g. also used in cygwin and FreeBSD. Regular expressions are used only for matching, i.e. the code for locating subexpressions has not been interfaced to LISP. In the following sections the manpages of regex have been added and adopted to its usage in PLOB!

4.7.1 regex(3)

NAME

Function **p-make-regex** [API Manual 1997, p. 76], function **p-compile-regex** [API Manual 1997, p. 59] - regular-expression library

SYNOPSIS

```
p-make-regex
  pattern
  &key
  ;; regcomp() flags
  (basic nil) (extended t) (icase nil) (nosub t) (newline nil)
  (nospec nil) (notmatching nil)
  ;; regexec() flags
  (notbol nil) (noteol nil)
  (trace nil) (large nil) (backref nil)
  (p-heap *default-persistent-heap*)

p-compile-regex
  objid
  &optional (p-heap *default-persistent-heap*)
```

DESCRIPTION

These routines implement POSIX 1003.2 regular expressions ("RE"s); see 'regex(7)' on page 59.

Function **p-make-regex** [API Manual 1997, p. 76] creates a persistent regular expression, function **p-compile-regex** [API Manual 1997, p. 59] compiles the regular expression.

At compiling, the following flags passed in at a call to function **p-make-regex** [API Manual 1997, p. 76] are used:

`:basic` This is a synonym for 0, provided as a counterpart to `:extended` to improve readability. This is an extension, compatible with but not specified by POSIX 1003.2, and should be used with caution in software intended to be portable to other systems.

`:extended` Compile modern ("extended") REs, rather than the obsolete ("basic") REs that are the default.

`:icase` Compile for matching that ignores upper/lower case distinctions. See 'regex(7)' on page 59.

`:nosub` Compile for matching that need only report success or failure, not what was matched.

`:newline` Compile for newline-sensitive matching. By default, newline is a completely ordinary character with no special meaning in either REs or strings. With this flag, '`[^`' bracket expressions and '.' never match newline, a '`^`' anchor matches the null string after any newline in the string in addition to its normal function, and the '`$`' anchor matches the null string before any newline in the string in addition to its normal function.

`:nospec` Compile with recognition of all special characters turned off. All characters are thus considered ordinary, so the "RE" is a literal string. This is an extension, compatible with but not specified by POSIX 1003.2, and should be used with caution in software intended to be portable to other systems. `:extended` and `:nospec` may not be used in the same call to regcomp.

`:notmatching` Signal a match for strings not matching the pattern.

The regular expression can be used directly as a first argument to function **p-compare** [API Manual 1997, p. 59]. At matching, the following flags passed in to the call to function **p-make-regex** [API Manual 1997, p. 76] are used:

`:notbol` The first character of the string is not the beginning of a line, so the '`^`' anchor should not match before it. This does not affect the behavior of newlines under `:newline`.

`:noteol` The NUL terminating the string does not end a line, so the '`$`' anchor should not match before it. This does not affect the behavior of newlines under `:newline`.

`:trace` Tracing of execution.

`:large` Force large representation.

`:backref` Force use of backref code.

See 'regex(7)' on page 59 for a discussion of what is matched in situations where an RE or a portion thereof could match any of several substrings of string.

See function **p-compare** [API Manual 1997, p. 59] for a desrciption of its return values.

None of these functions references global variables except for tables of constants; all are safe for use from multiple threads if the arguments are safe.

IMPLEMENTATION CHOICES

There are a number of decisions that 1003.2 leaves up to the implementor, either by explicitly saying "undefined" or by virtue of them being forbidden by the RE grammar. This implementation treats them as follows.

See 'regex(7)' on page 59 for a discussion of the definition of case-independent matching.

There is no particular limit on the length of REs, except insofar as memory is limited. Memory usage is approximately linear in RE size, and largely insensitive to RE complexity, except for bounded repetitions. See 'BUGS' on page 58 for one short RE using them that will run almost any system out of memory.

A backslashed character other than one specifically given a magic meaning by 1003.2 (such magic meanings occur only in obsolete [“basic”] REs) is taken as an ordinary character.

Any unmatched [is a `+regex-ebrack+` error.

Equivalence classes cannot begin or end bracket-expression ranges. The endpoint of one range cannot begin another.

RE_DUP_MAX, the limit on repetition counts in bounded repetitions, is 255.

A repetition operator (?, *, +, or bounds) cannot follow another repetition operator. A repetition operator cannot begin an expression or subexpression or follow ‘^’ or ‘|’.

‘|’ cannot appear first or last in a (sub)expression or after another ‘|’, i.e. an operand of ‘|’ cannot be an empty subexpression. An empty parenthesized subexpression, ‘()’, is legal and matches an empty (sub)string. An empty string is not a legal RE.

A ‘{’ followed by a digit is considered the beginning of bounds for a bounded repetition, which must then follow the syntax for bounds. A ‘{’ not followed by a digit is considered an ordinary character.

‘^’ and ‘$’ beginning and ending subexpressions in obsolete (“basic”) REs are anchors, not ordinary characters.

SEE ALSO

grep(1), ‘regex(7)’ on page 59

POSIX 1003.2, sections 2.8 (Regular Expression Notation) and B.5 (C Binding for Regular Expression Matching).

DIAGNOSTICS

Non-zero error codes from regcomp and regexec include the following:

`+regex-nomatch+`	regexec() failed to match
`+regex-badpat+`	invalid regular expression
`+regex-ecollate+`	invalid collating element
`+regex-ectype+`	invalid character class
`+regex-eescape+`	applied to unescapable character
`+regex-esubreg+`	invalid backreference number
`+regex-ebrack+`	brackets [ ] not balanced
`+regex-eparen+`	parentheses ( ) not balanced
`+regex-ebrace+`	braces { } not balanced
`+regex-badbr+`	invalid repetition count(s) in { }
`+regex-erange+`	invalid character range in [ ]
`+regex-espace+`	ran out of memory
`+regex-badrpt+`	?, *, or + operand invalid
`+regex-empty+`	empty (sub)expression
`+regex-assert+`	“can’t happen”-you found a bug
`+regex-invarg+`	invalid argument, e.g. negative-length string

HISTORY

Written by Henry Spencer, formerly `mailto:henry@zoo.toronto.edu`.

BUGS

This is an alpha release with known defects. Please report problems.

There is one known functionality bug. The implementation of internationalization is incomplete: the locale is always assumed to be the default one of 1003.2, and only the collating elements etc. of that locale are available.

The back-reference code is subtle and doubts linger about its correctness in complex cases.

Regexec performance is poor. This will improve with later releases. Nmatch exceeding 0 is expensive; nmatch exceeding 1 is worse. Regexec is largely insensitive to RE complexity except that back references are massively expensive. RE length does matter; in particular, there is a strong speed bonus for keeping RE length under about 30 characters, with most special characters counting roughly double.

Regcomp implements bounded repetitions by macro expansion, which is costly in time and space if counts are large or bounded repetitions are nested. An RE like, say, '((((a{1,100}){1,100}){1,100}){1,100}){1,100}' will (eventually) run almost any existing machine out of swap space.

There are suspected problems with response to obscure error conditions. Notably, certain kinds of internal overflow, produced only by truly enormous REs or by multiply nested bounded repetitions, are probably not handled well.

Due to a mistake in 1003.2, things like 'a)b' are legal REs because ')' is a special character only in the presence of a previous unmatched '('. This can't be fixed until the spec is fixed.

The standard's definition of back references is vague. For example, does 'a\(\(b\)*\2\)*d' match 'abbbd'? Until the standard is clarified, behavior in such cases should not be relied on.

The implementation of word-boundary matching is a bit of a kludge, and bugs may lurk in combinations of word-boundary matching and anchoring.

4.7.2 regex(7)

NAME

regex - POSIX 1003.2 regular expressions

DESCRIPTION

Regular expressions ("RE"s), as defined in POSIX 1003.2, come in two forms: modern REs (roughly those of egrep; 1003.2 calls these "extended" REs) and obsolete REs (roughly those of ed; 1003.2 "basic" REs). Obsolete REs mostly exist for backward compatibility in some old programs; they will be discussed at the end. 1003.2 leaves some aspects of RE syntax and semantics open; '+' marks decisions on these aspects that may not be fully portable to other 1003.2 implementations.

A (modern) RE is one+ or more non-empty+ branches, separated by '|'. It matches anything that matches one of the branches.

A branch is one+ or more pieces, concatenated. It matches a match for the first, followed by a match for the second, etc.

A piece is an atom possibly followed by a single+ '*', '+', '?', or bound. An atom followed by '*' matches a sequence of 0 or more matches of the atom. An atom followed by '+' matches a sequence of 1 or more matches of the atom. An atom followed by '?' matches a sequence of 0 or 1 matches of the atom.

A bound is '{' followed by an unsigned decimal integer, possibly followed by ',' possibly followed by another unsigned decimal integer, always followed by '}'. The integers must lie between 0 and RE_DUP_MAX (255+) inclusive, and if there are two of them, the first may not exceed the second. An atom followed by a bound containing one integer i and no comma matches a sequence of exactly i matches of the atom. An atom followed by a bound containing one integer i and a comma matches a sequence of i or more matches of the atom. An atom followed by a bound containing two integers i and j matches a sequence of i through j (inclusive) matches of the atom.

An atom is a regular expression enclosed in '()' (matching a match for the regular expression), an empty set of '()' (matching the null string)+, a bracket expression (see below), '.' (matching any single character), '^' (matching the null string at the beginning of a line), '$' (matching the null string at the end of a line), a '\' followed by one of the characters '^.[$()|*+?{\' (matching that character taken as an ordinary character), a '\' followed by any other character+ (matching that character taken as an ordinary character, as if the '\' had not been present+), or a single character with no other significance (matching that character). A '{' followed by a character other than a digit is an ordinary character, not the beginning of a bound+. It is illegal to end an RE with '\'.

A bracket expression is a list of characters enclosed in '[]'. It normally matches any single character from the list (but see below). If the list begins with '^', it matches any single character (but see below) not from the rest of the list. If two characters in the list are separated by '-', this is shorthand for the full range of characters between those two (inclusive) in the collating sequence, e.g. '[0-9]' in ASCII matches any decimal digit. It is illegal+ for two ranges to share an endpoint, e.g. 'a-c-e'. Ranges are very collating-sequence-dependent, and portable programs should avoid relying on them.

To include a literal ']' in the list, make it the first character (following a possible '^'). To include a literal '-', make it the first or last character, or the second endpoint of a range. To use a literal '-' as the first endpoint of a range, enclose it in '[.' and '.]' to make it a collating element (see below). With the exception of these and some combinations using '[' (see next paragraphs), all other special characters, including '\', lose their special significance within a bracket expression.

Within a bracket expression, a collating element (a character, a multi-character sequence that collates as if it were a single character, or a collating-sequence name for either) enclosed in '[.' and '.]' stands for the sequence of characters of that collating element. The sequence is a single element of the bracket expression's list. A bracket expression containing a multi-character collating element can thus match more than one character, e.g. if the collating sequence includes a 'ch' collating element, then the RE '[[.ch.]]*c' matches the first five characters of 'chchcc'.

Within a bracket expression, a collating element enclosed in '[=' and '=]' is an equivalence class, standing for the sequences of characters of all collating elements equivalent to that one, including itself. (If there are no other equivalent collating elements, the treatment is as if the enclosing delimiters were '[.' and '.]'.) For example, if o and are the members of an equivalence class, then '[[=o=]]', '[[==]]', and '[o]' are all synonymous. An equivalence class may not+ be an endpoint of a range.

Within a bracket expression, the name of a character class enclosed in '[:' and ':]' stands for the list of all characters belonging to that class. Standard character class names are:

- alnum digitpunct
- alpha graphspace
- blank lowerupper
- cntrl printxdigit

These stand for the character classes defined in ctype(3). A locale may provide others. A character class may not be used as an endpoint of a range.

There are two special cases+ of bracket expressions: the bracket expressions '[[:<:]]' and '[[:>:]]' match the null string at the beginning and end of a word respectively. A word is defined as a sequence of word characters which is neither preceded nor followed by word characters. A word character is an alnum character (as defined by ctype(3)) or an underscore. This is an extension, compatible with but not specified by

POSIX 1003.2, and should be used with caution in software intended to be portable to other systems.

In the event that an RE could match more than one substring of a given string, the RE matches the one starting earliest in the string. If the RE could match more than one substring starting at that point, it matches the longest. Subexpressions also match the longest possible substrings, subject to the constraint that the whole match be as long as possible, with subexpressions starting earlier in the RE taking priority over ones starting later. Note that higher-level subexpressions thus take priority over their lower-level component subexpressions.

Match lengths are measured in characters, not collating elements. A null string is considered longer than no match at all. For example, ‘bb*’ matches the three middle characters of ‘abbbc’, ‘(wee|week)(knights|nights)’ matches all ten characters of ‘weeknights’, when ‘(.*).*’ is matched against ‘abc’ the parenthesized subexpression matches all three characters, and when ‘(a*)*’ is matched against ‘bc’ both the whole RE and the parenthesized subexpression match the null string.

If case-independent matching is specified, the effect is much as if all case distinctions had vanished from the alphabet. When an alphabetic that exists in multiple cases appears as an ordinary character outside a bracket expression, it is effectively transformed into a bracket expression containing both cases, e.g. ‘x’ becomes ‘[xX]’. When it appears inside a bracket expression, all case counterparts of it are added to the bracket expression, so that (e.g.) ‘[x]’ becomes ‘[xX]’ and ‘[^x]’ becomes ‘[^xX]’.

No particular limit is imposed on the length of REs+. Programs intended to be portable should not employ REs longer than 256 bytes, as an implementation can refuse to accept such REs and remain POSIX-compliant.

Obsolete (“basic”) regular expressions differ in several respects. ‘|’, ‘+’, and ‘?’ are ordinary characters and there is no equivalent for their functionality. The delimiters for bounds are ‘{’ and ‘}’, with ‘’ and ‘’ by themselves ordinary characters. The parentheses for nested subexpressions are ‘fl*and*‘’, with ‘(’ and ‘)’ by themselves ordinary characters. ‘^’ is an ordinary character except at the beginning of the RE or+ the beginning of a parenthesized subexpression, ‘$’ is an ordinary character except at the end of the RE or+ the end of a parenthesized subexpression, and ‘*’ is an ordinary character if it appears at the beginning of the RE or the beginning of a parenthesized subexpression (after a possible leading ‘^’). Finally, there is one new type of atom, a back reference: ‘\’ followed by a non-zero decimal digit d matches the same sequence of characters matched by the dth parenthesized subexpression (numbering subexpressions by the positions of their opening parentheses, left to right), so that (e.g.) ‘\([bc]\)\1’ matches ‘bb’ or ‘cc’ but not ‘bc’.

SEE ALSO

‘regex(3)’ on page 56

POSIX 1003.2, section 2.8 (Regular Expression Notation).

HISTORY

Written by Henry Spencer, based on the 1003.2 spec.

BUGS

Having two kinds of REs is a botch.

The current 1003.2 spec says that ‘)’ is an ordinary character in the absence of an unmatched ‘(’; this was an unintentional result of a wording error, and change is likely. Avoid relying on it.

Back references are a dreadful botch, posing major problems for efficient implementations. They are also somewhat vaguely defined (does 'a\(\(b\)*\2\)*d' match 'abbbd'?). Avoid using them.

1003.2's specification of case-independent matching is vague. The "one case implies all cases" definition given above is current consensus among implementors as to the right interpretation.

The syntax for word boundaries is incredibly ugly.

4.8 Class management

A class metaobject is stored into the database as a class description-object the first time an instance of the class is to be stored to the database. Vice versa, if a persistent instance is loaded from the database, its class definition is compiled into the transient LISP image. A class description-object is a poor man's version of a class metaobject: In contrary to the elegant definition of metaobjects based on their behavior as given in [AMOP 1991], it is defined in terms of its structure. This restriction was necessary since PLOB! can not store methods.

Conflicts are resolved at the moment that at storing a class a definition mismatch will result in the class being stored anew, and a conflict at loading a class will be solved by overwriting the transient definition by the persistent one. For future directions, see section 'Views onto persistent objects' on page 93.

Not all slots of a class metaobject are stored; some slots are omitted, since otherwise the transitive closure of the class metaobject would contain more or less all classes of the current LISP image. For example, the slot containing all subclasses of a class is omitted, since this would store all classes of the current LISP image to the database (when referencing the class' superclasses). Since PLOB! can not store binary function code, no methods are stored, too. System created methods, like the ones created by the `:accessor`, `:reader` or `:writer` slot options, are not stored as methods, but will be generated when the class is compiled into the current LISP image.

If a class is deleted from the class table by a call to function **p-delete-class** [API Manual 1997, p. 62], it will be reinserted if an instance of the class is referenced.

References Function **p-find-class** [API Manual 1997, p. 65], function **p-delete-class** [API Manual 1997, p. 62], function **p-class-of** [API Manual 1997, p. 58], function **p-apropos-classes** [API Manual 1997, p. 51].

4.9 Schema evolution

The default schema evolution for all structure classes is specified by the value of parameter ***default-structure-schema-evolution*** [Internal Manual 1997, p. 32]; for CLOS classes, the value of parameter ***default-clos-schema-evolution*** [Internal Manual 1997, p. 31] is used.

References Generic function **(setf schema-evolution)** [API Manual 1997, p. 113], parameter ***default-structure-schema-evolution*** [Internal Manual 1997, p. 32], parameter ***default-clos-schema-evolution*** [Internal Manual 1997, p. 31].

4.10 Other useful functions

Function **p-apropos** [API Manual 1997, p. 50] will print out all persistent symbols matching with their name the passed regular expression:

```
CL-USER 77 > (p-apropos "person")
common-lisp-user::*last-name->person-list*,
                  value: #<btree equal 9/678 short-objid=50329169>
common-lisp-user::person
common-lisp-user::person-age (defined)
common-lisp-user::person-first-name (defined)
common-lisp-user::person-last-name (defined)
common-lisp-user::person-name (defined)
common-lisp-user::person-occupation (defined)
common-lisp-user::person-sex (defined)
common-lisp-user::person-soc-sec-# (defined)
CL-USER 78 > (p-apropos "[os].*lock")
plob::one-lock
plob::sum-lock
```

This is the output after file `plob-2.11/src/example/plob-example.lisp` has been compiled and loaded and some random persons have been generated (see function **random-person** [API Manual 1997, p. 146]). A `(defined)` indicates that a function binding has been done to the persistent symbol as explained in section 'Binary function code' on page 36.

Functions **p-statistics** [API Manual 1997, p. 87] and **p-configuration** [API Manual 1997, p. 61] will return some low level database informations.

Calling function **p-stabilise** [API Manual 1997, p. 87] will force a flush of all pending file operations in the server.

References Function **p-apropos** [API Manual 1997, p. 50], function **p-statistics** [API Manual 1997, p. 87], function **p-configuration** [API Manual 1997, p. 61], function **p-stabilise** [API Manual 1997, p. 87].

4.11 Performance

Don't let the sun go down on me.

— Elton John

[PLOB! 1.0 was developed on a SUN SPARCstation.]

The equipment used for performance testing was a 200 MHz Pentium-II PC with 64 MB RAM, standard EIDE Harddisk, Windows/NT 4.0 build 1381 service pack 3. The LISP system used was LispWorks Common LISP 4.0.1. The checked PLOB! version was 2.07. The numbers in the very right column are operations per second. Those numbers marked 'S+' are with a server running on `localhost` (that means, on the same machine as the LispWorks client using the server); the numbers marked by 'S-' are for the serverless version. In the performance measurements, care was taken that all operations involved a real database access, that means, no cache was used for the storing or loading operations.

4.11.1 Sequences

This section contains performance measurements for sequences, namely strings, lists and vectors.

Step	**Code**	**Description**	**Op./s**
1	`(store-object (make-string 32 :initial-element #\Space))`	Store strings with a length of 32 characters.	S+ 182 S- 401
2	`(load-object <objid>)`	Load strings with a length of 32 characters. <*objid*> references one of the persistent strings as generated in the last performance measurement step.	S+ 92 S- 467
3	`(store-object (make-list <length> :initial-element <fixnum>))`	Store a linear list with <*length*> elements, containing only immediates; Op./s is the number of stored list elements per second.	S+ 560 S- 1666
4	`(load-object <objid>)`	Load a linear list; <*objid*> references a persistent linear list as generated in step 1. Op./s is the number of loaded list elements per second.	S+ 414 S- 1428
5	`(store-object (make-array <length> :initial-element <float>))`	Store a simple vector with <*length*> elements, containing non-immediate <*floats*>; Op./s is the number of stored vector elements per second.	S+ 600 S- 4761
6	`(load-object <objid>)`	Load a simple vector; <*objid*> references a persistent simple vector as generated in step 5. Op./s is the number of loaded vector elements per second.	S+ 2100 S- 2857
7	`(store-object (make-array <length> :initial-element <fixnum>))`	Store a simple vector with <*length*> elements, containing only immediates; Op./s is the number of stored vector elements per second.	S+ 14000 S- 12500
8	`(load-object <objid>)`	Load a simple vector; <*objid*> references a persistent simple vector as generated in step 7. Op./s is the number of loaded vector elements per second.	S+ 20895 S- 33333

Step 7 is the only performance test step where the server mode outperforms the serverless mode. The only explanation I have is that this performance test step has been done with a rather big array containing 100000 elements, and that the blockwise (and implicit concurrent) transfer of the server mode does a better job here than the serverless mode, which does no blockwise or concurrent transfer at all.

4.11.2 CLOS instances

Following CLOS class has been used for testing tight-bounded persistency (section 'Tight binding' on page 19):

```
(defclass p-example-clos-class ()

  ((slot-3 :initform nil :extent :transient)

   (slot-5 :initform nil :extent :cached)

   (slot-7 :initform nil :extent :cached-write-through))

  (:metaclass persistent-metaclass))
```

An `:extent` of `:transient` means that the slot is not represented in the database at all. An `:extent` of `:cached` means that the slot is represented in the database, but will only be updated when explicitely requesting so. An `:extent` of `:cached-write-through` will update the slot in the database each time it is written to; this is the default extent used by PLOB! (section 'Additional slot options' on page 19).

Step	Code	Description	Op./s
9	`(make-instance 'p-example-clos-class)`	Allocation of a persistent instance and initialization of their slots according to their `:initform`.	S+ 165 S- 285
10	`(load-object <objid>)`	Load of a persistent instance; *<objid>* references a persistent instance as generated in step 9.	S+ 240 S- 344
11	`(setf (slot-value <object> 'slot-3) <fixnum>)`	Set a `:transient` slot of a tight-bounded persistent class (implies no storing of the slot's value to the database). *<object>* references a persistent instance as generated in step 9.	21000
12	`(slot-value <object> 'slot-3)`	Load the value of a `:transient` slot of a tight-bounded persistent class (implies no loading of the slot's value from the database). *<object>* references a persistent instance as generated in step 9.	29400
13	`(setf (slot-value <object> 'slot-5) <fixnum>)`	Set a `:cached` slot of a tight-bounded persistent class (implies no storing of the slot's value to the database). *<object>* references a persistent instance as generated in step 9.	22000

Step	Code	Description	Op./s
14	`(slot-value <object> 'slot-5)`	Load the value of a `:cached` slot of a tight-bounded persistent class (implies no loading of the slot's value from the database). <*object*> references a persistent instance as generated in step 9.	31300
15	`(setf (slot-value <object> 'slot-7) <fixnum>)`	Set a `:cached-write-through` slot of a tight-bounded persistent class to an immediate fixnum. <*object*> references a persistent instance as generated in step 9.	S+ 384 S- 1315
16	`(slot-value <object> 'slot-7)`	Load the value of a `:cached-write-through` slot of a tight-bounded persistent class. The value loaded is an immediate fixnum. <*object*> references a persistent instance as generated in step 9.	31300

For steps 11–14 and 16 there are no performance differences between the serverless and server mode, because no data transfer between the client and the database is involved (for these steps, the LISP layer is never left).

4.11.3 Database-specific functions

Step	Code	Description	Op./s
17	`(p-allocate-cons)`	Allocation of an empty persistent cons cell.	S+ 1639 S- 9090
18	`(with-transaction () nil)`	Empty transaction.	S+ 770 S- 5000
19	`(with-transaction () (write-lock-store) nil)`	Empty transaction locking the whole database (this performance is comparable to the performance of single-object level locking).	S+ 476 S- 1818

4.11.4 Different usage of transactions and locking

In this section, the performance is shown for the single operation of step 9, but with different usage of transactions and object locking. The default behavior of PLOB! as shown in step 9 involves one transaction for each call to generic function **make-instance** [CLtLII 1990, p. 848] and the (implicit) usage of some locks at storing the object. Using transactions and locks on bigger 'chunks' of objects improves performance for a single user significantly, but will make the 'granularity' of object access more coarse, making concurrent access slower.

Step	Code	Description	Op./s

9	`(make-instance`
	`'p-example-clos-`
	`class)`
		Allocation of a persistent instance and initialization of their slots according to their `:initform` (this is step 9 repeated here for completeness).	S+ 165
			S- 285

20	`(with-transaction ()`
	`(dotimes (i 1000)`
	`(make-instance`
	`'p-example-clos-`
	`class)))`
		1000 instances of step 9 generated in a single transaction; Op./s is the number of stored instances per second.	S+ 175
			S- 322

21	`(with-transaction ()`
	`(write-lock-store)`
	`(dotimes (i 1000)`
	`(make-instance`
	`'p-example-clos-`
	`class)))`
		1000 instances of step 9 generated in a single transaction with an exclusive write lock set onto the whole database; Op./s is the number of stored instances per second.	S+ 270
			S- 357

4.11.5 File space allocation

This section explains how much file space is allocated by PLOB! for a persistent object. All persistent objects are stored in a (memory-mapped) file; this is the file named `stablestore` in each database directory. Immediates use up no file space at all.

Non-CLOS objects

Non-immediates, non-CLOS objects and `defstruct` objects have an overhead of 6 words $\equiv$ 24 bytes. The total file space allocated for such an object is the sum of the overhead plus 1 word $\equiv$ 4 bytes per 'slot'. For example, a cons cell has 2 slots (`car` and `cdr`) and will take up a total of 6 + 2 = 8 words $\equiv$ 24 + 8 = 32 bytes.

CLOS objects

For CLOS instances, the overhead is 14 words $\equiv$ 56 bytes plus 1 word $\equiv$ 4 bytes per slot. For example, a CLOS instance with 2 slots will take up a total of 14 + 2 = 16 words $\equiv$ 56 + 8 = 64 bytes.

4.12 Technical data

Architecture	Client/server
Operating systems	Solaris 2.x, Linux kernel version 2.x, Windows XP. IRIX 6.x is supported on request.
LISP systems	LISPWORKS Common LISP 4.4, ALLEGRO Common LISP 7
Databases per Server	Any reasonable number of open databases, depending on installed memory.
Databases per Client	1
Multi user access	Multiple clients can connect to one server process.

Object space size	Maximum of 384 MB per database. This is a technical limit of the low-level persistent storage used for the server.
Access control	Login deny/allow on per-machine basis
Lock protocol	Hierarchical locking.
Lock levels	Whole storage, object, slot.
Lock modes	Read-only, read, write; read-only intent, read intent, write intent
Transactions	Flat transactions. Concurrent transactions are supported, both between different clients and between different database sessions of a single client.
Access protocol	Two phased transactions and locking of the persistent object at object state transfer between client and server.
Indexes supported	Btrees on slots of persistent CLOS instances are maintained automatically; direct usage of btrees is possible.
LISP types	All types defined in [CLtLII 1990] are supported, with the exception of binary function code.
Name spaces	Persistent packages containing persistent symbols.

Chapter 5

Administration

This chapter describes how to startup PLOB! and how the PLOB! server is administrated.

5.1 Administrating the client

5.1.1 Starting on the client side

For an example of a startup file see `plob-2.11/src/example/plob-toplevel.lisp`. To load all PLOB! files at once after installation is complete, load the file `defsystem-plob` and evaluate `(load-plob)`. Unless a session to the server is explicitly opened by a call to function **open-my-session** [API Manual 1997, p. 45], a session will be opened when necessary.

If PLOB! should write more or fewer messages about its proceeding to stream **t**, set variable ***verbose*** [Internal Manual 1997, p. 262] to an appropiate value.

5.1.2 Finishing on the client side

All sessions opened to the server will be closed when the LISPWORKS or ALLEGRO process will terminate. Calling generic function **close-heap** [API Manual 1997, p. 28] will force a close of all sessions at runtime. If there are no other clients connected to the server, closing the last client will trigger the garbage collection.

5.1.3 Creating a new database

New databases on the server can be created from the client's side either by calling function **p-create-database** [API Manual 1997, p. 61], or by calling function **open-my-session** [API Manual 1997, p. 45] and passing the URL of the new database as argument. If the second approach is used, the user is asked to confirm that a new database should be created. For example, this will create and open a database on host `ki8` named `my-base`:

```
CL-USER 79 > (open-my-session "//ki8/my-base")
;;;; Bootstrap   : Opening tcp://ki8/my-base
Error: cplob.c(152): fnOpen:
       Could not open database tcp://ki8/my-base
  [condition type: POSTORE-CERROR]
```

```
Restart actions (select using :continue):
 0: Try to create it.
;; ...
[1] CL-USER 80 : :cont
```

5.2 Administrating the server

For administrating the server process, the `plobdadmin` program is provided within the distribution. The PLOB! administrator should either change her/his `PATH` environment variable to include the `plobdadmin` program found in the operating system specific configuration directory `plob-2.11/conf/<operating system>/plobdadmin[.exe]` or put a (symbolic) link to it into one of the directories found in the `PATH` variable. For doing the administration, the PLOB! administrator must execute the `plobdadmin` program on the machine which runs the server process. The server process is started with the effective user-ID of the PLOB! administrator. If at configuration the client authentication is set to `AUTH_DES`, the server must be started with root rights.

For usage of `plobdadmin`, see its help text obtained by calling `plobdadmin -h`. When `plobdadmin` is called without options, it starts up in interactive mode, that means it prints a prompt and awaits input from `stdin`. Since `plobdadmin` is linked against the RPC client shared library of PLOB, `librpclientplob.[so,dll]` must be found in the `LD_LIBRARY_PATH` (for UNIX) or `PATH` (for Windows XP).

5.2.1 Starting and stopping the server

Normally, the server needs only be started when the machine running the server was rebooted, since the server does not terminate itself when the last client disconnects. If a restart is necessary, login to the server host (if the server host is not `localhost`) and change to the database root directory. Call program `plobdadmin -connect`, this will start the server process.

Calling program `plobdadmin -stop` will stop the server process. To get rid of orphan sessions (that means, sessions whose client process has been lost for some reasons), do a `plobdadmin -restart`.

5.2.2 The PLOB! administrator

The user which opens the stable heap the very first time is made the PLOB! system administrator. The administrator is allowed to reset (function **p-reset** [API Manual 1997, p. 82]), restart (function **p-restart** [API Manual 1997, p. 82]) or exit (function **p-exit** [API Manual 1997, p. 65]) the daemon process from her or his LISP listener and – a little bit more important – may allow or deny logins from client machines (function **(setf p-machine-loginp)** [API Manual 1997, p. 74]). The administrator is identified by her or his login name and the machine on which the client process runs; the administator which is accepted by the server can be resetted either by calling the function **p-reset** [API Manual 1997, p. 82] by the (current) administrator or by using the `reset` subcommand of the `plobdadmin` script. If the adiministrator is resetted, the next user which opens the heap will become the new administrator.

5.2.3 Remote daemon control

As mentioned already, the administrator has remote access for administrating the daemon. Function **p-reset** [API Manual 1997, p. 82], function **p-restart** [API Manual 1997, p. 82] and function **p-exit** [API Manual 1997, p. 65] may be used by the administrator in a listener to administrate the daemon. The name and host of the administrator of a persistent heap is stored within the persistent heap itself; in consequence, for calling function **p-reset** [API Manual 1997, p. 82], the administrator's listener process must be connected to the persistent heap which should be resetted. After the daemon has been terminated by a call to function **p-exit** [API Manual 1997, p. 65], it can only be restarted from a shell by calling program `plobdadmin` with a `-connect` argument.

When configured correctly, function **p-admin** [API Manual 1997, p. 45] can be used to call script `plobdadmin` from within a listener. Prerequisite is that `rsh` will work on the host running the LISP system.

References Function **p-restart** [API Manual 1997, p. 82], function **p-exit** [API Manual 1997, p. 65], function **p-reset** [API Manual 1997, p. 82].

5.2.4 Login administration

References Function **(setf p-machine-loginp)** [API Manual 1997, p. 74], function **show-machines** [API Manual 1997, p. 116].

5.2.5 Hot Backup

Starting from version 2.11, hot backup of the database files is supported. It works by (logically) suspending the server from working for the time of copying, i.e. the clients requests are stopped for the time the copy takes place. Since copying should only involve a few seconds, the clients will hardly notice it. A proposal for the overall proceeding for doing a hot backup is as follows:

- Start the `plobdadmin` administration tool in the database root directory:

  ```
  /opt/data/plob/>./plobdadmin
  ```

- In the administration tool, connect to the database' server, suspend it, call a shell script to save the database directory, and resume the server:

  ```
  tcp://localhost/database>connect database
  tcp://localhost/database>suspend daily backup
  tcp://localhost/database>! <a shell script for backup>
  tcp://localhost/database>resume
  tcp://localhost/database>quit
  ```

  The string after the suspend command is a comment which is delivered to the waiting clients explaining why the server was suspended. The shell script should copy all files contained in the database directory, not only the `stablestore` file.

- `plobdadmin` can also be called non-interactive by providing a script file name, e.g.:

  ```
  /opt/data/plob/>./plobdadmin -source <a plobdadmin script>
  ```

For suspending and resuming the server, client-side LISP function calls are also available. A suspended server does not accept any calls except calls to suspend (which will then simply return an information about the suspend status) and resume. All other calls will block, also calls raised e.g. for printing objects, since these calls incure a database access.

References Function **p-suspend** [API Manual 1997, p. 95], function **p-resume** [API Manual 1997, p. 82].

5.2.6 Restoring a database

For a restore, the server process should be stopped (and therefore all clients get disconnected), the database files should be copied back into the database directory, and the server should be restarted afterwards. A hot restore is not supported, i.e. suspending the server and replacing the database files by a backup copy gives unexpected results, since the server process and its clients rely on their associated database not to change its state unexpectedly.

5.3 Running PLOB! serverless

It is possible to run PLOB! with no server at all. In serverless mode, the server's code is directly loaded into the current LISP client image. The following sections describe how to start PLOB! in serverless mode. The sections after the next section describe the main differences between using PLOB! in server mode and in serverless mode.

5.3.1 Starting PLOB! in serverless mode

To start PLOB! in serverless mode, specify a protocol name of `local` followed by a database directory name within the database URL to open. For example:

```
CL-USER 81 > (open-my-session "local:c:/home/kirschke/database")
;;;; Bootstrap   : Opening local:c:/home/kirschke/database
;;...
```

This will open the database located in directory `c:/home/kirschke/database`; if this database does not exist, you are asked if it should be created. When using a Windows XP filename, the protocol name `local` must be specified if the database to open contains a drive letter; otherwise, PLOB! will mistakenly interpret the drive letter as a protocol name.

After the very first database has been opened in server mode or serverless mode, it is not possible to switch from one mode to the other. The LISP system must be restarted to switch between both modes.[1]

5.3.2 Differences between server mode and serverless mode

To understand the differences, some technical background is given. In server mode, the LISP client process communicates with the server process by RPCs; this is a request-reply serial communication channel resulting in a rather weak coupling. This means for example, if one of these processes abends, the other process will live on; so, the other process still has a chance to save its data.

In serverless mode, PLOB's server functionality is directly linked into the current LISP image and running within the same process, resulting in a rather tight binding. All resources are shared between LISP and PLOB; this poses a problem e.g. on Linux, since LISP and PLOB! can not agree on how to share the address space between them. Also, failures of one of both subsystems leading to a process abend will abend the other subsystem, too.

One reason for using the serverless mode is performance, since serverless mode incures much fewer communication overhead than server mode. In serverless mode,

[1]This has the technical reason that a shared library cannot be unloaded safely from a running LISP image.

PLOB's functions are running about twice as fast as in server mode, some functions are even more faster. Another advantage is that the locking algorithm of PLOB! now will truly suspend LISP threads waiting for a lock to be granted. In server mode, this is done by polling since all RPC communication can only be initiated by the client.

Of course, the serverless mode needs no server administration at all, especially no active PLOB! server is needed. For PLOB's API, there are almost no differences between both modes; only some administration functions (like function **p-exit** [API Manual 1997, p. 65]) have been changed, since they make no sense in serverless mode.

5.4 Common error messages

This section explains the most common error messages and what to do when receiving such a message.

Error message In the LISP listener, following message is raised:

```
Error: cplob.c(383): fnClientCreate:
       Connect to host ki8.kogs.hh failed:
       localhost: RPC: Remote system error - Connection refused
       (Check if the PLOB! daemon is running on the host)
```

What happened? The daemon has not yet been started.

To Do Ask the PLOB! administrator to start the daemon on the server machine by using program `plobdadmin`, subcommand `open`; meanwhile (1998/05/07), it is also possible to start the daemon directly without program `plobdadmin`. A start can also be done by using function **p-admin** [API Manual 1997, p. 45] (section 'Remote daemon control' on page 70) at the debugger's prompt, for example:

```
;;;; Bootstrap   : Opening tcp://localhost/plob
Error: cplob.c(383): fnClientCreate:
       Connect to host ki8.kogs.hh failed:
       localhost: RPC: Remote system error - Connection refused
       (Check if the PLOB! daemon is running on the host)
Restart actions (select using :continue):
;; ...
[1] CL-USER 82 : (p-admin :open)
0        ;; Error code returned from remote called shell script plobdadmin
[1] CL-USER 83 : :cont
```

Error message In the LISP listener, following message is raised:

```
Error: cplob.c(371): fnClientCreate:
       Connect to host ki8.kogs.hh failed:
       ki8.kogs.hh: RPC: Miscellaneous tli error -
       An event requires attention
```

What happened? The daemon is not running, although the client expected a running daemon. Probably, there is a communication problem, or the daemon terminated due to an internal failure.

To Do Ask the PLOB! administrator to look for the failure condition. The daemon can be started again on the server machine by using the script `plobdadmin`, subcommand `open`. A start can also be done by using function **p-admin** [API Manual 1997, p. 45] (section 'Remote daemon control' on page 70).

Error message In the LISP listener, following message is raised:

```
Error: It looks as if the LISP PLOB root object should be formatted.
```

What happened? You opened the stable heap the very first time; the LISP root object was not found in the stable heap.

To Do Acutally, this is no error but an emergency break to escape from a maybe unwanted formatting of the stable heap. Normally, do a `:continue` and everything should work fine.

Error message When starting the `plobd` server process using `plobdadmin -connect` from the shell, the following error message is shown:

```
Cannot register service: RPC: Unable to receive; errno = Connection refused
unable to register (PLOBD, PLOBDVERS, udp).
```

What happened? The `portmap` (for Solaris: `rpcbind`) daemon is not running; this daemon is part of the operating system and no part of PLOB!

To Do Start the `portmap` (for Solaris: `rpcbind`) daemon on your machine as root user. The `portmap` (for Solaris: `rpcbind`) daemon should be located in `/usr/sbin`.

Error message In the LISP listener, following message is raised:

```
Error: cplob.c(147): fnCreateAuth:
       Creating the AUTH_DES authentication failed. Check if
       ‘ki8.informatik.uni-hamburg.de’ names a host in your
       net domain and that keyserv is running on your local
       client machine. Check if calling the shell command
       ‘keylogin’ might help.
```

What happened? The client could not create its authentication data.

To Do If issuing a ‘keylogin’ on the shell prompt does not help, the server name in parameter ***database-url*** [API Manual 1997, p. 30] is perhaps not what you mean: If PLOB! was compiled with `AUTH_DES` authentication, the parameter ***database-url*** [API Manual 1997, p. 30] must contain a host name and a domain name (for example `"ki8.kogs.hh"`) of your local net domain and *not* a fully qualified Internet address.

Error message In the LISP listener, following message is raised:

```
Error: From server at executing client:fnSHopen:
       Login to stable heap failed.
```

What happened? The daemon process is running, but denied to log you into the stable heap.

To Do Ask the PLOB! administrator to view into the `messages.log` file associated with the daemon to see the reason why the login was denied. This file is located on the server in the database directory. Perhaps the PLOB! administrator has denied the login for your machine.

Bibliography

[AMOP 1991] Gregor Kiczales, Jim des Rivières, and Daniel G. Bobrow. *The Art of the Metaobject Protocol*. The MIT Press, Cambridge, Mass., 1991.

[API Manual 1997] Heiko Kirschke. *Persistent LISP Objects: Application Interface Reference Manual*, November 1997. Local copy.

[Atkinson et al. 1992] Malcolm Atkinson, Françoise Bancilhon, David DeWitt, Klaus Dittrich, David Maier, and Stanley Zdonik. The Object-Oriented Database System Manifesto. In Françoise Bancilhon, Claude Delobel, and Paris Kanellakis, editors, *Building an Object-Oriented Database System*, pages 3–20. Morgan Kaufmann Publ. Inc., San Mateo, CA, 1992.

[Beckmann et al. 1990] Norbert Beckmann, Hans-Peter Kriegel, Ralf Schneider, and Bernhard Seeger. The R*-tree: An efficient and robust access method for points and rectangles. In Hector Garcia-Molina and H. V. Jagadish, editors, *Proceedings of the 1990 ACM SIGMOD International Conference on Management of Data*, Atlantic City, NJ, June 1990, volume **19** (2), pages 322–331.

[Berchtold et al. 1996] Stefan Berchtold, Daniel A. Keim, and Hans-Peter Kriegel. The X-tree: An Index Structure for High-Dimensional Data. In T. M. Vijayaraman, Alejandro P. Buchmann, C. Mohan, and Nandlal L. Sarda, editors, *VLDB'96, Proceedings of 22th International Conference on Very Large Data Bases*, Mumbai (Bombay), India, September 1996, pages 28–39. Morgan Kaufmann Publ. Inc., San Francicso, CA. Remote copy.

[CLtLII 1990] Guy L. Steele Jr. *Common LISP the Language*. Digital Press, Bedford, MA, 2 edition, 1990.

[Codd 1979] E. F. Codd. Extending The Database Relational Model to Capture More Meaning. *ACM Transactions on Database Systems* **4** (4), 397–434, December 1979.

[Gray & Reuter 1993] Jim Gray and Andreas Reuter. *Transaction Processing: Concepts and Techniques*. Morgan Kaufmann Publ. Inc., San Francicso, CA, 1993.

[Installation Guide 1997] Heiko Kirschke. *Persistent LISP Objects: Installation Guide*, November 1997. Local copy.

[Internal Manual 1997] Heiko Kirschke. *Persistent LISP Objects: Internal Interface Reference Manual*, November 1997. Local copy.

[Kirschke 1994] Heiko Kirschke. Persistenz in objekt-orientierten Programmiersprachen am Beispiel von CLOS. Master's thesis, Fachbereich Informatik, Universität Hamburg, October 1994.

[Kirschke 1995] Heiko Kirschke. Persistency in a Dynamic Object-Oriented Programming Language. Technical report, Fachbereich Informatik, Universität Hamburg, Hamburg, August 1995. Local copy , remote copy.

[Kirschke 1999] Heiko Kirschke. Implementing Persistency in Common LISP. Technical report, European LISP User Conference, Amsterdam, May 1999. Local copy , remote copy.

[LispDoc Manual 1994] Heiko Kirschke. *LispDoc Manual*, 1994. Local copy.

[Müller 1991] Rainer Müller. Sprachprozessoren und Objektspeicher: Schnittstellenentwurf und -implementierung. Master's thesis, Fachbereich Informatik, Universität Hamburg, 1991.

[ODMG-2 1997] R. G. G. Cattell and Douglas K. Barry, editors. *The Object Database Standard: ODMG 2.0.* Morgan Kaufmann Publ. Inc., San Francicso, CA, 1997.

[O'Neil 1994] Patrick O'Neil. *Database Principles, Programming, Performance*. Morgan Kaufmann Publ. Inc., San Francicso, CA, 1994.

Appendix A

Architecture of PLOB!

For a detailed description of PLOB!'s architecture see [Kirschke 1999].

In its first version up to September 1996, PLOB! was a single-user system prepared for multi-user capabilities. At end of 1996, the system was redesigned into a client/server architecture.

Client The client is the LISP process using persistent objects. A client is connected to no or one server. At the moment, LISPWORKS Common LISP and ALLEGRO Common LISP processes are supported as clients.

Server The server is a UNIX process serving requests from the clients which connected itselves to the server. A server can handle many clients.

Figure A.1 shows the layers of PLOB! as actually implemented.

Based upon an evaluation of a few persistent memory systems done by [Müller 1991], it was decided to use the POSTORE library (Persistent Object Store) as low level store. The layers 2–5 have been implemented by the author, layers 2 and 3 in ANSI-C and layer 4 and 5 in Common LISP. The communication between layers 2 and 3 is done by RPCs (Remote Procedure Calls), an approved UNIX standard communication tool.

A.1 Layer 1: POSTORE layer

POSTORE is the persistent memory used in Napier88, a persistent programming language developed at the University of St. Andrews. This layer offers a persistent heap, a concept very similar to a transient heap, but the 'records' allocated from a persistent heap are persistent. Each POSTORE vector is subdivided into a first part interpreted as having references to other POSTORE vectors and a second part which is not interpreted by POSTORE any further; the second part is used to store the state of value-like objects, that means, objects not referencing other objects. This is used by POSTORE to implement a garbage collection based on reachability.

The POSTORE has no usable transaction processing in the sense of a database. The locking offered by POSTORE is a simple semaphore locking on a per-object basis.

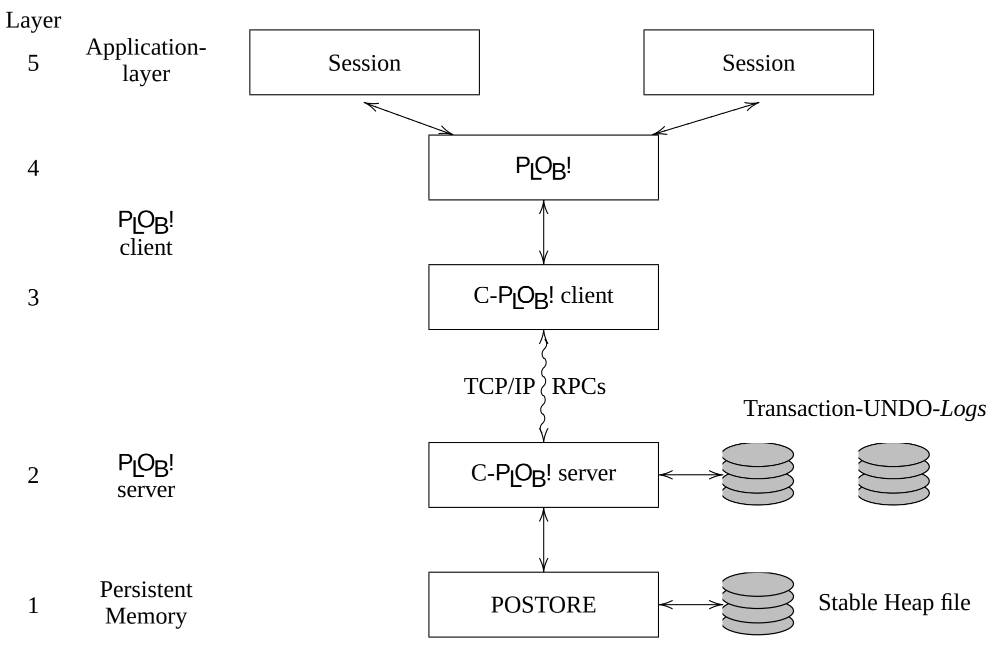


Figure A.1: PLOB! layers

A.2 Layer 2: C-PLOB! server layer

This layer adds a whole bunch of functionality to the system: Sessions, object locking and transactions, persistent B-Trees. Layer 1 and 2 make up the PLOB! server process.

A.3 Layer 3: C-PLOB! client layer

The task of this layer is to interface between the previous and the next layer. Besides this, it contains the code for connecting the client with the server process, implements some caching and pre-allocation of persistent objects etc. A part of the functionality of layer 2 is used directly on the client side by layer 3 to reduce the number of RPC calls to the server, for example, the print representation for persistent immediates is computed on the client side. Layer 3 up to layer 5 make up the PLOB! client process.

A.4 Layer 4: PLOB! layer

This layer implements an interface to the application layer for convenient usage of persistency. Its functionality is documented in this user's guide.

A.5 Layer 5: Application layer

The application layer uses the functionality offered by the previous layer: Persistent packages, persistent symbols, sessions, associative access by using persistent btrees.

A.6 Porting PLOB!

This section describes how to port PLOB! to a new, not-yet-supported operating system and to a new, not-yet-supported LISP system.

A.6.1 New Operating System

This section assumes that the port of the LISP system itself is already done and running stable. The PLOB!-specific tasks are described in the following sections.

Skills needed

The port to a new operating system involves mainly a successfull compilation of the low-level POSTORE library. This means that the tasks to be done have to do more or less almost nothing with LISP coding, but with operating-system specific programming in form of shell scripts, makefiles, and C system programming. The following skills are needed for a port to a new operating system:

- Good knowledge of `make` and `make` scripts
- Good knowledge of shell programming, esp. Bourne shell
- Good knowledge in C, and the specialities of the C preprocessor.
- Good knowledge about the virtual memory functions (`mmap`) and signal handling capabilities (`signal`, `sigaction`) of the target operating system.

Step 1: Check out from CVS

Check out PLOB!'s sources from

```
:pserver:anonymous@cvs..sourceforge.net:/cvsroot/plob
```

This will create a subdirectory `plob` containing the checked-out files.

Step 2: Select an identifier

Select an identifier for the operating system, e.g. `win32`. The identifier has to obey the rules for a single C token and has to be in lower case, i.e. the syntax is:

<opsys> ::= [a-z]{[a-z0-9]}*

Step 3: Build the POSTORE library

In the next step, the POSTORE library will be adapted to the target operating system. To do this, a high skill level in system-near programming in C in necessary, esp. on the virtual memory functions of the target operating system. The sources of the POSTORE library are located in directory `plob/postore`. This library is a 3rd party product used as a persistent heap. Only some general rules can be given here, since the differences between the operating systems might be quite large.

First step in compiling the POSTORE library is to create the operating system specific configuration file in `plob/postore/<opsys>_Makefile`. Use one of the exisiting `<opsys>_Makefiles` as base, e.g. the Linux file in `plob/postore/i586_Makefile`.

Next step in compiling the POSTORE library is to create the compiler specific configuration file in `plob/postore/${CC}_Makefile` with `CC` being set in the operating system specific configuration file. Use one of the exisiting `plob/postore/${CC}_Makefiles` as base, e.g. the gcc file in `plob/postore/gcc_Makefile`.

Now, edit `plob/postore/sstore/svmstore.c` to meet the target operating system. The source code contains a lot of operating-system specific code steered by preprocessor defines. The biggest challenge is to find a suitable virtual memory mapping function `mmap` in the target system, and to check for the various flags to be passed into `mmap`. Another challenge is to set up the signal handling (`sigaction`, `signal`) for passing page faults to the page fault handling procedure `page_fault`. Besides `plob/postore/sstore/svmstore.c`, the persistent heap code itself is contained in `plob/postore/sheap/sheap.c`; this should need no special adaption for a specific operating system.

The library and its supporting programs can be build by calling `make` in directory `plob/postore`. When the build is successfull, try calling `plob/postore/release/<opsys>bin/sstoreforr`
It takes a directory name as argument and tries to initialize that directory as a stable store directory. If this work, the POSTORE library port was successfull, otherwise some debugging is necessary. An unsuccessull compilation is indicated by error messages like 'unexpected page fault', in this case the `page_fault` procedure and signal handling in `svmstore.c` needs some more refinement.

Step 4: Check RPC capabilities

The target operating system should support standard SUN RPCs (RPC: RFC1057, XDR: RFC1014), this is e.g. the case for almost all *nix variants (this mechanism is also sometimes called ONC). If the operating system does not support standard SUN RPCs, subdirectory `plob/oncrpc-1.12` contains an implementation of these RPCs and could be adapted accordingly. Standard SUN RPC support can be detected by calling `rpcgen` in a shell. If that program is found, the probability of having SUN RPCs on the target machine is quite high.

Step 5: Adapt plob/conf/make.rules.in

File `plob/conf/make.rules.in` contains operating-system specific rules for `make`. For the target operating system, the file suffixes should be checked (see comment 'File name suffixes:'). The operating system and compiler specific settings have to be checked (see comment 'OS/compiler-specific settings:'). Add the <opsys> identifier to file `plob/conf/make.rules.in` to the line starting with `ALLOPSYS=`.

Step 6: Generate Makefiles

In a shell with current directory `plob/src`, call `./mk makefiles`. This will (re-)build all makefiles for the target operating system. If this fails, the rules in file `plob/conf/make.rules.in` should be checked. Re-iterate this step until the makefiles have been build successfully.

Step 7: Generate binary directories

In a shell with current directory `plob/src`, call `./mk initial`. This will create all target system specific subdirectories for placing binary, target system specific code. It should be sufficient to do this call only once for the target system.

Copy in the POSTORE library into the PLOB! binary directory. This has to be done manually by copying `plob/postore/release/<opsys>lib/libpostore.a` to `plob/lib/<opsys>/libpostore.a`.

Step 8: Building PLOB!

In a shell with current directory `plob/src`, call `./mk`. This will create all target system specific binaries. Some of the source codes contain preprocessor directives for adapting the #includes to the different operating systems, this might need some adjustment. File splobadmin.c also contains some signal handling code.

Step 9: Adapt plob/src/lisp/defsystem-plob.lisp

The last step is to extend plob/src/lisp/defsystem-plob.lisp by the new supported operating system, mainly to allow defsystem to map to the correct foreign function libraries to be loaded. See the comments in plob/src/lisp/defsystem-plob.lisp for further details.

Not supported operating systems

A port to HP-UX is not possible due to a mismatch between the virtual memory functions needed by PLOB!'s low-level POSTORE library and the actual implementation of these functions in HP-UX.

A.6.2 New LISP System

This section assumes that the port of the LISP system itself is already done and running stable. The PLOB!-specific tasks are described in the following sections. Another assumption is that PLOB!'s binaries for layers 3 and below (figure A.1) have been successfully build already (maybe for another LISP system). The following skills are needed for a port to a new LISP system:

- Moderate knowledge of make and make scripts
- Moderate knowledge of shell programming, esp. Bourne shell.
- Excellent knowledge of the foreign function interface of the target LISP system.
- Excellent knowledge of the Metaobject Protocol as implemented in the target LISP system.
- Experience and inspiration on debugging self-reflective system like CLOS for debugging PLOB!'s MOP binding.

Step 1: Check out from CVS

Check out PLOB!'s sources from

```
:pserver:anonymous@cvs.sourceforge.net:/cvsroot/plob
```

This will create a subdirectory plob containing the checked-out files.

Step 2: Select an identifier

Select an identifier for the target LISP system, e.g. cmucl. This identifier has to be in lower case. It should contain no suffix number, since the version number of the LISP system will be added afterwards. The identifier has to obey the rules for a single C token, i.e. the syntax is:

<lispsys> ::= {[a-z]{[a-z0-9]}*}+[a-z]

In the following text these conventions are used:

<Lispsys> means <lispsys> with its first letter capitalized

<LISPSYS> means <lispsys> with all of its letters capitalized

Step 3: Adapt plob/conf/make.rules.in

File `plob/conf/make.rules.in` contains target LISP system specific rules for generating the foreign function interface source code. Look at the MkAllegro*Files and MkLispWorks*Files and generate the corresponding Mk<Lispsys>*Files macros.

Step 4: Adapt plob/bin/c2lisp.h

Towards the end of file `plob/bin/c2lisp.h`, add the target LISP system specific include file as indicated there (in this example, for version 3 and 4 of the target LISP system):

```
#if defined(LISPWORKS3)
#include <c2lispworks3.h>
#elif defined(LISPWORKS4)
#include <c2lispworks4.h>
#elif defined(ALLEGRO4)
#include <c2allegro4.h>
#elif defined(ALLEGRO5)
#include <c2allegro5.h>
#elif defined(ALLEGRO6)
#include <c2allegro6.h>
#elif defined(ALLEGRO7)
#include <c2allegro7.h>
#elif defined(<LISPSYS>3)
#include <c2<lispsys>3.h
#elif defined(<LISPSYS>4)
#include <c2<lispsys>4.h>
#elif defined(another_lisp_system)
#include <c2another_lisp_system.h>
#else
#error Missing target LISP system.
#endif
```

Step 5: Create plob/bin/c2<lispsys>.h

Create the file containing the foreign function code generator macros; base this file on one of the existing `plob/bin/c2allegro*.h` or `plob/bin/c2lispworks*.h` files.

Step 6: Adapt plob/src/include/makefile.in

In `plob/src/include/makefile.in`, the rules for creating the LISP specific directory containing the foreign function interface code have to be added.

Extend rule with target `initial` with a dependency on the target LISP system directories; extend the existing rule:

```
drule ( initial, \
        allegro dash directories lispworks dash directories \
        <lispsys> dash directories , \
        noActions )
```

Add a new rule for creating the version specific target LISP system directories (in this example, for version 3 and 4 of the target LISP system):

```
rule ( <lispsys> dash directories, \
       <lispsys>3 <lispsys>4, \
       noActions )
```

Add the rule for creating the directories:

```
rule ( <lispsys>3 <lispsys>4, \
       noDependencies, \
       mkdir ruleTarget )
```

Add a preprocessor macro for the new set of foreign function interface source code files, e.g.

```
/* ----------------------------------------------------------------------
| LISP foreign language interface files: <lispsys>
 ---------------------------------------------------------------------- */
#define <Lispsys>CodeTarget <lispsys> dash code
```

Extend the rule with target `lisp` with a dependency on the target LISP foreign function code files; extend the existing rule:

```
rule ( lisp, \
       HarlequinCodeTarget AllegroCodeTarget <Lispsys>CodeTarget , \
       noActions )
```

Behind that rule, add the file generating macros defined in `plob/conf/make.rules.in` (in this example, for version 3 and 4 of the target LISP system):

```
Mk<Lispsys>3aFiles
Mk<Lispsys>3bFiles
Mk<Lispsys>3cFiles
Mk<Lispsys>3dFiles

Mk<Lispsys>4aFiles
Mk<Lispsys>4bFiles
Mk<Lispsys>4cFiles
Mk<Lispsys>4dFiles
```

Add the rules for building the target LISP system foreign function interface source code (in this example, for version 3 and 4 of the target LISP system):

```
<LISPSYS>3FILES=\
PlobFiles(<lispsys>3 slash,dot lisp)
<LISPSYS>4FILES=\
PlobFiles(<lispsys>4 slash,dot lisp)
<LISPSYS>FILES=$(<LISPSYS>3FILES) $(<LISPSYS>4FILES)
rule ( <lispsys> dash code, \
       $(<LISPSYS>FILES), \
       noActions )
```

Step 7: (Re-)generate foreign function interface sources

In a shell with current directory `plob/src/include`, call `./mk <lispsys>-code`. This will (re-)generate the foreign function interface source code for the target LISP systems.

Step 8: Adapt plob/src/lisp/defsystem-plob.lisp

Adapt the definitions of parameters `+plob-include-lisp-directory+` and `+plob-target-lisp-directory+` to capture the new target LISP system. Load `defsystem-plob.lisp` and check if the logical pathname transformations work as expected (e.g. by looking at constant `+plob-members+`).

Step 9: Extend plob/src/lisp/ff-mapping.lisp

The challenging task is to extend `plob/src/lisp/ff-mapping.lisp` by the macro definitions suited for the target LISP system, namely the macros `define-foreign-function` and `define-foreign-callable`. Macro `define-foreign-function` has to expand into a definition which describes to the target LISP system a foreign function to be called by the target LISP system. Macro `define-foreign-callable` has to expand into a definition which describes to the target LISP system a foreign function which will call a LISP function of the target LISP system, e.g. for error callbacks.

Step 10: System specific code

LISP system specific code is also to be found in `plob-sysdep.lisp` and has to be adapted to the target LISP system.

Step 11: Metaobject Protocol

PLOB! uses the Metaobject Protocol generally as described in [AMOP 1991] and more specifically as implemented in the target LISP system (be assured, that *makes* a difference). Both LispWorks Common LISP and Allegro Common LISP are relatively close to the definition given in [AMOP 1991]. Most of the problems with the MOP binding stem from the non-compatible parts of the implementation (which are often not documented), often for performance reasons, and from problems about run-time stability of a self-reflective system as CLOS actually is.

As a consequence, only some rules of thumb can be given to check the MOP binding. Assumingly, a new target LISP system might exhibit a behavior similar to that of one of the currently supported systems, so it makes sense to look for the conditional statements especially in files `plob/src/lisp/plob-clos-slot-value.lisp`, `plob/src/lisp/plob-clos-` and `plob/src/lisp/plob-metaclass-*.lisp`, and read the comments in these files.

A.6.3 New Operating and LISP System

When considering doing a port both to a new operating system and a new LISP system, my recommendation is to start with the port of the LISP system running on an operating system currently supported by PLOB!, and do the operating system specific port afterwards. The reason is that the operating system specific port involves operating system specific low-level functions (esp. virtual memory functions); testing a new operating system port becomes a lot easier when the tests can be done from a running application level.

Appendix B

Release Notes

This section contains the release notes and describes the known bugs of PLOB!. Further features can be found in [Kirschke 1994].

B.1 Release Notes

This section describes the changes done between the different releases of PLOB!

B.1.1 Release 2.11

Released May 1, 2005.

- Support for LISPWORKS Common LISP 4.4 and ALLEGRO Common LISP 7 has been added.
- Hot backup possible by server suspending and resume, improved `plobdadmin` tool with support for hot backup, additional shell escape command
- Added regular expressions for comparing persistent strings and filtering on Btree iterators
- Added example code in `plob-dupkeys.lisp` for handling of duplicate keys in btrees
- `plobdadmin` is now statically linked
- The LISP sources are no longer divided by the different LISP systems
- LISPWORKS Common LISP fasl files are now written to an extra directory

Improved compare functionality

The compare functionality used e.g. for searching in Btrees has been improved. Now, keys composed of lists and vectors can be used as long as their structure is equal.

Bugfix on transaction rollback

The transaction log files have been written to wrong when exceeding the internally used block size of 64K, this has been fixed. All operating system versions are affected by this bug and its fix.

Shutdown handling for DLLs on Windows

The client DLLs on Windows suffered from an incorrect shutdown handling which resulted in PLOB!'s state being logically terminated although still being in use. This has been fixed.

ALLEGRO Common LISP 7, LISPWORKS Common LISP 4.4

The latest LISP versions are now supported.

Virtual memory management on Windows

Although the memory management functions of Unix and Windows are quite similar, there are subtle differences between these. One difference leading to an instable behavior of PLOB! on Windows is about what is considered as free memory by the operating system. To my understanding, Unix uses a continuous memory region between a minimum and a maximum address per process. All memory above the maximum address is considered as free memory. On Unix, this free memory is used by PLOB! to map the database file into memory. On Windows, there is no free memory in this sense, i.e. there is no designated maximum address. Instead, all memory is administered in blocks, which can be free or allocated. Before version 2.11, PLOB! used the first of these free blocks returned by the file mapping call to Windows; especially, this could be a rather small block spanning only a very small memory region. This means that when the memory needed became larger, the block might be used up. In this case, PLOB! signalled errors of the kind 'cannot mmap memory'. In version 2.11, PLOB! now looks for a free memory block spanning at least 384 MB and uses this block.

This bug and the fix adheres only to the Windows version of PLOB!, the other versions are not affected by this behavior.

As a consequence, the overall database size of PLOB! has been reduced from 1 GB to 384 MB, since the possibility of finding a free 1 GB memory region on Windows is quite low. Unfortunately, older versions of Windows databases cannot be opened by version 2.11 any more.

Testing

This release of PLOB! was tested on the following platforms.

OS	LISP version	Date	Tester	Remarks
Windows	ALLEGRO Common LISP 7	2005-04	Kirschke	plob-example
Windows	LISPWORKS Common LISP 4.3	2005-04	Kirschke	plob-example
Windows	LISPWORKS Common LISP 4.4	2005-05	Kirschke	plob-example
Linux	ALLEGRO Common LISP 7	2005-04	Kirschke	plob-example
Linux	LISPWORKS Common LISP 4.4	2005-05	Kirschke	plob-example
Solaris	ALLEGRO Common LISP 7	2005-05	Kirschke	plob-example

Known bugs

Multithreading access from a single LISPWORKS Common LISP process on Windows does not work, it raises runtime exceptions.

B.1.2 Release 2.10

Released May 1, 2002.

- Support for Windows XP, LISPWORKS Common LISP 4.2 and ALLEGRO Common LISP 6.2 has been added.
- Further support for IRIX will be on request. Contact me if you are interested in an IRIX port.
- The sources are now maintained within CVS. Currently, I try to set up a CVS archive at SourceForge and a possibility for anonymous ftp.

Allegro Common LISP 6.0

2001-02-06: Release 2.10 supports Allegro Common LISP 6.0.

B.1.3 Release 2.09

Released May 22, 2000.

Multiple servers

The code for running multiple servers on a single host has been fixed. Prior to 2.09, the RPC version number was used to discriminate between different databases. The ONC RPC documentation says that only for adjacent version numbers the portmap daemon ensures a correct mapping to the registered server. Since PLOB! did not obey this rule, the success of connecting to a database not being the master database was more or less given by chance.

LispWorks Common LISP on Linux

2000-05-22: LISPWORKS Common LISP 4.1 on Linux is now supported. Although not tested, LISPWORKS Common LISP 4.1 on Solaris should work, too. LISPWORKS Common LISP 4.1 on Linux and Solaris have the restriction that strings cannot exceed the length of `plob::+max-ff-string-length+`.

B.1.4 Release 2.08

Released December 1, 1998.

64 bit proof

1998-11-05: The code is now 64 bit proof; the btree interface and code had to be changed a lot, since this code was written with the assumption that `sizeof ( int ) == sizeof ( void* )`; this has been fixed. A pointer can now have any size.

Bignums for ACL 5.0

1998-11-19: Bignums are now handled correctly for ACL 5.0.

Rewritten plobdadmin utility

1998-11-25: The `plobdadmin` has been rewritten in C. Its source code can be found in `plob-2.11/src/admin/`.

Heterogenous architectures for client and server

The client/server code has been fixed; arguments are now marshalled correctly for all client/server combinations of heterogenous architectures. The 'values' section of a postore vector is now transferred along with a type tag, so that marshalling can select the right procedure.

B.1.5 Release 2.04

Parameter *verbose*

1998-02-06: Setting variable ***verbose*** [Internal Manual 1997, p. 262] to `nil` or 0 will continue all non-fatal `cerrors` without any user intervention.

Windows/NT support

1998-02-26: The server `plobd.exe` is running under Microsoft Windows/NT 4.0. 1998-04-15: PLOB's client code for LISPWORKS Common LISP 4.0.1 is running under Windows/NT 4.0. Fixed final release on 1998-05-05.

Btrees

1998-03-10: Constant **+plob-min-marker+** [API Manual 1997, p. 109] and constant **+plob-max-marker+** [API Manual 1997, p. 109] refer now to the first and last existing object in a btree, and no longer to the object 'before' or 'behind' the first or last existing object, as it was before. The page size of (empty) btrees can now be changed. 1998-05-05: Fixed code for inspecting btrees in LISPWORKS Common LISP. 1998-06-25: BTrees can now be iterated in ascending and descending order. The intervals for a search request can now be specified as open and closed intervals.

Built in types representation

1998-03-10: Type tags are represented as first class objects by instances of class **built-in-class**.

Multithreading

1998-04-22: The RPC client code for Solaris and Windows/NT is now multithreading safe. For IRIX and Linux, no multithreading RPC code can be generated by `rpcgen`.

Storing and loading of metaobjects

1998-05-05: For storing and loading of metaobjects, the `*root-persistent-heap*` is now used instead of the `*default-persistent-heap*`. This decouples transaction handling for metaobjects from the transaction handling for non-metaobjects. And, it uses the `*root-persistent-heap*` which was used before only during the bootstrap.

1998-11-19: This idea was not very good, since it raised lock conflicts for symbols used for classes and in the user's code. For storing and loading of metaobjects, now the `*default-persistent-heap*` is used again.

B.1.6 Release 2.02

Documentation

1997-08 – 1998-01: Wrote completely new user's guide.

1997-12-01: The manual has been split into two documents, one documenting the external API (file `eref.pdf`) and the other one documenting the internal entities (file `iref.pdf`).

1998-01: All documentation is now available in Portable Document Format (PDF), too, for improved cross-referencing and text searching.

Macro with-transaction

1998-01-28: Macro **with-transaction** [API Manual 1997, p. 135] has now an optional argument, which passes the database where the transaction should be started. For the moment, always use an empty optional argument:

```
(with-transaction ()
  <forms>)
```

Multiple servers on one machine

1998-02-04: More than one database can now be served on a single machine.

Each database directory contains a file `version.inf` with a RPC version number (1 for the 'main server', >1 for each database directory). Each open request from a client to a database is directed to the 'main server' (or any other server running), which looks up the RPC version number from the RPC version number file, start up a daemon (if non is running), and return the RPC version number to the client. This RPC version number is in turn used by the client to create the connection to the daemon serving the database. A new started server process will terminate itself after the last client has disconnected.

Administration

1998-02-04: Moved code from script `plobdadmin` into the server's C code; databases can now be created by calling function **p-create-database** [API Manual 1997, p. 61].

URL-based database naming

1998-02-06: The baroq splitting of a database' name into three components of transport protocol (former parameter ***default-server-transport***), host (former parameter ***default-server-host***) and directory (former parameter ***default-server-directory***) has been removed. Instead, a database is now named by a single URL (Universal Resource Locator), see class **url** [Internal Manual 1997, p. 261] for details. The name of the current database is found in parameter ***database-url*** [API Manual 1997, p. 30], whereas the default database is in parameter ***default-database-url*** [Internal Manual 1997, p. 31]; the effective database opened is formed by merging these both URLs.

B.1.7 Release 2.00 and 2.01

Macro character for persistent symbols changed

1997-02: In the first version the macro character `#l` (hash sign followed by the letter 'l') was used for adressing persistent symbols. At the port from LISPWORKS to ALLEGRO, it was found that the macro character `#l` is already occupied by ALLEGRO for reading logical pathnames, so the `#!` (hash sign followed by an exclamation mark) macro character is now used for adressing persistent symbols.

On-demand loading for slots of persistent CLOS objects

1997-08: The slots of classes declared with a `:metaclass persistent-metaclass` are now loaded on-demand.

Direct `p-...` functions

1997-10: The functions for directly working on persistent objects have been fixed to modify destructively a transient representation, too, if it is found in the cache.

Parameter *default-setf-depth*

1997-10: The parameter ***default-setf-depth*** was removed; parameter ***default-depth*** [Internal Manual 1997, p. 31] is now used as default depth for both storing and loading of objects.

Example code in file `plob-example.lisp`

Fixed 1997-09: The example showing usage of transactions did not work, since the `:extent` of the example class **person** [API Manual 1997, p. 143] was changed from `:persistent` to `:cached-write-through`; so, an object reload is necessary after a transaction's abort.

Initialization of transient slots

Fixed 1997-11: Slots declared `:transient` are now initialized to their `:initform`.

Structure accessors

Fixed 1997-12-01: In ALLEGRO Common LISP, the structure constructors and structure slot readers are now retrieved correctly from ALLEGRO's internal structure descriptions.

Read-only lock conflict

Fixed 1997-12-15: Added missing conflict between write-intent and read-only locks.

B.2 Known bugs and limitations

This section contains known bugs still left to be fixed for the next release. When one of the following bugs has been fixed in a next release, it will either be removed completely or be moved to section 'Release Notes' on page 85.

B.2.1 Crash on transfer of large persistent vectors

There is one bug not associated directly with PLOB, but with one of the RPC layers of UNIX: When a RPC call requests to transfer very big data blocks, the server or client may crash because it can't allocate enough memory. The affected RPC layer is the xdr-layer, doing the data conversion between the client and the server process. The xdr-layer is coded (by the provider of the used UNIX system) rather straight-forward: It allocates as much memory as it needs and does *no* blockwise transfer for big data blocks. IMHO, doing a 'good' blockwise transfer would also increase the fault tolerance of the connection between client and server; a feature which should be implemented by the RPC layer and *not* the RPC-using application layer.

In a few words, the consequences for PLOB! are that the server or client may crash if and only if a very large persistent vector ('vector' in the sense of Common LISP) is transferred between client and server. The concrete length coupled with 'very large' depends on the amount of memory which is available to the xdr-layer, so it is difficult to name a concrete length here. All which can be said is that it should be possible to transfer vectors with at least some thousand elements; for vectors with some tenthousand elements, this may not be possible.

Since vectors are used in the internal representation of arrays, this bug may also occure when transferring large arrays. In persistent structure and CLOS objects, persistent vectors are used for representing the instance data contained in their slots too (one element per slot), but I don't think that there will be persistent structure or CLOS objects containing thousands of slots.

PLOB! is already prepared to do a blockwise transfer for large vectors, but up to now (March 4th, 1997) I had no time to complete the code for the blockwise transfer.

B.2.2 Transferring of large, complicated graph-like structures

The code for transferring the state of objects between transient and persistent memory is a recursive-descent algorithm on each of an object's slots. For large, complicated graph-like structures this will grow the LISP stack very much; more worse, the time needed for one object transfer increases with the LISP stack size, too. 'Large' in this sense means some hundred objects in the transitive closure of an instance to store or load.

It would be a good idea to make something like a 'store-plan' or a 'load-plan' for the transfer of all object states referenced by the transitive closure of an instance to be stored or loaded, for example, transfer the states of referenced objects before the states of referencing objects.

B.2.3 Cache littering LISP memory

Since PLOB! uses a cache internally in the LISP code, this cache may become very large after some storing or loading of persistent objects. The cache can be cleared by a call to function **clear-cache** [API Manual 1997, p. 28], but this will also break the bindings of some transient LISP objects to their persistent identity.

The correct solution to this problem would be using weak pointers in the hash tables implementing the cache. Unfortunately, LISPWORKS Common LISP has no weak pointers at all. ALLEGRO Common LISP 4.3 has weak pointers on vectors, ALLEGRO Common LISP 5.0 will support hash tables with weak pointers.

B.2.4 Changes to database not propagated to each client's cache

When one client changes the state of a persistent object, this change isn't propagated to other clients holding the object's state too in their cache. The new object's state will be seen by the other clients on a next reload of the object (this can be forced by evaluating `(clear-cache)` and `(load-object <objid>)` in each of the client's listener which should see the new state).

A solution would be to add a lock mode `cached` to the already existing lock modes `read`, `read-only`, `write` and the `...-intent` modes. So, when the object's state changes, each of the clients holding the object in its cache could be notified of the change.

There would be a siginificant amount of overhead on maintaining consistency between the `cache`-locks represented in the persistent heap on the server side and the cache represented in the client's LISP process.

Idea: Do the communication between client and server in an own thread started at least by the client. So, a client waiting for input will do this in its own thread and won't block the 'main' client (LISP) process.

B.2.5 Missing `use-package` feature

A `use-package` feature for persistent packages would be necessary to resolve ambiguities among the names of persistent classes used across more than one LISP system (for details see remarks at function **p-find-class** [API Manual 1997, p. 65]). At the moment, there is no support for a `use-package` feature for persistent packages.

B.2.6 Slot option `:extent :persistent`

Limitation for ALLEGRO: The extent `:persistent` will also allocate transient memory for the slot. This is a limitation of ALLEGRO's MOP, since it allows only for `:instance` and `:class` slot allocations.

For LISPWORKS, `:extent :persistent` slots are not represented in transient memory, since PLOB! patches the class metaobjects of LISPWORKS to have the desired effect.

B.2.7 slot-value for classes with `:metaclass persistent-metaclass`

PLOB! will only gain control over a slot's state for classes with a class option `:metaclass persistent-metaclass`. Only for those classes, function **slot-value** [CLtLII 1990, p. 857] and function **(setf slot-value)** [CLtLII 1990, p. 857] call their generic function **slot-value-using-class** [AMOP 1991, p. 235] and generic function **(setf slot-value-using-class)** [AMOP 1991, p. 231] counterparts with methods specialized in PLOB.

B.2.8 Solaris 2.6

Use either the newest Solaris' `cc` or `gcc` version 2.8.0 for compiling under Solaris 2.6, `gcc` version 2.7.2 does not work under Solaris 2.6 at all.

B.2.9 Schema evolution

Schema evolution still seems to be a bit buggy; the LISP code for schema evolution should be translated to C and moved to the server's side.

B.2.10 Missing compare methods

The compare methods for ratios and bignums are still missing. For complex numbers, 2-dimensional indexes are missing.

B.2.11 LispWorks 3.2.0

PLOB's LISP code will not compile under LispWorks 3.2.0, since some defsystem features and the foreign function interface have been changed by Harlequin with 3.2.2. An upgrade to LispWorks 3.2.2 or 4.0.1 is necessary.

B.2.12 LispWorks 4.0.1

With LISPWORKS Common LISP 4.0.1 for Windows/NT, bit vectors cannot be stored.

B.2.13 Function subtypep in ACL 4.3.0

In Allegro CL 4.3.0, function subtypep may return incorrect results or raise an error instead of returning a result. If the stack backtrace in the debugger tells that the error is within function subtypep, an upgrade to Allegro CL 4.3.1 is necessary or the appropriate patches should be downloaded from Franz' ftp site.

B.3 To do and future features

This section contains things still left to be done for the next release. When one of the following topics has been implemented in a next release, it will be moved to section 'Release Notes' on page 85.

B.3.1 Views onto persistent objects

Write function [GS]etSlotValues (replacement for SH_read_indices(), SH_write_indices()) which takes the *objid* of an object and the *objid* of an 'expected class'; coerce internal physical representation to/from class expected. With GetSlotValues, a list of added, transient slots should be returned, too; these slots are then initialized by LISP by a call to shared-initialize.

In the server, differences between two versions of a class should be represented by a kind of 'Delta-object', which can transform from the 'old' to the 'new' physical representation. This would also be useful for establishing views onto objects. Views are considered as 'weak classes', allowing no durable (but only transient) schema evolution onto its instances.

On the LISP side, add a slot `descr` to class **persistent-clos-object** [Internal Manual 1997, p. 168], containing the class description the object actually has. This class description stored within each instance can be used to detect if the object should undergo a schema evolution. Problem for loose-bounded classes: Where should the information about the class description go? Perhaps into the cache; entries into the cache for loose bound CLOS objects should be a `defstruct` mapping a transient object onto its objid and actual class description.

Create bultin class ViewSet, which contains instances belonging to a single view with equal slot values. Could also be used within btrees to represent the set of objects belonging to a single, non-unique key.

B.3.2 Multidimensional indexes

Add multidimensional indexes, for example x-trees [Berchtold et al. 1996, Beckmann et al. 1990].

B.3.3 Multi-threaded server

With moderate effort, the server could be made multithreaded. This could be used to reactivate the suspend- and wakeup-algorithms implemented in PLOB! 1.0 used for clients waiting on locks to be granted. Also, integrating some event support would be nice.

B.3.4 Name spaces

Add more name spaces, for example a hierarchical name space, similar to a file system. There is alr

B.3.5 Better representation

Introduce explicit or better representations for databases (these are not represented at all) and transactions (these are represented as transaction ids embedded into a session).

Appendix C

Contributions and acknowledgements

First of all, I would like to thank Carsten Schröder for his advice and interesting discussions when starting my work on PLOB! He provided me with the first ideas on object-oriented persistency and assisted me in understanding LISP, CLOS and MOP. Without these conversations, this system surely would not look like as it does now.

I would like to thank the Persistent Programming Research Group at St. Andrews University for the permission to use and re-distribute their POSTORE library my server is built upon. The port to SGI's IRIX would not have been possible without the help of Larry Hunter. The questions of Jong-won Choi have been a big help at writing this user's guide. Will Hartung reported the bug on starting the `portmap` daemon. Ralf Möller was the first one who did a stress testing on PLOB! by using it in conjunction with `cl-http` and storing a lot of real-world data. The cooperation with him fixed many bugs and gave me some hints on how PLOB! could be improved. Martti Halminen contributed to this user's guide by doing the first proof reading. Thanks to Martin Gergeleit for his port of the original SUN ONC RPC library to Windows/NT, and for his permission to use his port without any royalties involved. Thanks to Paul Meurer for his help fixing the Windows/NT LISP client code. Thanks to RainerJoswig for setting up and maintaining the first PLOB! mailing list (`plob@lisp.de`) and the WWW site hosting the first PLOB! archives. Myriam Abramson pointed out some bugs, thanks for that.

After 3 years between releasing 2.10 and 2.11, many things had to be fixed for release 2.11. I'd like to thank LispWorks and Franz, Inc. for providing me with generous eval licenses for my ongoing work. Release 2.11 has been tested by Bogdan Norenko, Klaus Harbo and Martin Simmons.

Appendix D

Distribution

This chapter contains miscellaneous topics having to do with the distribution of PLOB.

D.1 WWW and email addresses

The official distribution site is `http://plob.sourceforge.net`. The author's email address is `mailto:Heiko.Kirschke@acm.org`. New versions will be announced to the mailing list `mailto:plob-discussion@lists.sourceforge.net` (for subscribing, either use the mailing list Web interface at `http://lists.sourceforge.net/lists/listinfo/plob-discussion` or send an email to `mailto:plob-discussion-request@lists.sourceforge.net` with `subscribe` in the email's text). The list is intended for getting support on problems with the installation or usage of PLOB.

D.2 Submitting a bug

Submit bugs to the bug tracking utility at `http://www.sourceforge.net/projects/plob`. If you submit a bug, please supply the following informations:

1. The machine and operating system used, for example as returned by calling `uname -a` in a shell.
2. The LISP system and version used.
3. The PLOB! version used.
4. If applicable, the output shown in the LISP listener which led to the error. If the LISP debugger is raised, please add a stack backtrace, namely both `:zoom` and `:bt` for ALLEGRO Common LISP or `:bq` and `:bb` for LISPWORKS Common LISP.
5. If applicable, take a look into the server's logfile `messages.log` located in the database directory and try to identify any error output which might have to do with the error shown at the LISP listener's error prompt. If in doubt, send the whole server's logfile.

D.3 Directory structure

This section explains the structure of the directories delivered with PLOB.

Subdirectory `plob-2.11/bin/`

Find here the server process `plobd` and some shell scripts used for compiling, generating documentation etc.

Subdirectory `plob-2.11/conf/`

UNIX `make` configuration and rule files.

Subdirectory `plob-2.11/database/`

This conatains an empty database file `stablestore.empty` used by the PLOB! server for storing the persistent objects.

Subdirectory `plob-2.11/lib/`

C libraries: The library `libplob.a` with object code common to the server and client, the server library `libsplob.a`, the client library `libcplob.a` and the POSTORE library `libpostore.a`.

Subdirectory `plob-2.11/mail/`

Some mailings done to different sites.

Subdirectory `plob-2.11/ps/`

The POSTSCRIPT files with the documentation.

Subdirectory `plob-2.11/src/server/`

The C code for the PLOB! server.

Subdirectory `plob-2.11/src/client/`

The C code for the PLOB! client.

Subdirectory `plob-2.11/src/lisp/`

The LISP code for the PLOB! client.

Subdirectory `plob-2.11/src/common/`

The C code used by both the PLOB! server and client.

Subdirectory `plob-2.11/src/include/`

C header files.

Subdirectory `plob-2.11/src/include/allegro/`, `plob-2.11/src/include/lispwork`

The LISP files generated from C header files for the foreign function interface.

Subdirectory `plob-2.11/src/lisp-doc/`

The LISP module and TEX styles used for preparing the reference manual [API Manual 1997] by extracting the documentation strings from the LISP sources.

Subdirectory `plob-2.11/src/lispworks/memory_representation/`

Some early experiments about the memory representation of transient objects in LISPWORKS.

Subdirectories `plob-2.11/tex/inputs/`, `plob-2.11/tex/manual/`

The documentation style and TEX source files.

D.4 Standard `make` targets

Each `Makefile` found in one subdirectory of `plob-2.11/` has following standard targets. One or some of these standard targets can be given as arguments to a call of `make` as additional command line arguments. Some `Makefiles` have more targets than described here; consult the `Makefile` comments to find out which additional targets are defined. It is not necessary to use always the 'top-level' makefile `plob-2.11/Makefile` when only files in one subdirectory have been changed; calling `make` in each subdirectory will re-build only that subdirectory.

Standard `make` target `all`

This is the default target; all sources are compiled with default compiler settings.

Standard `make` target `initial`

Calling this target 'initializes' the subdirectory, for example by setting up symbolic links, creating directories etc. This target should only be called exactly once, but no damage will result from calling it more than once.

Standard `make` target `clean`

All 'garbage' files are removed; these are files with their names matching `*`∼ (emacs backup files), `core`, `a.out` etc. No object files are removed.

Standard `make` target `dist-clean`

Like standard target 'clean' plus all object files, libraries and otherwise generated files are removed. 'dist-clean' is an abbrevation for 'distribution-clean'. All files not belonging to a distribution are removed.

Standard `make` target `toc`

This target is only found in makefiles which generate POSTSCRIPT files. It calls TEX up to three times to make sure that the table of contents of the documentation will be correct.

D.5 Disclaimer

The work on PLOB! in its entire contents, that means all its software and its documentation, has been carried out privately. My employer, CSC PLOENZKE AG has nothing to do with the work presented here. CSC PLOENZKE AG neither develops nor supports PLOB! nor will do so in the future in any way.

D.6 License Terms

There are *(sic!)* a finite number of jokes in the universe.

— David Byrne: Stop Making Sense