Elephant User Manual

Elephant version 0.9

By Ian Eslick with Robert Read and Ben Lee

Short Contents

Table of Contents

1 Introduction

Elephant is a persistent object protocol and database for Common Lisp. The persistent protocol component of elephant overrides class creation and standard slot accesses using the Meta-Object Protocol (MOP) to render slot values persistent. Database functionality includes the ability to persistently index and retrieve ordered sets of class instances and ordinary lisp values. Elephant has an extensive test suite and the core functionality is becoming quite mature.

The Elephant code base is available under the LLGPL license. Data stores each come with their own, separate license and you will have to evaluate the implications of using them yourself.

1.1 History

Elephant was originally envisioned as a lightweight interface layer on top of the Berkeley DB library, a widely-distributed embedded database that many unix systems have installed by default. Berkeley DB is ACID compliant, transactional, process and thread safe, and fast relative to relational databases.

Elephant has been extended to provide support for multiple backends, specifically a relational database backend based on CL-SQL which has been tested with Postgres and SQLite 3, and probably support other relational systems easily. It supports, with some care, multi-repository operation and enables convenient migration of data between repositories.

The support for relational backends and migration to the LLGPL was to allow for broader use of Elephant in both not-for-profit and commercial settings. Several additional backends are planned for future releases including a native Lisp implementation released under the LLGPL.

Elephant's current development focus is to enhance the feature set including a native lisp backend, a simple query language, and flexible persistence models that selectively break one or more of the ACID constraints to improve performance.

1.2 Elephant Goals

- **Transparency:** most Lisp values are easy to persist without significant effort or special syntax. You can interact with the DB entirely from Lisp. There is no requirement to use domain-specific languages, such as SQL, to access persistent resources. Elephant loads via ASDF and requires no external server (except for some SQL backends like Postgres).
- **Simplicity:** a small library with few surprises for the programmer. Lisp and Berkeley DB together are an excellent substrate; Elephant tries to leverage their features as much as possible. Support for additional backends are load-time options and more or less transparent to the user.
- **Safety:** ACID, transactions. Concurrent with good multi-user (BDB) and multi-threaded semantics (BDB/SQL), isolation, locking and deadlock detection. (Deadlock detection does require an external process to be launched for Berkeley DB)
- **Performance:** leverage Berkeley DB performance and/or Relational database reliability. In addition to fast concurrent / transactional modes, elephant will (eventually) offer

an accelerated single-user as well as pure in-memory mode that should be comparable to prevalence style solutions, but employ a common programmer interface.

- **Historical continuity:** Elephant does not try to innovate significantly over prior Lisp persistent object stores such as AllegroStore (also based on Berkeley DB), the new AllegroCache, the Symbolics system Statice and PLOB. Anyone familiar with those systems will recognize the Elephant interface.
- **License Flexibility:** Elephant is released under the LLGPL. Because it supports multiple implementation of the backend, one can choose a backend with licensing and other features appropriate to your needs.

1.3 More Information

Join the Elephant mailing lists to ask your questions and receive updates. You can also review archives for past discussions and questions. Pointers can be found on the Elephant website at

`http://www.common-lisp.net/project/elephant`.

Installation instructions can be found in the Chapter 3 [Installation], page 23 section. Bugs can be reported via the Elephant Trac system at

`http://trac.common-lisp.net/elephant/`.

This also serves as a good starting point for finding out what new features or capabilities you can contribute to Elephant. The Trac system also contains a wiki with design discussions and a FAQ.

2 Tutorial

2.1 Overview

Elephant is a Persistence Metaprotocol and Database for Common Lisp. It provides the ability for users to define and interact with persistent objects and to transparently store ordinary lisp values. Persistent objects are CLOS instances that overload the ordinary slot access semantics so that every write to a slot is passed through and written to disk. Non-persistent lisp objects and values can be written to slots and will be automatically persisted. In addition, Elephant provides a persistent index which maintains an ordered collection of lisp values or persistent object references.

The use of persistent objects makes coding concise, convenient, and powerful, and makes persistence almost invisible to the programmer. However, Elephant also allows the same basic data dictionary of key/value retrieval that BerkeleyDB provides.

When someone says "database," most people think of SQL Relational Data Base Management Systems (e.g. Oracle, Postgresql, MySql). Those systems store data in statically typed tables with unique shared values to connect rows in separate tables. Objects can be mapped into these tables in an object-relational mapping that assigns objects to rows and slot values to columns in a row's table. If a slot references another type of object, a unique ID can be used to reference that object's table. CL-SQL, for example, provides facilities for this kind of object-relational mapping and there are many systems for other languages that do the same (i.e. Hibernate for Java).

While Elephant can use either RDBMSs or Berkeley DB as a data store, the model it supports is that of objects stored in persistent indices. Unlike systems such as Hibernate for Java, the user does not need to construct or worry about a mapping from the object space into the database. Elephant relies on LISP rather than SQL for its data manipulation language. Elephant is designed to be a simple and convenient tool for the programmer.

Elephant consists of a small universe of basic concepts:

- **Store controller:** the interface between lisp and a data store. Most operations require or accept a store controller, or a default store controller stored in `*store-controller*` to function.
- **Persistent Sets:** A simple persistent collection is provided which allows the creation of persistent sets.
- **BTrees:** Elephant provides a persistent key-value abstraction based on the BTree data structure. Values can be written to or read from a BTree and are stored in a sorted order.
- **Stored values:** most lisp values, including standard objects, arrays, etc can be used as either key or value in a persistent BTree.
- **Persistent objects:** An object where most slot values are stored in the data store and are written to or retrieved from disk on slot accesses. Storing a persistent object stores only a reference, allowing for object identity.
- **Object indexing:** The ability to lookup and sort objects by their slot values rather than by explicit inclusion in a collection.

- **Transactions:** a dynamic context for executing operations on objects or collections such that the side effects exhibit the ACID (atomicity, consistency, isolation and durability) properties of database.

There are a set of more advanced concepts you will learn about later, but these basic concepts will serve to acquaint you with Elephant.

If you do not already have Elephant installed and building correctly, read the Chapter 3 [Installation], page 23 section of this manual and then move on to Section 2.2 [Getting Started], page 4.

2.2 Getting Started

The first step in using elephant is to open a store controller. A store controller is an object that coordinates lisp program access to the chosen data store.

To obtain a store controller, you call `open-store` with a store specification. A store specification is a list containing a backend specifier (`:BDB` or `:CLSQL`) and a backend-specific reference.

For :BDB, the second element is a string or pathname that references a local directory for the database files. This directory must be created prior to calling open-store.

```
(open-store '(:BDB ‘‘/users/me/db/my-db/’’))
```

For :CLSQL the second argument is another list consisting of a specific SQL database and the name of a database file or connection record to the SQL server. Examples are:

```
(open-store '(:CLSQL (:SQLITE "/users/me/db/sqlite.db")))
(open-store '(:CLSQL (:POSTGRESQL "localhost.localdomain"
                                  "mydb" "myuser" ""))))
```

We use Berkeley DB as our example backend. To open a BDB store-controller we can do the following:

```
(asdf:operate 'asdf:load-op :elephant)
(use-package :elephant)
(setf *test-db-spec*
      '(:BDB "/home/me/db/testdb/"))
(open-store *test-db-spec*)
```

We do not need to store the reference to the store just now as it is automatically assigned to the variable, `*store-controller*`. For a deeper discussion of store controller management see the Chapter 4 [User Guide], page 33.

When you're done with your session, release the store-controller's resources by calling `close-store`.

Also there is a convenience macro `with-open-store` that will open and close the store, but opening the store is an expensive operation so it is generally better to leave the store open until your application no longer needs it.

2.3 The Store Root

What values live between lisp sessions is called *liveness*. Liveness in a store is determined by whether the value can be reached from the root of the store. The root is a special BTree in which other BTrees and lisp values can be stored. This BTree has a special interface through the store controller. (There is a second root BTree called the class root which will be discussed later.)

You can put something into the root object by

```
(add-to-root "my key" "my value")
=> "my value"
```

and get things out via

```
(get-from-root "my key")
=> "my value"
=> T
```

The second value indicates whether the key was found. This is important if your key-value pair can have nil as a value.

You can perform other basic operations as well.

```
(root-existsp "my key")
=> T
(remove-from-root "my key")
=> T
(get-from-root "my key")
=> NIL
=> NIL
```

To access all the objects in the root, the simplest way is to simply call `map-root` with a function to apply to each key-value pair.

```
(map-root
  (lambda (k v)
     (format t "key: ~A value:~A~%" k v)))
```

You can also access the root object directly.

```
(controller-root *store-controller*)
=> #<DB-BDB::BDB-BTREE  #x10e86042>
```

It is an instance of a class "btree"; see Section 4.6 [Persistent BTrees], page 44.

2.4 Serialization

What can you put into the store besides strings? Almost all lisp values and objects can be stored: numbers, symbols, strings, nil, characters, pathnames, conses, hash-tables, arrays, CLOS objects and structs. Nested and circular things are allowed. Nested and circular things are allowed. You can store basically anything except compiled functions, closures, class objects, packages and streams. Functions can be stored as uncompiled lambda expressions. (Compiled functions and other kinds of objects may eventually get supported too.)

Elephant needs to use a representation of data that is independant of a specific lisp or data store. Therefore all lisp values that are stored must be *serialized* into a canonical format. Because Berkeley DB supports variable length binary buffers, Elephant uses a binary serialization system. This process has some important consequences that it is very important to understand:

1. **Lisp identity can't be preserved**. Since this is a store which persists across invocations of Lisp, this probably doesn't even make sense. However if you get an object from the index, store it to a lisp variable, then get it again - they will not be eq:

   ```
   (setq foo (cons nil nil))
   => (NIL)
   (add-to-root "my key" foo)
   => (NIL)
   (add-to-root "my other key" foo)
   => (NIL)
   (eq (get-from-root "my key")
         (get-from-root "my other key"))
   => NIL
   ```

2. **Nested aggregates are stored in one buffer**. If you store an set of objects in a hash table you try to store a hash table, all of those objects will get stored in one large binary buffer with the hash keys. This is true for all other aggregates that can store type T (cons, array, standard object, etc).
3. **Mutated substructure does not persist**.

   ```
   (setf (car foo) T)
   => T
   (get-from-root "my key")
   => (NIL)
   ```

   This will affect all aggregate types: objects, conses, hash-tables, et cetera. (You can of course manually re-store the cons.) In this sense elephant does not automatically provide persistent collections. If you want to persist every access, you have to use BTrees (see Section 4.6 [Persistent BTrees], page 44).
4. **Serialization and deserialization can be costly**. While serialization is pretty fast, but it is still expensive to store large objects wholesale. Also, since object identity is impossible to maintain, deserialization must re-cons or re-allocate the entire object every time increasing the number of GCs the system does. This eager allocation is contrary to how most people want to use a database: one of the reasons to use a database is if your objects can't fit into main memory all at once.
5. **Merge-conflicts in heavily multi-process/threaded situations**. This is the common read-modify-write problem in all databases. We will talk more about this in the Section 2.9 [Using Transactions], page 15 section.

This may seem terribly restrictive, but don't despair, we'll solve most of these problems in the next section.....

2.5 Persistent Classes

The Common Lisp Object System and the Metaobject Protocol, gives us the tools to solve these problems for objects:

```
(defclass my-persistent-class ()
  ((slot1 :accessor slot1)
   (slot2 :accessor slot2))
  (:metaclass persistent-metaclass))
```

```
(setq foo (make-instance 'my-persistent-class))
=> #<MY-PERSISTENT-CLASS {492F4F85}>

(add-to-root "foo" foo)
=> NIL
(add-to-root "bar" foo)
=> NIL
(eq (get-from-root "foo")
    (get-from-root "bar"))
=> T
```

What's going on here? Persistent classes, that is, classes which use the `persistent-metaclass` metaclass, are given unique IDs (accessable through `ele::oid`). They are serialized simply by their OID and class. Slot values are stored separately (and invisible to the user) keyed by OID and slot. Loading (deserializing) a persistent class

```
(get-from-root "foo")
=> #<MY-PERSISTENT-CLASS {492F4F85}>
```

instantiates the object or finds it in a memory cache if it already exists. (The cache is a weak hash-table, so gets flushed on GCs if no other references to the persistent object are kept in memory). The slot values are NOT loaded until you ask for them. In fact, the persisted slots don't have space allocated for them in the instances, because we're reading from the database.

```
(setf (slot1 foo) "one")
=> "one"
(setf (slot2 foo) "two")
=> "two"
(slot1 foo)
=> "one"
(slot2 foo)
=> "two"
```

Changes made to them propogate automatically:

```
(setf (slot1 foo) "three")
=> "three"
(slot1 (get-from-root "bar"))
=> "three"
```

You can also create persistent classes using the convenience macro `defpclass`.

```
(defpclass my-persistent-class ()
  ((slot1 :accessor slot1)
   (slot2 :accessor slot2)))
```

Although it is hard to see here, serialization / deserialization of persistent classes is fast, much faster than ordinary CLOS objects. Finally, they do not suffer from merge-conflicts when accessed within a transaction (see below). In short: persistent classes solve the problems associated with storing ordinary CLOS objects. We'll see later that BTrees solve the problems associated with storing hash-tables.

2.6 Rules about Persistent Classes

Using the `persistent-metaclass` metaclass declares all slots to be persistent by default. To make a non-persistent slot use the `:transient t` flag. Class slots `:allocation :class` are never persisted, for either persistent or ordinary classes. (Someday, if we choose to store class objects, this policy may change).

Persistent classes may inherit from other classes. Slots inherited from persistent classes remain persistent. Transient slots and slots inherited from ordinary classes remain transient. Ordinary classes cannot inherit from persistent classes – otherwise persistent slots could not be stored!

```
(defclass stdclass1 ()
  ((slot1 :initarg :slot1 :accessor slot1)))

(defclass stdclass2 (stdclass1)
  ((slot2 :initarg :slot2 :accessor slot2)))

(defpclass pclass1 (stdclass2)
  ((slot1 :initarg :slot1 :accessor slot1)
   (slot3 :initarg :slot3 :accessor slot3)))

(make-instance 'pclass1 :slot1 1 :slot2 2 :slot3 3)
=> #<PCLASS1 {x10deb88a}>

(add-to-root 'pinst *)
=> #<PCLASS1 {x10deb88a}>

(slot1 pinst)
=> 1

(slot2 pinst)
=> 2

(slot3 pinst)
=> 3
```

Now we can simulate a new lisp session by flushing the instance cache, reloading our object then see what slots remain. Here persistent slot1 should shadow the standard slot1 and thus be persistent. Slot3 is persistent by default and slot2, since it is inherited from a standard class should be transient.

```
(elephant::flush-instance-cache *store-controller*)
=> #<EQL hash-table with weak values, 0 entries {x11198a02}>

(setf pinst (get-from-root 'pinst))
=> #<PCLASS1 {x1119b652}>

(slot1 pinst)
=> 1
```

```
(slot-boundp pinst slot2 pinst)
=> nil

(slot3 pinst)
=> 3
```

Using persistent objects has implications for the performance of your system. Note that the database is read every time you access a slot. This is a feature, not a bug, especially in concurrent situations: you want the most recent commits by other threads, right? This can be used as a weak form of IPC. But also note that in particular, if your slot value is not an immediate value or persistent object, reading will cons or freshly allocate storage for the value.

Gets are not an expensive operation; you can perform thousands to tens of thousands of primitive reads per second. However, if you're concerned, cache large values in memory and avoid writing them back to disk as long as you can.

2.7 Persistent collections

The remaining problem outlined in the section on Section 2.4 [Serialization], page 5 is that operations which mutate collection types do not have persistent side effects. We have solved this problem for objects, but not for collections such as as arrays, hashes or lists. Elephant provides two solutions to this problem: the `pset` and `btree` classes. Each provides persistent addition, deletion and mutation of elements, but the pset is a simple data structure that may be more efficient in memory and time than the more general btree.

2.7.1 Using PSets

The persistent set maintains a persistent, unordered collection of objects. They inherit all the important properties of persistent objects: identity and fast serialization. They also resolve the mutated substructure and nested aggregates problem for collections. Every mutating write to a `pset` is an independent and persistent operation and you can serialize or deserialize a `pset` without serializing any of it's key-value pairs.

The `pset` is also a very convenient data structure for enabling a persistent slot contain a collection that can be updated without deserializing and/or reserializing a list, array or hash table on every access.

Let's explore this data structure through a (very) simple social networking example.

```
(defpclass person ()
  ((name :accessor person-name :initarg :name))
  ((friends :accessor person-friends :initarg :friends)))
```

Our goal here is to store a list of friends that each person has, this simple graph structure enables analyses such as who are the friends of my friends, or do I know someone who knows X or what person has the minimum degree of separation from everyone else?

Without psets, we would have to do something like this:

```
(defmethod add-friend ((me person) (them person))
  (let ((friends (person-friends me)))
    (pushnew them friends)
    (setf (person-friends me) friends)))

(defmethod remove-friend ((me person) (them person))
  (let ((remaining-friends (delete them (person-friends me))))
    (setf (person-friends me) remaining-friends)))

(defmethod map-friends (fn (me person))
  (mapc fn (person-friends me)))
```

Ouch! This results in a large amount of consing. We have to deserialize and generate a freshly consed list every time we call `person-friends` and then reserialize and discard it on every call to `(setf person-friends)`.

Instead, we can simply use a `pset` as the value of friends and implement the add and remove friend operations as follows:

```
(defpclass person ()
  ((name :accessor person-name :initarg :name))
  ((friends :accessor person-friends :initarg :friends
            :initform (make-pset))))

(defmethod add-friend ((me person) (them person))
  (insert-item them (person-friends me)))

(defmethod remove-friend ((me person) (them person))
  (remove-item them (person-friends me)))

(defmethod map-friends (fn (me person))
  (map-pset fn (person-friends me)))
```

If you want a list to be returned when the user calls person-friends themselves, you can simply rejigger things like this:

```
(defpclass person ()
  ((name :accessor person-name :initarg :name))
  ((friends :accessor person-friends-set :initarg :friends
            :initform (make-pset))))

(defmethod person-friends ((me person))
  (pset-list (person-friends-set me)))
```

If you just change the person-friends calls in our prior functions, the new set of functions removes `(setf person-friends)`, which doesn't make sense for a collection slot, allows users to get a list of the friends for easy list manipulations and avoids all the consing that plagued our earlier version.

You can use a `pset` in any way you like just like a persistent object. The only difference is the api used to manipulate it. Instead of slot accessors, we use insert, remove, map and find.

There is one drawback to persistent sets and that is that they are not garbage collected. Over time, orphaned sets will eat up alot of disk space. Therefore you need to explicitly free the space or resort to more frequent uses of the migrate procedure to compact your database. The pset supports the `drop-pset`

However, given that persistent objects have the same explicit storage property, using psets to create collection slots is a nice match.

2.7.2 Using BTrees

BTrees are collections of key-value pairs ordered by key with a log(N) random access time and a rich iteration mechanism. Like persistent sets, they solve all the collection problems of the prior sections. Every key-value pair is stored independently in Elephant just like persistent object slots.

The primary interface to `btree` objects is through `get-value`. You use `setf get-value` to store key-value pairs. This interface is very similar to `gethash`.

The following example creates a btree called `*friends-birthdays*` and adds it to the root so we can retrieve it during a later sessions. We then will add two key-value pairs consisting of the name of a friend and a universal time encoding their birthday.

```
(defvar *friends-birthdays* (make-btree))
=> *FRIENDS-BIRTHDAYS*

(add-to-root 'friends-birthdays *friends-birthdays*)
=> #<BTREE {4951CF6D}>

(setf (get-value "Ben" *friends-birthdays*)
      (encode-universal-time 0 0 0 14 4 1973))
=> 2312600400

(setf (get-value "Andrew" *friends-birthdays*)
      (encode-universal-time 0 0 0 22 12 1976))
=> 2429071200

(get-value "Andrew" *friends-birthdays*)
=> 2429071200
=> T

(decode-universal-time *)
=> 0
   0
   0
   22
   12
   1976
   2
   NIL
   6
```

In addition to the hash-table like interface, `btree` stores pairs sorted by the lisp value of the key, lowest to highest. This is works well for numbers, strings, symbols and persistent objects, but due to serialization semantics may be strange for other values like arrays, lists, standard-objects, etc.

Because elements are sorted by value, we can iterate over all the elements of the BTree in order. Notice that we entered the data in reverse alphabetic order, but will read it out in alphabetical order.

```
(map-btree (lambda (k v)
             (format t "name: ~A utime: ~A~%" k
               (subseq (multiple-value-list
                         (decode-universal-time v)) 3 6)))
           *friends-birthdays*)
"Andrew"
"Ben"
=> NIL
```

But what if we want to read out our friends from oldest to youngest? One way is to employ another btree that maps birthdays to names, but this requires multiple `get-value` calls for each update, increasing the burden on the programmer. Elephant provides several better ways to do this.

The next section Section 2.8 [Indexing Persistent Classes], page 12 shows you how to order and retrieve persistent classes by one or more slot values.

2.8 Indexing Persistent Classes

Class indexing simplifies the storing and retrieval of persistent objects. An indexed class stores every instance of the class that is created, ensuring that every object is automatically persisted between sessions.

```
(defpclass friend ()
  ((name :accessor name :initarg :name)
   (birthday :initarg :birthday))
  (:index t))
=> #<PERSISTENT-METACLASS FRIEND>

(defmethod print-object ((f friend) stream)
  (format stream "#<~A>" (name f)))

(defun encode-date (dmy)
  (apply #'encode-universal-time
    (append '(0 0 0) dmy)))

(defmethod (setf birthday) (dmy (f friend))
  (setf (slot-value f 'birthday)
        (encode-date dmy))
  dmy)

(defun decode-date (utime)
  (subseq (multiple-value-list (decode-universal-time utime)) 3 6))
```

```
(defmethod birthday ((f friend))
  (decode-date (slot-value f 'birthday)))
```

Notice the class argument ":index t". This tells Elephant to store a reference to this class. Under the covers, there are a set of btrees that keep track of classes, but we won't need to worry about that as all the functionality has been nicely packaged for you.

We also created our own birthday accessor for convenience so it accepts and returns birthdays in a list consisting of month, day and year such as `(27 3 1972)`. The index key will be the encoded universal time, however.

Now we can easily manipulate all the instances of a class.

```
(defun print-friend (friend)
  (format t " name: ~A birthdate: ~A~%"
          (name friend) (birthday friend)))

(make-instance 'friend :name "Carlos"
                       :birthday (encode-date '(1 1 1972)))
(make-instance 'friend :name "Adriana"
                       :birthday (encode-date '(24 4 1980)))
(make-instance 'friend :name "Zaid"
                       :birthday (encode-date '(14 8 1976)))

(get-instances-by-class 'friends)
=> (#<Carlos> #<Adriana> #<Zaid>)

(mapcar #'print-friend *)
 name: Carlos birthdate: (1 1 1972)
 name: Adriana birthdate: (24 4 1980)
 name: Zaid birthdate: (14 8 1976)
=> (#<Carlos> #<Adriana> #<Zaid>)
```

But what if we have thousands of friends? Aside from never getting work done, our get-instances-by-class will be doing a great deal of consing, eating up lots of memory and wasting our time. Fortunately there is a more efficient way of dealing with all the instances of a class.

```
(map-class #'print-friend 'friend)
 name: Carlos birthdate: (1 1 1972)
 name: Adriana birthdate: (24 4 1980)
 name: Zaid birthdate: (14 8 1976)
=> NIL
```

`map-class` has the advantage that it does not keep references to objects after they are processed. The garbage collector can come along, clear references from the weak instance cache so that your working set is finite. The list version above conses all objects into memory before you can do anything with them. The deserialization costs are very low in both cases.

Notice that the order in which the records are printed are not sorted according to either name or birthdate. Elephant makes no guarantee about the ordering of class elements, so you cannot depend on the insertion ordering shown here.

So what if we want ordered elements? How do we access our friends according to name and birthdate? This is where slot indices come into play.

```
(defpclass friend ()
  ((name :accessor name :initarg :name :index t)
   (birthday :initarg :birthday :index t)))
```

Notice the :index argument to the slots and that we dropped the class :index argument. Specifying that a slot is indexed automatically registers the class as indexed. While slot indices increase the cost of writes and disk storage, each entry is only slightly larger than the size of the slot value. Numbers, small strings and symbols are good candidate types for indexed slots, but any value may be used, even different types. Once a slot is indexed, we can use the index to retrieve objects by slot values.

`get-instances-by-value` will retrieve all instances that are equal to the value argument.

```
(get-instances-by-value 'friends 'name "Carlos")
=> (#<Carlos>)
```

But more interestingly, we can retrieve objects for a range of values.

```
(get-instances-by-range 'friends 'name "Adam" "Devin")
=> (#<Adriana> #<Carlos>)

(get-instances-by-range 'friend 'birthday
                        (encode-date '(1 1 1974))
                        (encode-date '(31 12 1984)))
=> (#<Zaid> #<Adriana>)

(mapc #'print-friend *)
 name: Zaid birthdate: (14 8 1976)
 name: Adriana birthdate: (24 4 1980)
=> (#<Zaid> #<Adriana>)
```

To retrieve all instances of a class in the order of the index instead of the arbitrary order returned by `get-instances-by-class` you can use nil in the place of the start and end values to indicate the first or last element. (Note: to retrieve instances null values, use `get-instances-by-value` with nil as the argument).

```
(get-instances-by-range 'friend 'name nil "Sandra")
=> (#<Adriana> #<Carlos>)

(get-instances-by-range 'friend 'name nil nil)
=> (#<Adriana> #<Carlos> #<Zaid>)
```

There are also functions for mapping over instances of a slot index. To map over duplicate values, use the :value keyword argument. To map by range, use the :start and :end arguments.

```
(map-inverted-index #'print-friend 'friend 'name :value "Carlos")
 name: Carlos birthdate: (1 1 1972)
=> NIL

(map-inverted-index #'print-friend 'friend 'name
                  :start "Adam" :end "Devin")
 name: Adriana birthdate: (24 4 1980)
 name: Carlos birthdate: (1 1 1972)
=> NIL

(map-inverted-index #'print-friend 'friend 'birthday
                  :start (encode-date '(1 1 1974))
                  :end (encode-date '(31 12 1984)))
 name: Zaid birthdate: (14 8 1976)
 name: Adriana birthdate: (24 4 1980)
=> NIL

(map-inverted-index #'print-friend 'friend 'birthday
                  :start nil
                  :end (encode-date '(10 10 1978)))
 name: Carlos birthdate: (1 1 1972)
 name: Zaid birthdate: (14 8 1976)
=> NIL

(map-inverted-index #'print-friend 'friend 'birthday
                  :start (encode-date '(10 10 1975))
                  :end nil)
 name: Zaid birthdate: (14 8 1976)
 name: Adriana birthdate: (24 4 1980)
=> NIL
```

The Chapter 4 [User Guide], page 33 contains a descriptions of the advanced features of Section 4.4 [Class Indices], page 43 such as "derived indicies" that allow you to order classes according to an arbitrary function, a dynamic API for adding and removing slots and how to set a policy for resolving conflicts between the code image and a database where the indexing specification differs.

This same facility is also available for your own use. For more information see Section 4.8 [BTree Indexing], page 46.

2.9 Using Transactions

One of the most important features of a database is that operations enforce the ACID properties: Atomic, Consistent, Isolated, and Durable. In plainspeak, this means that a set of changes is made all at once, that the database is never partially updated, that each set of changes happens sequentially and that a change, once made, is not lost.

Elephant provides this protection for all primitive operations. For example, when you write a value to an indexed slot, the update to the persistent slot record as well as the slot index is protected by a transaction that performs all the updates atomically and thus enforcing consistency.

2.9.1 Why do we need Transactions?

Most real applications will need to use explicit transactions rather than relying on the primitives alone because you will want multiple read-modify-update operations act as an atomic unit. A good example for this is a banking system. If a thread is going to modify a balance, we don't want another thread modifying it in the middle of the operation or one of the modifications may be lost.

```
(defvar *accounts* (make-btree))

(defun add-account (account)
  (setf (get-value account *account*)

(defun balance (account)
  (get-value account *accounts*))

(defun (setf balance) (amount account)
  (setf (get-value account *accounts*) amount))

(defun deposit (account amount)
  "This shows a read and a write function call to
   get then set the balance"
  (let ((balance (balance account)))
    (setf (balance account)
          (+ balance amount))))

(defun withdraw (account amount)
  "A nice concise lisp version for withdraw"
  (decf (balance account) amount))

(add-account 'me)
=> 0
(deposit 'me 100)
=> 100
(balance 'me)
=> 100
(withdraw 'me 25)
=> 75
(balance 'me)
=> 75
```

This simple bank example has a significant vulnerability. If two threads read the same balance and one writes a new balance followed by the other, the second balance was written without access to the balance provided by the first and so the first transaction is lost.

The way to avoid this is to group a set of operations together, such as the read and write in `deposit` and `withdraw`. We accomplish this by establishing a dynamic context called a transaction.

During a transaction, all changes are cached until the transaction is committed. The changes made by a committed transaction happens all at once. Transactions can also be aborted due to errors that happen while they are active or because of contention. Contention is when another thread writes to a variable that the current transaction is reading. As in the bank example above, if one transaction writes the balance after the current one has read it, then the current one should start over so it has an accurate balance to work with. A transaction aborted due to contention is usually restarted until it has failed too many times.

The simplest and best way to use transactions in Elephant is to simply wrap all the operations in the `with-transaction` macro. Any statements in the body of the macro are executed within the same transaction. Thus we would modify our example above as follows:

```
(defun deposit (account amount)
  (with-transaction ()
    (let ((balance (balance account)))
      (setf (balance account)
            (+ balance amount)))))

(defun withdraw (account amount)
  (with-transaction ()
    (decf (balance account) amount)))
```

And presto, we have an ACID compliant, thread-safe, persistent banking system!

2.9.2 Using `with-transaction`

What is `with-transaction` really doing for us? It first starts a new transaction, attempts to execute the body, and commits the transaction if successful. If anytime during the dynamic extent of this process there is a conflict with another thread's transaction, an error, or other non-local transfer of control, the transaction is aborted. If it was aborted due to contention or deadlock, it attempts to retry the transaction a fixed number of times by re-executing the whole body.

And this brings us to two important constraints on transaction bodies: no dynamic nesting and idempotent side-effects.

2.9.3 Nesting Transactions

In general, you want to avoid nested uses of `with-transaction` statements over multiple functions. Nested transactions are valid for some data stores (namely Berkeley DB), but typically only a single transaction can be active at a time. The purpose of a nested transaction in data stores that support them is to break a long transaction into subsets. This way if there is contention on a given subset of variables, only the inner transaction is restarted while the larger transaction can continue. When the inner transaction commits its results, those results become part of the outer transaction but are not written to disk until the outer transaction commits.

If you have transaction protected primitive operations (such as `deposit` and `withdraw`) and you want to perform a group of such transactions, for example a transfer between accounts, you can use the macro `ensure-transaction` instead of `with-transaction`.

```
(defun deposit (account amount)
  "Wrap the balance read and the setf with the new balance"
  (ensure-transaction ()
    (let ((balance (balance account)))
      (setf (balance account)
            (+ balance amount)))))
```

```
(defun deposit (account amount)
  "A more concise version with decf doing both read and write"
  (ensure-transaction ()
    (decf (balance account) amount)))

(defun withdraw (account amount)
  (ensure-transaction ()
    (decf (balance account) amount)))

(defun transfer (src dst amount)
  "There are four primitive read/write operations
   grouped together in this transaction"
  (with-transaction ()
    (withdraw src amount)
    (deposit dst amount)))
```

`ensure-transaction` is exactly like `with-transaction` except it will reuse an existing transaction, if there is one, or create a new one. There is no harm, in fact, in using this macro all the time.

Notice the use of `decf` and `incf` above. The primary reason to use Lisp is that it is good at hiding complexity using shorthand constructs just like this. This also means it is also going to be good at hiding data dependencies that should be captured in a transaction!

2.9.4 Idempotent Side Effects

Within the body of a with-transaction, any non database operations need to be *idempotent*. That is the side effects of the body must be the same no matter how many times the body is executed. This is done automatically for side effects on the database, but not for side effects like pushing a value on a lisp list, or creating a new standard object.

```
(defparameter *transient-objects* nil)

(defun load-transients (n)
   "This is the wrong way!"
   (with-transaction ()
      (loop for i from 0 upto n do
         (push (get-from-root i) *transient-objects*))))
```

In this contrived example we are pulling a set of standard objects from the database using an integer key and pushing them onto a list for later use. However, if there is a conflict where some other process writes a key-value pair to a matching key, the whole transaction will abort and the loop will be run again. In a heavily contended system you might see results like the following.

```
(defun test-list ()
   (setf *transient-objects* nil)
   (load-transients)
   (length *transient-objects*))

(test-list 3)
```

```
=> 3

(test-list 3)
=> 5

(test-list 3)
=> 4
```

So the solution is to make sure that the operation on the lisp parameters is atomic if the transaction completes.

```
(defun load-transients (n)
  "This is a better way"
  (setq *transient-objects*
        (with-transaction ()
            (loop for i from 0 upto n collect
                  (get-from-root i)))))
```

(Of course we would need to use `nreverse` if we cared about the order of instances in `*transient-objects*`)

The best rule-of-thumb is to ensure that transaction bodies are purely functional as above, except for side effects to persistent objects and btrees.

If you really do need to execute side-effects into lisp memory, such as writes to transient slots, make sure they are idempotent and that other processes cannot read the written values until the transaction completes.

2.9.5 Transactions and Performance

By now transactions almost look like more work than they are worth! Fortunately, there are also performance benefits to explicit use of transactions. Transactions gather together all the writes that are supposed to made to the database and store them in memory until the transaction commits, and only then writes them to the disk.

The most time-intensive component of a transaction is waiting while flushing newly written data to disk. Using the default auto-committing behavior requires a disk flush for every primitive write operation. This is very, very expensive! Because all the values read or written are cached in memory until the transaction completes, the number of flushes can be dramatically reduced.

But don't take my word for it, run the following statements and see for yourself the visceral impact transactions can have on system performance.

```
(defpclass test ()
  ((slot1 :accessor slot1 :initarg :slot1)))

(time (loop for i from 0 upto 100 do
          (make-instance 'test :slot1 i)))
```

This can take a long time, well over a minute on the CLSQL data store. Here each new objects that is created has to independantly write its value to disk and accept a disk flush cost.

```
(time (with-transaction ()
            (loop for i from 0 upto 100 do
               (make-instance 'test :slot1 i))))
```

Wrapping this operation in a transaction dramatically increases the time from 10's of seconds to a second or less.

```
     (time (with-transaction ()
            (loop for i from 0 upto 1000 do
               (make-instance 'test :slot1 i))))
```

When we increase the number of objects within the transaction, the time cost does not go up linearly. This is because the total time to write a hundred simple objects is still dominated by the disk writes.

These are huge differences in performance! However we cannot have infinitely sized transactions due to the finite size of the data store's memory cache. Large operations (such as loading data into a database) need to be split into a sequential set of smaller transactions. When dealing with persistent objects a good rule of thumb is to keep the number of objects touched in a transaction well under 1000.

2.9.6 Transactions and Applications

Designing and tuning a transactional architecture can become quite complex. Moreover, bugs in your system can be very difficult to find as they only show up when transactions are interleaved within a larger, multi-threaded application.

In many cases you can simply ignore transactions. For example, when you don't have any other concurrent processes running. In this case all operations are sequential and there is no chance of conflicts. You would only want to use transactions to improve performance on repeated sets of operations.

You can also ignore transactions if your application can guarantee that concurrency won't generate any conflicts. For example, a web app that guarantees only one thread will write to objects in a particular session can avoid transactions altogether. However, it is good to be careful about making these assumptions. In the above example, a reporting function that iterates over sessions, users or other objects may still see partial updates (i.e. a user's id was written prior to the query, but not the name). However, if you don't care about these infrequent glitches, this case would still hold.

If these cases don't apply to your application, or you aren't sure, you will fare best by programming defensively. Break your system into the smallest logical sets of primitive operations (i.e. `withdraw` and `deposit`) using `ensure-transaction` and then wrap the highest level calls made to your system in with-transaction when the operations absolutely have to commit together or you need the extra performance. Try not to have more than two levels of transactional accesses with the top using with-transaction and the bottom using ensure-transaction.

See Section 4.11 [Transaction Details], page 49 for more details and Chapter 6 [Design Patterns], page 69 for examples of how systems can be designed and tuned using transactions.

2.10 Advanced Topics

The tutorial covers the essential topics and concepts for using Elephant. Many people will find that these features are the ones that are most often needed and used in ordinary applications.

More sophisticated uses of Elephant may require additional features that are covered in the user guide. The following is a list of major features in the user guide that were not covered in this tutorial.

- **Using Multiple Threads and Processes** What constraints must be accommodated to use Elephant data stores in multiple threads? What capabilities are there to share data stores among multiple processes or machines?
- **Class Heirarchies and Queries** There are some subtle issues to take into account when querying persistent classes. For example, how do you query a base class of type people to get subclass instances such as employee, manager, consultant, etc?
- **Derived Class Indices** You can create your own indices for classes that are arbitrary lisp functions of the persistent object.
- **Dynamic Class Index Management** It is possible to add and remove indexes from classes at runtime.
- **Class Definition/Database Conflict Resolution** When you startup lisp, there are potential conflicts between the class definition and the indexing records in the database. There are some constraints to account for and some facilities to manage how slots, class indices and
- **Indexed BTrees** Indexed BTrees are just like BTrees, except it is possible to add indexes which are BTrees who's values are primary keys in the parent `indexed-btree`. This allows for multiple ordering and groupings of the values of a BTree.
- **BTree Cursors** If you need to do more than iterate over a collection, or you need to delete elements of the collection as you iterate cursors are an important data structure. They implement a variety of operators for moving backward and forward over a btree, including ranged operations and iterating of duplicate or unique values.
- **Using the Map Operators** Mapping operators can be very efficient if properly utilized.
- **Using Multiple Stores** Multiple store controllers can be open simultaneously. However it does make the code more complex and you need to be careful about how you use them to avoid crashes and other unpleasant side effects.
- **Custom Transaction Architecture** You can implement your own version of `with-transaction` using the underlying controller methods for starting, aborting and committing transactions. You had better know what you are doing, however!
- **Handling Errors and Conditions** There are a variety of errors that can occur in Elephant that need to be dealt with by applications.
- **Deadlock Detection in Berkeley DB** Berkeley DB requires an external process to detect deadlock conditions among transactions. The :deadlock-detect keyword argument to open-store for Berkeley DB specs will launch this process on most lisps.

Further, see Chapter 6 [Design Patterns], page 69 for information about Elephant design patterns, solutions to common problems and other scenarios with multiple possible solutions.

3 Installation

3.1 Requirements

Elephant is a multi-platform, multi-lisp and multi-backend system. As such there is a great deal of complexity in testing. The system has tried to minimize external dependencies as much as possible to ease installation, but it still requires some patience and care to bring Elephant up on any given platform. This section attempts to simplify this for new users as much as possible. Patches and suggestions will be gladly accepted.

3.1.1 Supported Lisp, Platform and Data store combinations

Elephant supports SBCL, Allegro, Lispworks, OpenMCL and CMUCL. Each lisp is supported on each of the platforms it runs on: Mac OS X, Linux and Windows. As of release 0.6.1, both 32-bit and 64-bit systems should be supported.

Due to the small number of developers and the large number of configurations providing full test coverage is problematic. There are:

1. Five lisp environments
2. Three Operating System platforms
3. 32-bit or 64-bit OS/compilation configuration
4. Three data store configurations: Berkeley DB, SQLite3 and Postgresql

which means that the total number of combinations to be tested could be as much as:

$lisps * os * radix * dstore = 5 * 3 * 2 * 3 = 90 configurations$

Not all of these combinations are valid, but the implication is that not every combination will be tested in any given release. The developers and user base regularly use the following platforms

- 32/64-bit SBCL on Linux and Mac OS X
- 32-bit Lispworks on Windows and Mac OS X
- 32-bit Allegro on Mac OS X

The CLSQL backend is used predominantly under SBCL on Linux and Mac OS X at the time of writing. The developers will do their best to accomodate users who are keen to test other combinations, but the above configurations will be the most stable and reliable.

Elephant is now quite stable in general, so don't be afraid to try an unemphasized combination - chances are it is just a little more work to bring it up. In particular, Elephant can probably work with MySQL or Oracle with just a little work, but nobody has asked for this yet.

3.1.2 Library dependencies

The Elephant core system requires:

1. asdf – `http://www.cliki.net/asdf`
2. uffi – requires version 1.5.18 or later, `http://uffi.b9.com/` or `http://www.cliki.net/UFFI`
3. cl-base64 – `http://www.cliki.net/cl-base64`

4. gcc – Your system needs GCC (or Cygwin) to build the Elephant C-based serializer library. (Precompiled DLL's are available for Windows platforms on the download page.
5. rt – The RT regression test sytem is required to run the test suite: `http://www.cliki.net/RT`

Follow the instructions at these URLs to download and setup the libraries. (Note: uffi and cl-base64 are asdf-installable for those of you with asdf-install on your system). Elephant, however, is not asdf-installable today.

In addition to these libraries, each data store has their own dependencies as discussed in Section 3.4 [Berkeley DB], page 26 and Section 3.7 [CL-SQL], page 27.

3.2 Configuring Elephant

Before you can load the elephant packages into your running lisp, you need to setup the configuration file. Copy the reference file config.sexp from the root directory to my-config.sexp in the root directory. my-config.sexp contains a lisp reader-formatted list of key-value pairs that tells elephant where to find various libraries and how to build them.

For example:

```
#+(and (or sbcl allegro) macosx)
((:berkeley-db-include-dir . "/opt/local/include/db45/")
 (:berkeley-db-lib-dir . "/opt/local/lib/db45/")
 (:berkeley-db-lib . "/opt/local/lib/db45/libdb-4.5.dylib")
 (:berkeley-db-deadlock . "/opt/local/bin/db45_deadlock")
 (:compiler . :gcc))
```

The following is a guide to the various parameters. For simplicity, we include all the parameters here, although we will go into more detail in each of the data store sections.

- **:compiler** – This tells Elephant which compiler to use to build any C libraries. The only options currently are :gcc on Unix platforms and :cygwin for the Windows platform.
- **:berkeley-db-include-dir** – The pathname for the Berkeley DB include files (db.h)
- **:berkeley-db-lib-dir** – The pathname for all the Berkeley DB library files
- **:berkeley-db-lib** – The full pathname for the specific Berkeley DB library (libdb45.so)
- **:berkeley-db-deadlock** – The full pathname to the BDB utility function db_deadlock
- **:pthread-lib** – Not needed for SBCL 9.17+
- **:clsql-lib** – Currently unused, adds paths to the CL-SQL library search function

The config.sexp file contains a set of example configurations to start from, but you will most likely need to modify it for your system.

Elephant has one small C library that it uses for binary serialization. This means that you need to have gcc in your path (see Section 3.9 [Elephant on Windows], page 29 for exceptions on the Windows platform).

3.3 Loading Elephant

3.3.1 Loading Elephant via ASDF

Now that you have loaded all the dependencies and created your configuration file you can load the Elephant packages and definitions:

```
(asdf:operate 'asdf:load-op :elephant)
```

This will load the cl-base64 and uffi libraries. It will also automatically compile and load the C library. The build process no longer depends on a Makefile and has been verified on most platforms, but if you have a problem please report it, and any output you can capture, to the developers at `elephant-devel@common-lisp.net`. We will update the FAQ at `http://trac.common-lisp.net/elephant` with common problems users run into.

3.3.2 Two-Phase Load Process

Elephant uses a two-phase load process. The core code is loaded and the code for a given data store is loaded on demand when you call `open-store` with a specification referencing that data store. The second phase of the load process requires ASDF to be installed on your system.

(NOTE: There are some good reasons and not so good reasons for this process. One reason you cannot load ele-bdb.asd directly as it depends on lisp code defined in elephant.asd. We decided not to fix this in the 0.9 release although later releases may improve on this).

3.3.3 Packages

Now that Elephant has been loaded, you can call `use-package` in the cl-user package,

```
CL-USER> (use-package :elephant)
=> T
```

use a predefined user package,

```
CL-USER> (in-package :elephant-user)
=> T

ELE-USER>
```

or import the symbols into your own project package from :elephant.

```
(defpackage :my-project
  (:use :common-lisp :elephant))
```

The imported symbols are all that is needed to control Elephant databases and are documented in detail in Chapter 5 [User API Reference], page 57

3.3.4 Opening a Store

As discussed in the tutoral, you need to open a store to begin using Elephant:

```
(open-store '(:BDB "/Users/owner/db/my-bdb/"))
...
ASDF loading messages
...
=> #<BDB-STORE-CONTROLLER>

(open-store '(:CLSQL (:POSTGRESQL "localhost.localdomain"
                                  "mydb" "myuser" ""))))
```

```
...
ASDF loading messages
...
=> #<SQL-STORE-CONTROLLER>
```

The first time you load a specific data store, Elephant will call ASDF to load all the specified data store's dependencies, connect to a database and return the `store-controller` subclass instance for that data store.

3.4 Berkeley DB

The Berkeley DB Data Store started out as a very simple data dictionary in the Berkeley Unix operating system. There are many "Xdb" systems that use the same API, or a similarly one. A free for non-commercial use version of Berkeley DB is provided by Oracle corporation with commercial licenses available. Please follow the download and installation procedures defined here:

http://www.oracle.com/technology/products/berkeley-db/db/index.html

Elephant only works with version 4.5 of BerkeleyDB.

3.5 Setting up Berkeley DB

We recommend that you download and build a distribution from Oracle. Some problems have been reported with linking to Debian, Cygwin or other packages. This is especially true for Windows users.

Beyond ensuring that the file "my-config.sexp" points to your BDB installation directories and files, nothing else should be required to configure the example that uses a local "testdb" directory as a dabase (under "tests") in the top-level Elephant directory.

On one Fedora based system, the "my-config.sexp" file looked like this:

```
((:berkeley-db-include-dir . "/usr/local/BerkeleyDB.4.5/include")
 (:berkeley-db-lib-dir . "/usr/local/BerkeleyDB.4.5/lib")
 (:berkeley-db-lib . "/usr/local/BerkeleyDB.4.5/lib/libdb.so")
 (:berkeley-db-deadlock . "/usr/local/BerkeleyDB.4.5/bin/db_deadlock")
 (:pthread-lib . nil)
 (:clsql-lib . "/usr/local/share/common-lisp/")
 (:compiler . :gcc))
```

The Section 3.10 [Test Suites], page 29 give a nice example of using BDB by running the test using the specification:

```
'(:BDB "<elephant-root>/tests/testdb/")
```

Once you start working on an application, you will want to change the path to a directory that is appropriate for your application, and use that as the specification passed to `open-store` on application startup.

3.6 Upgrading Berkeley DB Databases

When there is a new release of Elephant, it will depend on a new version of Berkeley DB. If so, you must upgrade your BDB databases to use the new version Elephant. This forced upgrade is a consequence of Elephant not parsing the BDB header files which tend to change various important constants with each release. These patches are usually minor. Upgrading also happens because Elephant tries to leverage new features of Berkeley DB.

The rest of this section talks about how to upgrade your existing Berkeley DB databases, opening them in the new Elephant version and migrating them to a newly created Elephant database.

3.6.1 Upgrading to 0.9

This section outlines how to upgrade from Elephant version 0.6.0 and Berkeley DB 4.3.

1. Install BDB 4.5 (keep 4.3 around for now)
2. Setup my-config.sexp to point to the appropriate BDB 4.5 directories
3. Upgrade your existing database directory to 4.5
   - Run db43_recover in your 0.6 database
   - Optional: run db43_archive -d to remove all logs not part of a checkpoint This will make catastrophic recovery impossible, but reduces the amount of data you have to backup.
   - Backup your db files and remaining logs
   - Run db45_checkpoint -1 in the database directory
4. Upgrade 0.6 data to a fresh 0.9 database
   - Open your old database: `(setf sc (open-store '(:BDB "/Users/me/db/ele060/")))`
   - Run upgrade: `(upgrade sc '(:BDB "/Users/me/db/ele090/"))`
5. Test your new application and report any bugs that arise to elephant-devel@common-lisp.net

(NOTE: close-store may fail when closing the old 0.6 database, this is OK.)

(NOTE: 64-bit lisps will not successfully upgrade 32-bit 0.6 databases. Use a 32-bit version of your lisp to update to 0.9 and then open that database in your 64-bit lisp. There should be no compatibility problems. Best to test your application on a 32-bit lisp if you can, just to be sure.)

3.6.2 Upgrade from Elephant 0.5

Follow the upgrade procedures outlined in the Elephant 0.6.0 INSTALL file to upgrade your database from 0.5 to 0.6.0. Then follow the above procedures for upgrading to 0.9.

(NOTE: It may not take much work to make 0.9 upgrade directly from 0.5 However there are so few (none?) 0.5 users that it wasn't deemed worth the work given that there's an upgrade path available.)

3.7 CL-SQL

Although originally designed as an interface to the BerkeleyDB system, the original Elephant system has been extended to support the use of relational database management systems as the implementation of the persistent store. This relies on Kevin Rosenberg's CL-SQL interface, which provides access to a large number of relational systems.

A major motivation of this extension is that one one might prefer the licensing of a different system. For example, at the time of this writing, it is our interpretation that one cannot use the BerkeleyDB system behind a public website http://www.sleepycat.com/download/licensinginfo.shtml#redistribute unless one releases the entire web application as open source.

Neither the PostGres DBMS nor SQLite 3, nor Elephant itself, imposes any such restriction.

Other reasons to use a relational database system might include: familiarity with those systems, the fact that some part of your application needs to use the truly relational aspects of those systems, preference for the tools associated with those systems, etc.

Elephant provides functions for migrating data seamlessly between data stores. One can quite easily move data from a BerkeleyDB repository to a PostGres repository, and vice versa. This offers at least the possibility than one can develop using one data store, for example BerkeleyDB, and then later move to Postgres. One could even operate simultaneously out of multiple repositories, if there were a good reason to do so.

The SQL implementation shares the serializer with the BDB data store, but base64 encodes the resulting binary stream. This data is placed into a single table in the SQL data store.

All functionality except for nested transaction support and cursor-puts supported by the BerkeleyDB data store is supported by the CL-SQL data store. CL-SQL transaction integrity under concurrent operation has not been extensively stress tested.

Additionally, it is NOT the case that the Elephant system currently provides transaction support across multiple repositories; it provides transaction support on a per-repository basis.

The PostGres backend is currently about 5 times slower than the BerkeleyDB backend. As of the time of this writing, only PostGres and SqlLite 3 have been tested as CL-SQL backends.

3.8 CL-SQL Example

To set up a PostGres based back end, you should:

1. Install postgres and make sure postmaster is running. Postgres may be installed on your system; you may be able to use a package manager to install it, or you can install it from the PostgresSQL site directly (http://www.postgresql.org/).
2. Create a database called “test” and set its permissions to be reached by whatever connection specification you intend to use. The tests use:

   ```
   (defvar *testpg-path*
   '(:postgreql "localhost.localdomain" "test" "postgres" ""))
   ```

   which means that connections must be allowed to the database test, user “postgres”, no password, connected from the same machine “localhost.localdomain”. (This would be changed to something more secure in a real application.) Typically you edit the file : pg_hba.conf to enable various kinds of connections in postgres.
3. Be sure to enable socket connection to postgres when you invoke the postmaster.
4. Test that you can connect to the database with these credentials by running: `psql -h 127.0.0.1 -U postgres test` before you attempt to connect with Elephant.

Furthermore, you must grant practically all creation/read/write privileges to the user postgres on this schema, so that it can construct the tables it needs.

Upon first opening a CL-SQL based store controller, the tables, indexes, sequences, and so on needed by the Elephant system will be created in the schema named “test” automatically.

3.9 Elephant on Windows

The build process on Windows currently only works with GCC under Cygwin. The process can be a bit tricky, so if it doesn’t work out of the box or you don’t want to install cygwin, we recommend that you download the DLLs from the Elephant website download page (`http://www.common-lisp.net/project/elephant/downloads.html’`).

Unpack the .zip file into the elephant root directory. Ensure that your `my-config.sexp` file configuration for Windows has `:prebuilt-binaries` set to “t” so it will know to look in the elephant root during the asdf loading process.

For Berkeley DB users we recommend downloading the Windows binary distribution of Berkeley DB 4.5 to minimize any potential linking issues.

3.10 Test Suites

Elephant has matured quite a bit over the past year or two. Hopefully, it will work out-of-the-box for you.

However, if you are using an LISP implementation different than the ones on which it is developed and maintained (see Section 3.1 [Requirements], page 23) or you have a problem that you think may be a bug, you may want to run the test suites. If you report a bug, we will ask you to run these tests and report the output. Running them when you first install the system may give you a sense of confidence and understanding that makes it worth the trouble.

There are three files that execute the tests. You should choose one as a starting point based on what backend(s) you are using. If using BerekeleyDB, use

```
BerkeleyDB-tests.lisp
```

If using both, use both of the above and also use:

```
MigrationTests.lisp
```

The text of this file is included here to give the casual reader an idea of how elepant test can be run in general:

```
;; If you are only using one back-end, you may prefer:
;; SQLDB-test.lisp or BerkeleyDB-tests.lisp
(asdf:operate 'asdf:load-op :elephant)
(asdf:operate 'asdf:load-op :ele-clsql)
(asdf:operate 'asdf:load-op :ele-bdb)
(asdf:operate 'asdf:load-op :ele-sqlite3)

(asdf:operate 'asdf:load-op :elephant-tests)

(in-package "ELEPHANT-TESTS")

;; Test Postgres backend
(setq *default-spec* *testpg-spec*)
```

```
(do-backend-tests)

;; Test BDB backend
(setq *default-spec* *testbdb-spec*)
(do-backend-tests)

;; Test SQLite 3
(setq *default-spec* *testsqlite3-spec*)
(do-backend-tests)

;; Test a Migration of data from BDB to postgres
(do-migration-tests *testbdb-spec* *testpg-spec*)

;; An example usage.
(open-store *testpg-spec*)
(add-to-root "x1" "y1")
(get-from-root "x1")

(add-to-root "x2" '(a 4 "spud"))
(get-from-root "x2")
```

The appropriate test should execute for you with no errors. If you get errors, you may wish to report it the `elephant-devel at common-lisp.net` email list.

Setting up SQLite3 is even easier. Install SQLite3 (I had to use the source rather than the binary install, in order to get the dynamic libraries constructed.)

An example use of SQLLite3 would be:

```
(asdf:operate 'asdf:load-op :elephant)
(asdf:operate 'asdf:load-op :ele-clsql)
(asdf:operate 'asdf:load-op :ele-sqlite3)
(in-package "ELEPHANT-TESTS")
(setq *test-path-primary* '(:sqlite3 "testdb"))
(do-all-tests-spec *test-path-primary*)
```

The file RUNTESTS.lisp, although possibly not exactly what you want, contains useful example code.

You can of course migrate between the three currently supported repository strategies in any combination: BDB, Postgresql, and SQLite3.

In all probability, other relational datbases would be very easy to support but have not yet been tested. The basic pattern of the "path" specifiers is (cons clsqal-database-type-symbol (normal-clsql-connection-specifier)).

3.11 Documentation

If you are getting the documentation as a released tar file, you will probably find the documenation in .html or .pdf form in the release, or can find it at the Elephant website.

If you want to compile the documentation youself, for example, if you can think of a way to improve this manual, then you will do something similar to this in a shell or command-line prompt:

```
cd doc
make
make pdf
```

This process will populate the “./includes” directory with references automatically extracted from the list code. Currently this docstring extraction process relies on SBCL, but with minor modifications the scripts should work with other lisp environemnts.

The Makefile will then compile the texinfo documentation source into an HTML file and a PDF file which will be left in the “doc/” directory. An info style HTML tree is also created in the “doc/elephant” directory. This tree contains one node per HTML file.

Don’t edit anything in the “doc/elephant” directory or the “doc/includes” directories, as everything in these directories is generated. Instead, edit the “.texinfo” files in the doc directory.

4 User Guide

4.1 The Store Controller

An instance of the `store-controller` class mediates interactions between Lisp and a data store. All elephant operations are performed in the context of a store controller. To be more specific, a data store provides a subclass of `store-controller` specialized to that data store. Typically this object contains pointers to the disk files, foreign memory regions and any other necessary bookkeeping information to support Elephant operations such as slot writes and btree operations. The store also contains the root objects and other bookeeping common to all data stores.

To obtain a `store-controller` object, call the function `open-store` with a store controller specification. The current data store specification formats are:

- Berkeley DB: '(:BDB "/path/to/datastore/directory/")
- CLSQL: '(:CLSQL (<sql-db-name> <sql-connect-command>))

Valid CLSQL database tags for `<sql-db-name>` are `:SQLITE` and `:POSTGRESQL`. The `<sql-connect-command>` is what you would pass to CLSQL's `connect` command.

The open store function uses the first symbol in the specification (i.e. :BDB or :CLSQL) to dispatch instance creation to the specified data store which returns a specialized instance of `store-controller`. `open-store` then initializes the store using an internal call to `open-controller`.

The final step of `open-store` is to set the global variable `*store-controller*`. This special variable is used as a default value in the optional or keyword arguments to number of operations such as:

- `make-instance` for persistent objects
- `get-from-root` and `add-to-root` for accessing a store's root
- `make-btree` for creating persistent index instances

Each of these functions also accepts an explicit store controller argument for use in multiple store environments. Normal applications should only be aware that this global parameter is used. For further discussion of `*store-controller*` see Section 4.12 [Multi-repository Operation], page 51.

Additionally, `open-store` accepts data store specific keyword arguments. For example, you can force recovery to be run on Berkeley DB data stores:

```
(open-store *my-spec* :recover t)
```

The data store sections of the user guide (Section 4.17 [Berkeley DB Data Store], page 53 and Section 4.18 [CLSQL Data Store], page 56) list all the data-store specific options to various elephant functions.

When you finish your application, `close-store` will close the store controller. Failing to do this properly may lead to a need to run recovery on the data store during the next session. Again, see the relevant data store sections for more detail.

4.2 Serialization details

There are consequences to trying to move values from lisp memory onto disk in order to persist them. The first consequence is that that pointers cannot be guaranteed to be valid and so references to lisp objects cannot be maintained. This is very similar to the problems with passing references in foreign function interfaces. The second, and more frustrating limitation is that lisp operations that commit side effects on aggregate objects, such as objects, arrays, etc, cannot be trapped and replicated on the disk representation. This leads up to a very important consequence: all lisp objects are stored by *value*. This policy has a number of consequences which are detailed below.

4.2.1 Restrictions of Store-by-Value

1. **Lisp identity can't be preserved**. Since this is a store which persists across invocations of Lisp, this probably doesn't even make sense. However if you get an object from the index, store it to a lisp variable, then get it again - they will not be eq:

   ```
   (setq foo (cons nil nil))
   => (NIL)
   (add-to-root "my key" foo)
   => (NIL)
   (add-to-root "my other key" foo)
   => (NIL)
   (eq (get-from-root "my key")
         (get-from-root "my other key"))
   => NIL
   ```

2. **Nested aggregates are serialized recursively into a single buffer**. If you store an set of objects in a hash table you try to store a hash table, all of those objects will get stored in one large binary buffer with the hash keys. This is true for all aggregates that can store type T (cons, array, standard object, etc).
3. **Circular References**. One benefit provided by the serializer is that the recursive serialization process does not lead to infinite loops when they encounter circular references among aggregate types. It accomplishes this by assigning an ID to any non-atomic object and keeping a mapping between previously serialized objects and these ids. This same mapping is used to reconstruct references in lisp memory on deserialization such that the original structure is properly reproduced.
4. **Storage limitations**. The serializer writes sequentially into a contiguous foreign byte array before passing that array to a given data store's API. There are practical limits to the size of the foreign buffer that lisp can allocate (usually somewhere on the order of 10-100MB due to address space fragmentation). Moreoever, most data stores will have a practical limit to the size of a transaction or the size of key or value they will store. Either of these considerations should encourage you to plan to limit the size of objects that you serialize to disk. A good rule of thumb is to stay under a handful of megabytes. We have successfully serialized arrays over 100MB in the past, but have not tested the robustness of these large values over time.
5. **Mutated substructure does not persist**.

   ```
   (setf (car foo) T)
   => T
   ```

```
(get-from-root "my key")
=> (NIL)
```

This will affect all aggregate types: objects, conses, hash-tables, et cetera. (You can of course manually re-store the cons.) In this sense elephant does not automatically provide persistent collections. If you want to persist every access, you have to use Persistent Sets (see Section 4.5 [Persistent Sets], page 44) or BTrees (see Section 4.6 [Persistent BTrees], page 44).

6. **Serialization and deserialization can be costly**. While serialization is pretty fast, but it is still expensive to store large objects wholesale. Also, since object identity is impossible to maintain, deserialization must re-cons or re-allocate the entire object every time increasing the number of GCs the system does. This eager allocation is contrary to how most people want to use a database: one of the reasons to use a database is if your objects can't fit into main memory all at once.
7. **Merge-conflicts in heavily multi-process/threaded situations**. This is the common read-modify-write problem in all databases. We will talk more about this in the Section 4.11 [Transaction Details], page 49 section.
8. **Byte Ordering**. The primitive elements such as integers are written to disk in the native byte ordering of the machine on which the lisp runs. This means that little endian machines cannot read values written by big endian machines and vice a versa.
9. **Unicode codes and Serialized Strings**. The characters and strings stored to disk can store and recover lisp character codes that implement unicode, but the character maps are the lisp character maps (produced by `char-code`) and not strict unicode codes so lisps may not be able to interoperably read characters unless they have identical character code maps for the character sets you are reading and writing. All standard ASCII strings should be portable. Here is what we know about specific lisps, but this should not be taken as gospel.
   - SBCL: In versions with the :sb-unicode feature (after 0.8.17) `char-code` produces proper Unicode codes
   - Allegro: In the interational version, `char-code` produces proper Unicode codes for codes < 2^16
   - OpenMCL: OpenMCL 1.1 supports unicode, we are unsure about earlier versions
   - Lispworks: Lispworks 5 does not, to our knowledge, produce proper Unicode characters. (*This can be fixed on request iff users ask for it and are willing to pay the performance hit*)

4.2.2 Atomic Types

Atomic types have no recursive substructure. That is they cannot contain arbitrary objects and are of a bounded size. (Bignums are an exception, but they have a predictable structure and cannot reference or otherwise encapsulate other objects). The following is a list of atoms and a discussion of how they are serialized.

- `nil`: nil has it's own special tag in the serializer so it is easily identifiable. `nil` is an awkward value as it is also a boolean. The boolean value `t` is stored as the symbol 'T.
- **fixnums**: The serializer will store both 32-bit and 64-bit fixnums. Both types of fixnums are readable by a 32-bit or 64-bit lisp, but 64-bit fixnums are only written if the underlying lisp is supports fixnums between 32 and 64 bits.

- **bignums**: Bignums are broken into a sequence of fixnum-sized chunks and assembled by masking words onto the bignum. This is awfully expensive, but it's always correct and fully portable.
- **small-float**: Supported only on Lispworks 5 where type `small-float` is not equivalent to type `single-float` as it is on all other supported platforms. Written to disk and deserialized as a single float so any memory footprint savings of `small-float` is lost.
- **single-float**: 32-bit floating point numbers
- **double-float**: 64-bit floating point numbers
- **rational**: A rational is merely a ratio of two integers stored as fixnums or bignums.
- **complex**: A complex is a pair of floating point values, rationals or integers.
- **char**: Standalone chars are represented by their char-code and are stored in 32-bit format to ensure that all lisps are stored correctly.
- **strings**: Strings can be represented as 8, 16 or 32 bit sequences depending on the character sizes used in the underlying lisp. Because strings can be such a large percentage of on-disk space, Elephant uses a peculiar method of encoding strings. Strings are converted from their in-memory representation using `char-code`. The size of the first character dictates the word width used for encoding. If a character violates the word width, the string encoding is aborted and the next larger width is chosen. The rationale here is that many strings consist of Latin characters with codes less than 256. Strings stored in other character sets tend to all be of codes `>` 256. Therefore it is likely that the first character will properly determine the word size of the string. (*On request, we can easily make a configuration option to fix the word width for encoding*)
- **pathname**: A pathname is merely the `namestring` of the path object stored as a string. The path object is reconstructed from the namestring using `parse-namestring` during deserialization.
- **symbol**: Symbols are stored as two strings, the package name and the symbol name in that package. When deserialized, the target package is searched for and the symbol is interned in that package.

4.2.3 Aggregate Types

The next list are *aggregate* types, meaning that elements of that type can contain references to elements of type `T`. That means, in theory, that storing an aggregate type to disk that refers to other objects can copy every reachable object! This is a direct and dire consequence of the "store-by-value" restriction. (see Section 4.3 [Persistent Classes and Objects], page 37 for how to design around the store-by-value restriction).

This list describes how aggregates are handled by the serializer.

- **cons**: Cons is simply stored as a cons record containing two nested elements. Linear lists are not treated specially (i.e. no cdr-coding) by the serializer.
- **array**: Arrays are stored as sequences of nested, serialized elements. The array parameters are also stored so that arrays with fill pointers, adjustable arrays can be stored and reconstructed. The only arrays that cannot be reproduced are displaced arrays, which are copied by value and reconstructed as standard arrays during deserialization.
- **hash-table**: Hash tables are stored as a sequence of key-value pairs, where the key and value can be any serializable value. On deserialization, the reconstructed key and value

quantities are written incrementally into the hash table. The hash table does remember it's test, rehash size and threshold and it's total count. The final size of the new hash table is set to `(* (/ size reshash-threshold) rehash-size)`.

- **struct**: Structure objects are serialized using the metaprotocol. Each slot where the value is bound is serialized by serializing the slot name and the value in sequence. The underlying lisp must support the `struct-constructor` method so that a new, empty instance of the structure can be created and then populated by the stored keys and values.
- **object**: Instances of subclasses of standard-object are stored almost identically to structs. The type of the object is stored and the object slots with bound values are serialized as slotname-value pairs. To read an object of this type, the lisp image must have the class defined and it must have at least the slots that are stored on disk. There is no good method for schema evolution (redefining objects to have less slots) of ordinary classes.

One final strategic consideration is to whether you plan on sharing the binary database between machines or between different lisp platforms on the same machine. This is almost possible today, but there are some restrictions. In the section Section 4.14 [Repository Migration and Upgrade], page 51 we will discuss possible ways of migrating an existing database across platforms and lisps.

4.3 Persistent Classes and Objects

Persistent classes are instances of the `persistent-metaclass` metaclass. All persistent classes keep track of which slots are `:persistent`, `:transient` and/or `:indexed` and are used as specializers in the persistence meta-object protocols (initialization of slots, slot-access, etc).

All persistent classes create objects that inherit from the `persistent` class. The `persistent` class provides two slots that contain a unique object identifier (oid) and a reference to the `store-controller` specification they are associated with. Persistent slots do not take up any storage space in memory, instead the `persistent-metaclass` slot access protocol redirects slot accesses into calls to the store controller. Typically, the underlying data store will then perform the necessary serialization, deserialization to read and write data to disk.

When a reference to a `persistent` instance itself is written to the database, for example as a key or value in a `btree`, only the unique ID and class of the instance is stored. When read, a persistent object instance is re-created (see below). This means that serialization of persistent objects is exceedingly cheap compared to standard objects. The subsection on instance creation below will discuss the lifecycle of a persistent object in more detail.

4.3.1 Persistent Class Definition

To create persistent classes, the user needs to specify the `persistent-metaclass` to the class initarg `:metaclass`.

```
(defclass my-pclass ()
   ((slot1 :accessor slot1 :initarg :slot1 :initform 1))
   (:metaclass persistent-metaclass))
```

The only differences between the syntax of standard and persistent class definitions is the ability to specify a slot storage policy and an index policy. Slot value storage policies are specified by a boolean argument to the slot initargs `:persistent`, `:transient` and `:indexed`. Slots are `:persistent` and not `:indexed` by default.

The `defpclass` macro is provided as a convenience to hide the `:metaclass` slot option.

```
(defpclass my-pclass ()
   ((pslot1 :accessor pslot1 :initarg :pslot1 :initform 'one)
    (pslot2 :accessor pslot2 :initarg :pslot2 :initform 'two
            :persistent t)
    (tslot1 :accessor tslot1 :initarg :tslot1 :initform 'three
            :transient t)))
```

In the definition above the class `my-pclass` is an instance of the metaclass `persistent-metaclass`. According to this definition `pslot1` and `pslot2` are persistent while `tslot1` is transient and stored in memory.

Slot storage class implications are straightforward. Persistent slot writes are durably stored to disk and reads are made from disk and can be part of a ACID compliant transaction . Transient slots are initialized on instance creation according to initforms or initargs. Transient slot values are never stored to nor loaded from the database and their accesses cannot be protected by transactions. (Ordinary multi-process synchronization would be required instead).

The `:index` option tells Elephant whether to maintain an inverted index that maps slot values to their parent objects. The behavior of indexed classes and class slots are discussed in depth in Section 4.4 [Class Indices], page 43.

Persistent classes have their metaobject protocols modified through specializations on `persistent-metaclass`. These specializations include the creation of special slot metaobjects: `transient-slot-definition`, `persistent-slot-definition` and direct and effective versions of each. For the MOP aficionado the highlights of the new class initialization protocols are as follows:

- `shared-initialize :around` ensures that this class inherits from `persistent-object` and `persistent` if it doesn't already and that the class option `:index` results in class indexes being indexed;.
- `direct-slot-initialization-class` returns the appropriate slot metaobject based on the values of the `:transient` and `:persistent` slot definition keywords. It also does some simple error checking for invalid combinations, for example, indexed transient slots.
- `effective-slot-definition-class` performs the same role as the above for effective slots.
- `slot-definition-allocation` returns the `:database` allocation for persistent slot definitions so the underlying lisp will not allocate instance or class storage under some lisps.
- `compute-effective-slot-definition-initargs` performs some error checking to ensure a subclass does not try to make an inherited persistent slot transient.
- `finalize-inheritance` called before the first instance is created in order to finalize the list of persistent slots to account for any forward referenced classes in the inheritence

list. Similarly the list of indexed slots is computed. This function is also called by the class indexing code if any calls are made that depend on knowing which slots are indexed.

Reinitialization is discussed in the section on class redefinition.

4.3.2 Instance Creation

Persistent objects are created just like standard objects, with a call to `make-instance`. Initforms and slot initargs behave as the user expects. The call to `make-instance` of a persistent class will fail unless there is a default `store-controller` instance in the variable `*store-controller*` or the `:sc` keyword argument is provided a valid store controller object. The store controller is required to provide a unique object id, initialize the specification pointer of the instance and to store the values of any initialized slots. The initialization process is as follows:

- `initialize-instance :before` is called to initialize the `oid` slot and the data store specification slot `dbcn-spc-pst`. The oid is set by the argument `:from-oid` or by calling the store controller for a new oid.
- `shared-initialize :around` is called to ensure that the underlying lisp does not bypass the metaobject protocol during slot initialization by manually initializing the persistent slots and passing the transient slots to the underlying lisp. Finally it adds the instance to the class index so that any inverted indicies are updated appropriately.

Persistent slots are initialized only under the following conditions:

- An initarg is provided to `make-instance`
- The database slot value is unbound, an initform exists and from-oid was not specified

After initialization the persistent instance is added to its host store controller's object cache. This cache is a weak hash table that maps oids to object instances. So after initialization the following state has been created:

- **Placeholder Instance:** An instance of the class is in memory, containing storage for the oid, the specification reference, lisp instance data and any transient slot values. We call this the placeholder instance which mediates access to persistent values, but does not itself persist.
- **Cached Reference:** A weak reference to the instance is in the store controller object cache
- **Memory References:** A normal reference to the instance is (maybe) retained by the caller of `make-instance`.
- **Database Slot Values:** The data store contains the persistent slot values that were initialized, indexed by the object id and slot name.
- **Database References:** If the resulting placeholder instance was written to a persistent slot, added to a btree or the class is indexed, a **reference** to the instance was written into the data store. Today this reference consists of an oid and a class name. If this reference is reachable, then the persistent object can be reconstructed using the `:from-oid` argument.

If you mnanually create an object using an OID which already exists in the database, `initargs` to `make-instance` take precedence over existing values in the database, which in turn take precedence over any `initforms` defined in the class.

4.3.3 Persistent Instance Lifecycle

The distributed nature of persistent instance storage results in some interesting behaviors, especially with respect to transient slots. The prior section detailed the state of the system after the original initialization of an object. The object can then be in a number of different states:

- **Resident:** The canonical state of an in-use persistent object as described in the initialization section above.
- **Unreferenced, Unreclaimed:** All memory references to the object have been dropped but the placeholder instance has not yet been garbage collected. The weak pointer still exists in the cache. If a database reference is fetched from the data store, the cached value will be used.
- **Non-resident:** The object only exists as reachable database references and slot values. This is the state after garbage collection of the placeholder instance.
- **Recreated:** An intermediary state where a non resident object is fetched from the data store and its placeholder object must be recreated prior to the object enter the resident state.

The garbage collection of the placeholder instance is an important feature. This means that we can have more objects in our system than are currently resident in memory. If this were not the case, what would be the point of an object database?

The recreated state deserves to be discussed in more detail. We learned earlier that the database reference contains the oid and class of the object, and of course we know the store-controller the reference is stored into[1], so this information is sufficient to reconstruct the placeholder instance.

When the reference is deserialized, its oid is used to look up the object in the store controller's object cache. If this fails, then the instance is created with a call much like this:

```
(make-instance 'pclass :from-oid 2000 :sc *store-controller*)
```

The `:from-oid` argument to `make-instance` overrides some of the normal make-instance behavior by inhibiting all initform initialization as the object's slots are assumed to be properly initialized from the original call to `make-instance`.

4.3.4 Using Transient Slots

What about transient slots? Transients slots are tied to the placeholder object where their storage is allocated. While the persistent slots are permanently stored in the data store, transient slots can be garbage collected when all memory references have been dropped, even if database references exist.

After collection, if you retrieve an object from the store, its transient slots will be reset to the slot initforms from the class definition. You can only reliably use `:initargs` to initialize transient or persistent slots during the initial call to `make-instance` or when manually creating the instance from an oid.

Here is an example illustrating the ephemeral nature of transient slots:

[1] If you attempt to store an object from one store into another, the system will issue an error condition called `cross-reference-error`

```
(setf pobj1 (make-instance 'my-pclass :pslot1 1 :tslot3 3))
=> #<MY-PCLASS>

(pslot1 pobj1) => 1
(pslot2 pobj1) => 'two
(tslot1 pobj1) => 3

(add-to-root 'pobj1 pobj1)

(setf pobj2 (get-from-root 'pobj1))
=> #<MY-PCLASS>

(pslot1 pobj2) => 1
(pslot2 pobj2) => 'two
(tslot1 pobj2) => 3

(setf pobj1 nil)
(setf pobj2 nil)
(gc)

(setf pobj3 (get-from-root 'pobj1))
(pslot1 pobj2) => 1
(pslot2 pobj2) => 'two
(tslot1 pobj2) => 'three
```

The implications of this behavior is that you need to think carefully about how to use employ transient values. Essentially you cannot make assumptions about the state of transient values in objects loaded from the store unless you know that they were loaded at some point in time and cannot be GC'ed (i.e. they are stored in a list or hash table).

A good policy is to initialize transient values using an `:after` method on `initialize-instance`. This allows you to initialize transient values using either system defaults or persistent slot values. That way you can ensure that the transient slots are always in a consistent state when accessed by the application, regardless of when the placeholder object was recreated.

In general, transient slots are a good place for intermediate values in a computation or to cache frequently read items to avoid deserialization overhead. `indexed-btree` is an example of this approach, an in-memory hash is cached in the transient slot for reads and writes are mirrored to a serialized hash in a persistent slot. The `:after` method just copies the persistent hash value to the transient slot.

4.3.5 Using Persistent Slots

Persistent slot use is straightforward. You can read from them, write to them or make them unbound. Remember that every access goes to the data store. This makes reads relatively expensive as they may result in a disk seek. Writes can be doubly expensive, especially outside a transaction, as the write will result in a synchronous disk synch operation.

Reads and writes require the home store controller to be valid and open. The placeholder object's specification pointer is used to retrieve the `store-controller` object. If this object is closed or mising, the system will give you a restart option to reopen the controller and continue.

Persistent slot behavior is implemented by overloading the relevant MOP functions controlling slot access:

- `slot-value-using-class`
- `(setf slot-value-using-class)`
- `slot-boundp-using-class`
- `slot-makunbound-using-class`

Each of these functions retrieves the home store-controller for the instance and then calls a method specialized on the class of that store controller. This method is responsible for mapping the oid and slotname of the slot access to the appropriate value in the data store.

4.3.6 Class Redefinition

Class redefinition is problematic in the current (0.9) version of Elephant. The usual CLOS mechanisms are properly implemented, but updating instances will only work for those instances that are in memory at the time. Instances that are non-resident will not be updated. This is usually not as big a problem as it seems, because the slot values are stored independently. An outline of the update procedure follows:

The function `update-instance-for-redefined-class` is called by CLOS whenever `defclass` is re-evaluated and results in a change in the list of slots.

For transient slots the behavior is the same as it is in CLOS for all in-memory slots.

- Added slots: are added to the object and their initforms called just as if they were created without initargs
- Discarded slots: are dropped and their values lost

Persistent slots have a slightly different behavior, as only resident (those with valid placeholder objects) objects are updated.

- Added slots (resident): are added to the object and the initforms are called only on in-memory objects, as in an empty call to `make-instance`
- Added slots (non-resident): the added slots will have unbound values
- Discarded slots (resident): slots are dropped from the class and become inaccessible, but their values are not deleted from the database. This is a precautionary measure as losing persistent data because of an accidental re-evaluation while editing a defclass could be painful. If you add the slot back, the original value will be accessible regardless of the initform.
- Discarded slots (non-resident): This has the same behavior as resident objects, as no side effects are made on the objects or their slots

There are additional considerations for matching class indexing options in the class object to the actual indices in the database. The following section will discuss synchronizing these if they diverge.

(Note: release 0.9.1 should fix this by providing an oid->class map that allows the system to cheaply iterate over all objects and update them appropriately. This hasn't been done yet due to performance implications. See Trac system for the appropriate tickets)

4.3.7 Support for `change-class`

Elephant also supports the `change-class` by overloading `update-instance-for-different-class`. The handling of slots in this case is identical to the class redefinition above. Persistent and transient slot values are retained if their name matches a slotname in the new class and initforms are called on newly added slots. Valid initargs for any slot will override this default behavior and set the slot value to the initarg value.

Because the instance is guaranteed to be resident, the operation has none of the resident/non-resident conflicts above.

Change class cannot convert between persistent and non-persistent classes and will flag an error if you try to do so. *(Note: this could be implemented in the future if users request it)*

4.4 Class Indices

You can enable/disable class indexing for an entire class. When you disable indexing all references to instances of that class are lost. If you re-enable class indexing only newly created classes will be stored in the class index. You can manually restore them by using `find-class-index` to get the clas index BTree if you have an alternate in-memory index.

You can add/remove a secondary index for a slot. So long as the class index remains, this can be done multiple times without losing any data.

There is also a facility for defining 'derived slots'. These can be non-slot parameters which are a function of the class's persistent slot values. For example you can use an index to keep an alternate representation available for fast indexing. If an object has an x,y coordinate, you could define a derived index for r,theta which stored references in polar coordinates. These would be ordered so you could iterate over a class-index to get objects in order of increasing radius from the origin or over a range of theta.

Beware, however, that derived indices have to compute their result every time you update any persistent instance's slot. This is because there is no way to know which persistent slots the derived index value(s) depends on. Thus there is a fairly significant computational cost to objects with frequent updates having derived indices. The storage cost, however, may be less as all that is added is the index value and an OID reference into the class index. To add a slot value you add a serialized OID+class-ref+slotname to index value which can be much larger if you use long slotnames and package names and unicode.

Thus, the question of if and how a given class should be indexed is very flexible and dynamic, and does not need to be determined at the beginning of your development. This represents the ability to "late bind" the decision of what to index.

In general, there is always a tradeoff: an indexed slot increases storage associated with that slot and slows down write operations. Reads however remain as fast as for unindexed persistent slots. The Elephant system makes it simple to choose where and when one wants to utilize this tradeoff.

Finally, that file '`src/elephant/classindex-utils.lisp`' documents tools for handling class redefinitions and the policy that should be used for synchronizing the classes with the database. This process is somewhat user customizable; documentation for this exists in the source file referenced above.

4.4.1 Synchronizing Classes and Data Stores

Sometimes you may change a defclass form and then connect to a database with instances that do not match the current defclass definition. Because of the defclass behavior above, there is no need to detect this case as the behavior will be as if all instances were non-resident at redefinition time. However, this is an issue for indexed classes as the cost of indexing is high. There is a synchronization policy which updates either the class or the online class indexing mechanism at the time you try to perform an index operation (i.e. when `find-class-index` is called).

A policy is selected by setting the value of `*default-indexed-class-synch-policy*` with the appropriate policy:

- :class - The class is the master, and indices are deleted for any slots that are no longer indexed
- :db - The database is the master and the class indexing annotations are updated so that the slots that satisfy `class-indexedp-by-name` are isomorphic to the existing indices in the db.
- :union - This does what you would expect, updates the class to match any existing indices and creates new indices.

Derived slots can be problematic as they may depend on slot values that no longer exist in the changed defclass. This will result in an error, so for now you will have to manage any mismatches such as this yourself.

Note: release 0.9.1 should fix both mismatches and performance issues related to derived indices by allowing the user to provide hints as to which slot values the index depends. This will allow the system to only update when the appropriate slots change and to delete or inhibit derived indicies when slots are deleted. We will also improve error handling for this case, so you can delete the derived index and continue performing the write to a persistent object that flagged the error.

4.5 Persistent Sets

Persistent sets are fairly straightforward and are well-introduced by the tutorial, please review the tutorial or read the reference section for persistent sets.

4.6 Persistent BTrees

A BTree is a data structure designed for on-disk databases. It's design goal is to minimize the number of disk seeks while traversing the tree structure. In contrast to a binary tree, the BTree exploits the properties of memory/disk data heirarchies. Disk seeks are expensive while loading large blocks of data is relatively inexpensive and in-memory scanning of a block of memory is much cheaper than a disk seek. This means a few, large nodes containing many keys is a more balanced data structure than

The BTree, or derivatives, are the basis of most record-oriented database including SQL servers, Berkeley DB and many others. Elephant directly exposes the BTree structure to the user so the user can decide how best to manage and traverse it. Many of Elephant's other facilities, such as the class indexing discussed above, are implemented on top of the BTree.

The basic interface to the BTree is via the `get-value` method. Both the key and the value are serialized and then the BTree is traversed according to the sorted order of the key and the value inserted in its sorted order. Insertion, access and deletion (via `remove-kv`) are all O(log N) complexity operations.

Sorting in BTrees requires some discussion. The sorting constraints on btrees are dictated by the original implementation on Berkeley DB. The Berkeley DB data store sorts keys based on their serialized representation. The CLSQL implementation has to sort based on the deserialized lisp value, so sorted traversals require reading all the objects into memory. This places some limitations on systems that exploit the CLSQL implementation (see Section 4.18 [CLSQL Data Store], page 56 for more information).

Sorting is done first by primitive type (string, standard-class, array, etc) and then by value within that type. The type order and internal sorting constraint is:

1. Numbers. All numbers are sorted as a class by their numeric value. Effectively all numbers are coerced into a double float and sorted relative to each other.
2. Strings. Because the serializer stores strings in variable width structures. Each width type is sorted separately, then sorted lexically. (NOTE: This should get fixed for 1.0. Strings should be sorted together)
3. Pathnames. Sorted by their string radix then lexically.
4. Symbols. Sorted by string radix, then lexically.
5. Aggregates. Sorted by type in the following order, then arbitrarily internally. Persistent instance references, cons, hash-table, standard objects, arrays, structs and then nil.

String comparisons are case insensitive today, so `"Adam" = "adam" > "Steve"` . When unicode support is finalized, comparisons will be case sensitive.

Like persistent sets, BTrees are not garbage collected so to recover the storage of a BTree, just run the function `drop-btree` to delete all the key-value pairs and return their storage to the database for reuse. The oid used by the btree, however, will not be recovered.

4.7 BTree Cursors

Aside from getting, setting and dropping key-value pairs from the database, you can also traverse the BTree structure one key-value pair at a time.

Cursors must be created in the context of an active transaction (i.e. a `with/ensure-transaction` body). A cursor is made through a call to the `make-cursor` method of the BTree you wish to traverse.

An existing cursors can also be duplicated within the same transaction by calling `cursor-duplicate` which avoids the overhead of setting a second cursor to the same location.

Cursors can be in two states: initialized and uninitialized. Section 5.6 [BTree Cursor API], page 62 for details.

To initialize a cursor, you have to use one of the initializing functions to select a key-value pair in the btree.

- `cursor-first` and `cursor-last`: initialize the cursor to the first and last element of the btree, respectively.
- `cursor-set` and `cursor-set-range`: Sets the cursor to the first key-pair values according to the specified key. If the set fails, the cursor will remain uninitialized. The

ranged set will set it to the first key-value pair where the key is equal to or greater than the key argument.

A valid cursor will return multiple values: `(exists? key value)`. The first argument tells whether or not the cursor is initialized and pointing at a proper value. The second two arguments are self-explanatory.

`cursor-current` returns the current state of the cursor, nil if it is uninitialized.

Once a cursor is properly initialized, it can be incremented or decremented, a simple constant-time operation on BTrees.

`cursor-next` and `cursor-prev` move the cursor a single step forward or back across the sorted key-value pairs. `cursor-next` moves in ascending order, `cursor-prev` in descending order.

Finally cursors can be used for side effects on the current key-value pair. The function `cursor-put` replaces the value (but does not increment the current value) and `cursor-delete` deletes the key-value pair and become uninitialized. It is a valid operation to use the `(setf get-value)` method while the cursor is active to change the value at the current cursor.

If cursors take place within a transaction, what happens when traversing a very large BTree? This depends on the data store policy regarding whether a cursor read locks its entire btree (or the subset that is being iterated over) or allows changes to any pairs its transaction has not changed. See your data store documenation for details.

4.8 BTree Indexing

One powerful feature of Elephant is the ability to add indexes to BTrees. An indexed btree is a subclass of the standard `btree` called `indexed-btree`. The indexed btree maintain a set of indices (instances of `btree-index`) which provide alternative ways of indexing into the values of the main btree.

Each index is itself a btree, but with the property that its values are matched to the keys of the main btree. That is if you have a btree with key-value pairs:

```
("henry" . #<name: Henry, age: 45>)
("larry" . #<name: Larry, age: 29>)
```

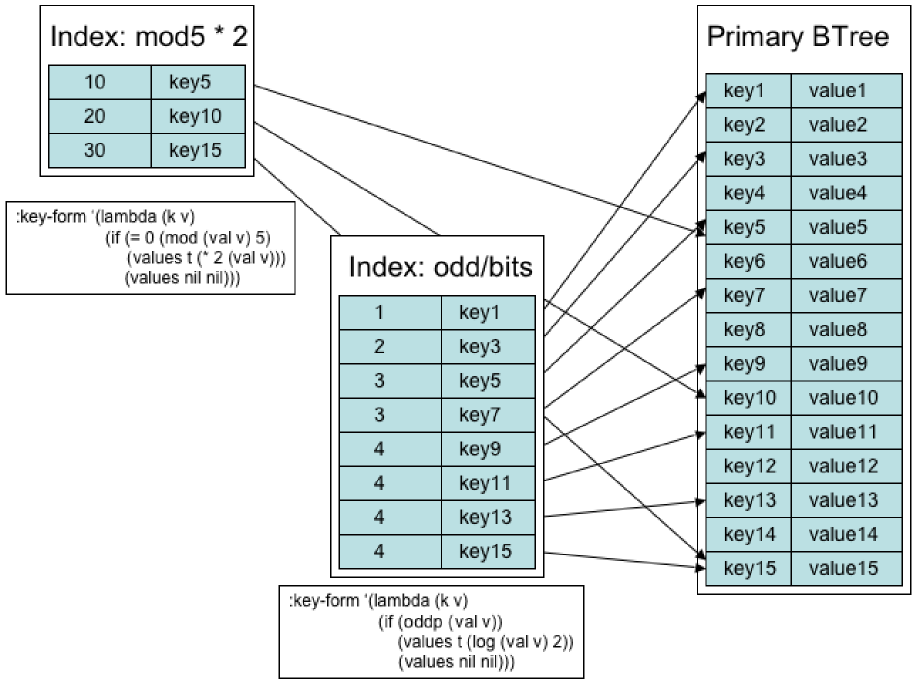
You can define an index that is populated by the age of the person object:

```
(29 . "Larry")
(45 . "Henry")
```

Now when you call `(get-value 29 index)` you get back `#<name: Larry, age: 29>`! Note also that these new pairs are ordered by age, the opposite of the alphabetic ordering of the names in the first two pairs. If you read through the tutorial, you may have guessed by now that this is the mechanism used to implement the class indexing capabilities previously described.

An index is created by using the `add-index` function. This function takes the `indexed-btree` you wish to index, an symbolic name for the index and a key-form which dictates how the index populates it's keys as a function of the main btree's keys and values. (It is a function of three arguments: the index itself, the key and the value).

A simple, contrived example is shown in the figure below:



Here we have a primary, indexed btree with a set of keys and values represented by symbols. We'll declare the function `val` to take a value symbol and extract it's number. The key-form in the `mod5 * 2` index is:

```
(lambda (k v)
  (if (= 0 (mod (val v) 5))
      (values t (* 2 (val v)))
      (values nil nil)))
```

When a key-value pair is written to the primary btree, the index is automatically updated through a call to the key-form. If the key-form above is called with `key1` and `value1`, `val` will return 1 which fails the if test. The second values statement, `(values nil nil)` indicates that this pair is not to be indexed. If I pass `key5` and `value5` to this same key form, I get back 10 as the `(val 'value5)` is 5 and `(= 0 (mod 5 5))` so the form returns `(values t 10)` meaning the index should add an index entry of 10 (`(* 2 5)`) associated with the key value `key5`.

So, of course, making the call `(get-value 10 index-mod5)` will return `value5`.

The second index in our little example calculates the number of bits in all odd numbered values. This illustrates an important property of the `btree-index`: it allows duplicate keys. Standard `btree` and `indexed-btree` classes are not allowed to have duplicate elements. The odd index allows us to ask simple questions like: "what are all the odd values with ids that fit into 4 bits?".

To extract this set, we have to use cursor functions specifically designed for the index that iterate over duplicate values.

4.9 Index Cursors

Index cursors are just like BTree cursors except you can get the main BTree value instead of the index value. There are also a parallel set of operations such as `cursor-pnext` instead of `cursor-next` which returns `exists`, `key`, `primary-btree-value` and `index-value = primary-btree-key`.

Operations that have the same behavior, but return primary btree values and keys are:

BTree Cursor Function		**Index Cursor Function**
`cursor-first`	=>	`cursor-pfirst`
`cursor-last`	=>	`cursor-plast`
`cursor-current`	=>	`cursor-pcurrent`
`cursor-next`	=>	`cursor-pnext`
`cursor-prev`	=>	`cursor-pprev`
`cursor-set`	=>	`cursor-pset`
`cursor-set-range`	=>	`cursor-set-prange`

The big difference between btree cursors and index cursors is that indices can have duplicate key values. This means we have to choose between incrementing over elements, unique key-values or only within a duplicate segment. There are cursor operations for each:

- Simple move. Standard btree operations work plus `cursor-pnext` and `cursor-pprev`
- Move to a different key value. `cursor-pnext-nodup` and `cursor-pprev-nodup`
- Move to next duplicate key value. `cursor-pnext-dup` and `cursor-pprev-dup`

After incrementing through a set of duplicate items using a `xxx-dup` function, the last next operation returns nil indicating there are no more duplicates. The consequence of this is that the cursor is now uninitialized (`cursor-initialized-p`) and needs to be reset by a set or set both call.

See Section 5.7 [Index Cursor API], page 64 for further details.

4.10 Multi-threaded Applications

Elephant is thread-safe by design. Users should not have to think about threading except to follow a couple of simple rules.

1. Do not perform transactions across multiple threads
2. Do not perform add/remove index operations on indexed-btrees in more than one thread.

This and common coding sense should be sufficient! Elephant's internal design for thread safety employs a number of policies to try to minimize using lisp locks and simplify analysis of multi threaded interactions:

1. **Rely on the thread safety of the data store databases**
2. **Ensure transaction isolation**
3. **Minimize dependency on thread-local special variables**
4. **Protect shared resources for a given store controller**.
5. **A use policy for shared objects (above)**

4.10.1 Shared Resources

Elephant has a few shared resources which are protected by standard locks. These are:

- The store controller connection table
- The instance cache
- The circularity buffer pool for the serializer
- The buffer-stream pool in memutils

In some cases, and on some lisp platforms, we try to use a fast lock strategy for frequently accessed items (the resource pools and instance cache especially).

4.10.2 Data Store Thread Safety and Transactions

Both CLSQL and Berkelely DB backends are thread safe. In CLSQL this is by ensuring that every thread has it's own handle into the SQL libraries or sockets. Berkeley DB is reentrant and handles locking internally.

Elephant depends on these guarantees especially for the isolation properties of transactions. All operations in the context of a given transaction should be isolated and atomic. It is important that a transaction not be shared across threads, however.

4.10.3 Minimize Dependency on Thread-Local Specials

Elephant uses several global variables as default arguments. Most of these were removed leaving only a couple to handle:

- `*store-controller*`. Store controller objects can be shared between threads and if a user resets this variable in a local thread to another controller, there is no problem with that either. Users of multiple concurrent stores can specify the store controller to all elephant API commands that don't get it from a persistent object implicitely. `*current-transaction*`. This is always set to the proper null value globally and should not be reset in local threads. Instead, transactions take place in a dynamic context that rebinds this variable as a special with the current transaction. This allows for a dynamic transaction stack for data stores that can nest transactions or when two datastores are both doing transactions concurrently.

4.11 Transaction Details

Transactions are dynamic contexts in which all side effects to persistent slots and other persistent objects such as BTrees are guaranteed to have the ACID properties: atomicity, consistency, isolation and durability. On a normal exit from context, the side effects are committed as a group. On a non-local exit, the transaction is aborted.

For most users, the tutorial section Section 2.9 [Using Transactions], page 15 is the best introduction to transactions. This section adds to that by exposing some of the details of how it is implemented.

To reiterate, there are a few important restrictions to adhere to:

- `*current-transaction*` is reserved for use by the transaction system. Users should not override, manipulate or close over this variable.
- The body of a transaction cannot throw, signal or jump without aborting the transaction. Any non-local exit is considered an aborting event. Catch signals inside the transaction and return a value instead.
- The dynamic extent of a transaction body must stay within the same thread

4.11.1 `with-transaction` internals

The `with-transaction` macro wraps the body expression with an anonymous lambda expression. This closure is passed to a call to the `execute-transaction` generic function which is specialized to the current data store.

The only bookkeeping done by the macro is ensuring that the `:parent` argument is checked for the current dynamic transaction context. If it is not owned by the default or provided store controller, then it is not passed to `execute-transaction`. This maintains a continuous dynamic stack transactions through the with/ensure transaction macros, but allows for a single leaf transaction to another store controller.

Be very careful about mixing transactions between store controller. This facility was only added to ensure that migrate worked correctly.

The macro processes keywords arguments `:store-controller` (defaults to `*store-controller*`), `:parent` (defaults to `*current-transaction*`) and `:retries` and passes the remaining keywords to the call to `execute-transaction` allowing the user to pass data store specific transaction keywords to their preferred data store. The consumed keywords are analyzed and then passed on to `execute-transaction`.

Any non-standard keywords for a given data store will be ignored by other data store implementation of `execute-transaction` so portable programs should not use keywords that change the semantics of the transaction.

`ensure-transaction` only calls `execute-transaction` if it needs to create a fresh transaction. If the transaction in `*current-transaction*` exists and belongs to the store controller passed to `ensure-transaction` then it merely calls the transaction closure, relying on the environment that created the transaction to handle any exit procedures and determining whether to abort or commit.

`*current-transaction*` contains transaction records during the dynamic execution of a transaction. These records capture any data store specific bookkeeping as well as the store-controller that the transaction is associated with.

4.11.2 `execute-transaction` internals

See the Chapter 7 [Elephant Architecture], page 85 section for details on how execute-transaction works. It will provide some deeper insight into the transaction system.

4.11.3 Building your own transactional framework

Data stores are required to implement three primitive transaction methods: `controller-start-transaction`, `controller-abort-transaction` and `controller-commit-transaction`. These are wrappers for the data store's primitive transaction mechanism. If you use these, it is up to you to make sure that you properly manage nested transactions, maintain the state of `*current-transaction*` handle any automated retries you might want, and handle detecting

If you use these, you are on your own - it is easy to make mistakes with transactions and create very complex bugs that are hard to track down. Most users are much better off sticking with the two transaction macros and the underlying `execute-transaction` method.

4.11.4 Analyzing Dynamic Transaction Behavior

You can trace `elephant::execute-transaction` to see the sequence of calls that occur dynamically and detect where and how many transactions are and are not happening.

4.12 Multi-repository Operation

Elephant maintains a small hashtable that maps "database specifications" into actual `store-controller` objects.

The basic strategy is that the "database specification" object is stored in every persistent object and collection so that the repository can be found. In this way, objects that reside in different repositories can coexist within the LISP object space, allowing data migration or multiple user stores.

All persistent instances store their oid and a store-controller reference in internal slots. Slot access and other protocols use this to provide access. This executes an auto-transaction or joins a surrounding transaction if the `transaction-record` in `*current-transaction*` matches the store.

When operating with multiple stores and nested transactions there are some subtle issues to work around: how to avoid writing one store with a transaction created in the context of another. A nested or ensured transaction is only indicated in the call to `execute-transaction` if the store controllers match, otherwise a new transaction for that store is created.

4.13 Multiple Processes and Distributed Applications

Just start up two lisp images and connect to the same database. Transactions will ensure there is no interaction between processes. This has not been extensively tested, but should work without any problem. Any field experience will get reflected in this section of the manual

Distributed applications may be supported if the underlying SQL server or an appropriate Berkeley DB database is used. We provide no documentation nor have we heard of this use-case. This remains fertile ground for future investigation.

4.14 Repository Migration and Upgrade

This version of Elephant supports migration between store controllers of any backend type.

The tests `migrate1` - `migrate5` are demonstrations of this capability.

There is a single generic function `migrate` that is used to copy different object types to a target repository. It is assumed that typically migrate will be called on two repositories and all live objects (those reachable in the root or class-root) will be copied to the target repository via recursive calls to migrate for specific objects.

When persistent instances are copied, their internal pointer will be updated to point to the new repository so after migration the lisp image should be merely updated to refer to the target repository in the *store-controller* variable or whatever variable the application is using to store the primary controller instance.

There are some limitations to the current migration implementation:

1. Migrate currently will not handle circular list objects
2. Migrate does not support arrays with nested persistent objects
3. Indexed classes only have their class index copied if you use the top level migration. Objects will be copied without slot data if you try to migrate an object outside of a store-to-store migration due to the class object belonging to one store or another
4. Migrate assumes that after migration, indexed classes belong to the target store.
5. In general, migration is a one-time activity and afterwards (or after a validation test) the source store should be closed. Any failures in migration should then be easy to catch.
6. Each call to migration will be good about keeping track of already copied objects to avoid duplication. Duplication shouldn't screw up the semantics, just add storage overhead but is to be avoided. However this information is not saved between calls and there's no other way to do comparisons between objects across stores (different oid namespaces) so user beware of the pitfalls of partial migrations...
7. Migrate keeps a memory-resident hash of all objects; this means you cannot currently migrate a store that has more data than your main memory. (This could be fixed by keeping the oid table in the target store and deleting it on completion)
8. Migration does not maintain OID equivalence so any datastructures which index into those will have to have a way to reconstruct themselves (better to keep the object references themselves rather than oids in general) but they can overload the migrate method to accomplish this cleanly

Users can customize migration if they create unusual datastructures that are not automatically supported by the existing `migrate` methods. For example, a datastructure that stores only object OIDs instead of serialized object references will need to overload migrate to ensure that all referenced objects are in fact copied (otherwise the OIDs will just be treated as fixnums potentially leaving dangling references.

To customize migration overload a version of migrate to specialize on your specific persistent class type.

```
(defmethod migrate ((dst store-controller) (src my-class)))
```

In the body of this method you can call `(call-next-method)` to get a destination repository object with all the slots copied over to the target repository which you can then overwrite. To avoid the default persistent slot copying, bind the dynamic variable `*inhibit-slot-writes*` in your user method using `with-inhibited-slot-copy` a convenience macro.

4.15 Performance Tuning

Performance is usually measured in transactions per second. Database reads are cheap. To get more transactions throughput, consider setting

```
(db-env-set-flags (controller-environment *store-controller*)
                  1 :txn-nosync t)
```

or look at other flags in the Berkeley DB docs. This will greatly increase your throughput at the cost of some durability; I get around a 100x improvement. Durability can be recovered with judicious use of checkpointing and replication, though this is currently not supported directly by Elephant – see the sleepycat docs.

The serializer is definitely fast on fixnums, strings, and persistent things. It is fast but consing with floats and doubles. YMMV with other values, though I've tried to make them fast.

Use `with-transactions` to avoid many automatic transactions, for example you'll find that this construct

```
(dotimes (i 1000) (add-to-root "key" "value"))
```

is much slower than

```
(with-transaction ()
 (dotimes (i 1000) (add-to-root "key" "value"))))
```

since there's only 1 transaction in the latter. However storing transaction state requires allocated main memory of which there is a finite amount so do not make your transactions too large.

Use the persistent classes and collections; if you're using transactions correctly they should be much faster.

If you don't need transactions you can turn them off. Opening the DB in less concurrent / transactional modes will be supported very soon (it's just an argument change, I think.) However you will need to ensure that multiple threads do not interleave access so single user mode is not suitable for use in web servers or other typically multi-threaded applications.

4.16 Garbage Collection

Garbage collection is not implemented as part of the persistent object protocol. However, the migration (see Section 4.14 [Repository Migration and Upgrade], page 51) mechanism will consolidate storage and recover OIDs which is an effective offline GC. No online solution is currently anticipated.

4.17 Berkeley DB Data Store

This section briefly describes special facilities of the Berkeley DB data store and explains how persistent objects map onto it. Elephant was originally written targeting only Berkeley DB. As such, the design of Elephant was heavily influenced by the Berkeley DB architecture.

Berkeley DB is a C library that very efficiently implements a database by allowing the application to directly manipulate the memory pools and disk storage without requiring communication through a server as in many relational database applications. The library supports multi-threaded and multi-process transactions through a shared memory region that provides for shared buffer pools, shared locks, etc. Each process in a multi-process application is independently linked to the library, but shares the memory pool and disk storage.

The following subsections discuss places where Berkeley DB provides additional facilities to the Elephant interfaces described above.

4.17.1 Architecture Overview

The Berkeley DB data store (indicated by a `:BDB` in the data store specification) supports the Elephant protocols using Berkeley DB as a backend. The primary features of the BDB library that are used are BTree databases, the transactional subsystem, a shared buffer pool and unique ID sequences.

All data written to the data store ends up in a BTree slot using a transaction. There are two databases, one for persistent slot values and one for btrees. The mapping of Elephant objects is quite simple.

Persistent slots are written to a btree using a unique key and the serialized value being written. The key is the oid of the persistent object concatenated to the serialized name of the slot being written. This ordering groups slots together on the disk

4.17.2 Opening a Store

When opening a store there are several special options you can invoke:

- `:recover` tells Berkeley DB to run recovery on the underlying database. This is reasonably cheap if you do not need to run recovery, but can take a very long time if you let your log files get too long. This option must be run in a single-threaded mode before other threads or processes are accessing the same database.
- `:recover-fatal` runs Berkeley DB catastrophic recovery (see BDB documentation).
- `:thread` set this to nil if you want to run single threaded, it avoids locking overhead on the environment. The default is to run *free-threaded*.
- The `:deadlock-detect` launches a background process via the run-shell commands of lisp. This background process connects to a Berkeley DB database and runs a regular check for deadlock, freeing locks as appropriate when it finds them. This can avoid a set of annoying crashes in Berkeley DB, the very crashes that, in part, motivated Franz to abandon AllegroStore and write the pure-Lisp AllegroCache.

4.17.3 Starting a Transaction

Berkeley DB transactions have a number of additional keyword parameters that can help you tune performance or change the semantics in Berkeley DB applications. They are summaried briefly here, see the BDB docs for detailed information:

- `:degree-2` This option provides for cursor stability, that is whatever object the cursor is currently at will not change, however prior values read may change. This can significantly enhance performance if you frequently map over a btree as it doesn't lock the entire btree, just the current element. All transactions running concurrently over the btree can commit without restarting. The global parameter `*map-using-degree2*` determines the default behavior of this option. It is set to true by default so that map has similar semantics to lists. This violates both *Atomicity and Consistency* depending on how it is used.

- `:read-uncommitted` Allows reading data that has been written by other transactions, this avoids the current thread blocking on a read access (for example you are merely dumping a btree for inspection) so long as you don't care whether the data you read changes or not. This violates *Atomicity and Consistency* depending on how it is used
- `:txn-nosync` Do not flush the log when this transaction completes. This means that you lose the *Durability* of a transaction, but gain performance by avoiding the expensive sync operation.
- `:txn-nowait` If a lock is unavailable, have the underlying database return a deadlock message immediately, rather than blocking, so that the transaction restarts.
- `:txn-sync` This is the default behavior and specifies that the transaction log of the current transaction is flushed to disk before the transaction commit routine returns. This provides full ACID compliance.
- `:transaction` This argument is for advanced use. It tells the Berkeley DB transaction subsystem the transaction it should use rather than to create a new one. The `:parent` argument provides a parent transaction that can result in a true nested transaction.

4.17.4 Special Commands

The berkeley DB data store exports some special facilities that are not currently supported by other data stores.

- `optimize-layout`. This function provides an interface to tell Berkeley DB to try to reclaim freed storage from the file system. This is of limited utility as it can only shrink database by the number of empty pages at the end of the file. Depending on what storage you have deleted, this can end up being only a handful or even zero pages. This will work well if you recently ran an experiment where you created a bunch of new data, then deleted it all and want to reclaim the space (i.e. you had runaway loop that was creating endless objects).
- `db-bdb:checkpoint`. This internal function forces the transaction log to be flushed and all active data to be written to the database so that the logs and database are in synch. This is good to run when you want to delete old log files and backup your database files as a coherent, recoverable set. Run checkpoing, close the database and then manually run "db_archive -d" on the database to remove old logs. Finally, copy the resulting data to stable storage. Read the Berkeley DB docs for more details of backing up and checkpointing.

4.17.5 Performance Tuning

Performance tuning for Berkeley DB is a complex topic and we will not cover it here. You need to understand the Berkeley DB data store architecture, the transaction architecture, the serializer and other such parameters. The primary performance related parameters are described in config.sexp. They are:

- `:berkeley-db-map-degree2` - Improve the efficiency of cursor traversals in the various mapping functions. Defaults to true, meaning a value you just read while mapping may change before the traversal is done. So if you operate only on the current cursor location, you are guaranteed that it's value is stable.
- `:berkeley-db-cachesize -` Change the size of the buffer cache for Berkeley DB to match your working set. Default is 10MB, or about twenty thousand indexed class

objects, or 50k standard persistent objects. You can save memory by reducing this value.

4.18 CLSQL Data Store

Elephant uses Kevin Rosenberg's excellent CLSQL CLSQL lisp binding to relational databases (it does not use the ORM functionality offered by that package.) CLSQL interfaces to many databases (Postgres, MySQL, Oracle, ODBC, SQLite3, Microsoft SQL Server (via ODBC)). Right now, Elephant has been tested with Postgress and SQLite3. Probably getting it to work with one of the others will take a small amount of debugging; in principle there is no reason it won't work out of the box. We invite users to try other database, and will quickly incorporate patches needed to make them work.

Because CLSQL is very generic, the CLSQL interface does not offer any special feature as discussed in the previous section Section 4.17 [Berkeley DB Data Store], page 53.

4.18.1 Basic CLSQL Implementation

The CLSQL uses base64 encoding to store binary data as text directly. This has the advantage that it works with all databases, which tend to differ widely in their treatment of Binary Large Objects (BLOBs.) It imposes some obvious overhead.

The CLSQL implementation is structurally exactly the same as the BDB implementation. A single table is created to hold all (key,value) pairs. An index on the key column provides efficient key lookup. No additional indexing offered by the underlying databases is used. This has the advantage that the API is exactly the same as the BDB api, and all of the functional indexes, cursors, and secondary indexes work exactly the same way. It does not exploit the performance that a database-specific solution would offer (see Section 4.19 [Postmodern Data Store], page 56 for an example of such a system.

Our basic strategy is to leave the CLSQL interface as simple as possible, in order to work with as many databases as possible. When there is enough motivation to support a backend that is specific to one database (and therefore probably faster), such an interface can be placed into the "contrib" directory and migrated into the main code base as time allows the complete integration with the test suite.

4.19 Postmodern Data Store

The postmodern data store is not yet integrated. It should be documented for the forthcoming release 0.9.1 or 0.9.2.

This backend will presumably be much faster, when used against PostGres, than the generic CLSQL store.

4.20 Native Lisp Data Store

The native lisp data store is unimplemented. It is tentatively planned for a 1.1 release sometime in the distant future. Yes, this is a deliberatively vague declaration.

5 User API Reference

5.1 Store Controllers

Store controllers provide the persistent storage for CLOS objects and BTree collections. Any persistent operations must be done in the context of a store controller. The default store-controller is stored in a global variable.

`elephant:*store-controller*` [Variable]
: The store controller which persistent objects talk to.

[Class elephant:store-controller], page 91 is associated with the following user methods and macros:

`elephant:with-open-store` *spec* **&body** *body* [Macro]
: Executes the body with an open controller, unconditionally closing the controller on exit.

`elephant:open-store` *spec* **&rest** *args* [Function]
: Conveniently open a store controller. Set *store-controller* to the new controller unless it is already set (opening a second controller means you must keep track of controllers yourself. *store-controller* is a convenience variable for single-store applications or single-store per thread apps. Multi-store apps should either confine their *store-controller* to a given dynamic context or wrap each store-specific op in a transaction using with or ensure transaction

`elephant:close-store` **&optional** *sc* [Function]
: Conveniently close the store controller.

`elephant:get-from-root` *key* **&key** *sc* [Function]
: Get the value associated with key from the root. Returns two values, the value, or nil, and a boolean indicating whether a value was found or not (so you know if nil is a value or an indication of non-presence)

`elephant:add-to-root` *key value* **&key** *sc* [Function]
: Add an arbitrary persistent thing to the root, so you can retrieve it in a later session. Anything referenced by an object added to the root is considered reachable and thus live

`elephant:remove-from-root` *key* **&key** *sc* [Function]
: Remove something from the root by the key value

`elephant:root-existsp` *key* **&key** *sc* [Function]
: Test whether a given key is instantiated in the root

`elephant:map-root` *fn* **&key** *sc* [Function]
: Takes a function of two arguments, key and value, to map over all key-value pairs in the root

5.2 Persistent Objects

[Class elephant:persistent-metaclass], page 93 can be used as the :metaclass argument in a defclass form to create a persistent object. Slots of the metaclass take the :index and :transient keyword arguments and the class accepts the :index keyword argument.

`elephant:defpclass` *cname parents slot-defs* **&rest** *class-opts* [Macro]

Shorthand for defining persistent objects. Wraps the main class definition with persistent-metaclass

`elephant:drop-pobject` *inst* [Generic Function]

drop-pobject reclaims persistent object storage by unbinding all persistent slot values. It can also helps catch errors where an object should be unreachable, but a reference still exists elsewhere in the `db`. On access, the unbound slots should flag an error in the application program. `important:` this function does not clear the cached object instance or any serialized references still in the db. Need a migration or `gc` for that! drop-instances is preferred as it implements the proper behavior for indexed classes

5.3 Persistent Object Indexing

5.3.1 Indexed Object Accessors

`elephant:map-class` *fn class* **&key** *collect* [Function]

Perform a map operation over all instances of class. Takes a function of one argument, a class instance. Setting the collect keyword to true will return a list of the values returned from calls to fn on successive instances of the class.

`elephant:map-inverted-index` *fn class index* **&rest** *args* **&key** *start end value from-end collect* [Function]

map-inverted-index maps a function of two variables, taking key and instance, over a subset of class instances in the order defined by the index. Specify the class and index by quoted name. The index may be a slot index or a derived index.

Read the docstring for map-index for details on what the various keywords do.

`elephant:get-instances-by-class` *class* [Generic Function]

Retrieve all instances from the class index as a list of objects

`elephant:get-instance-by-value` *class slot-name value* [Generic Function]

Retrieve instances from a slot index by value. Will return only the first instance if there are duplicates.

`elephant:get-instances-by-value` *class slot-name value* [Generic Function]

Returns a list of all instances where the slot value is equal to value.

`elephant:get-instances-by-range` *class idx-name start end* [Generic Function]

Returns a list of all instances that match values between start and end. An argument of nil to start or end indicates, respectively, the lowest or highest value in the index

`elephant:drop-instances` *instances* **&key** *sc* [Function]
Removes a list of persistent objects from all class indices and unbinds any slot values

5.3.2 Direct Class Index Manipulation

`elephant:find-class-index` *class* **&key** *errorp sc* [Generic Function]
This method is the way to access the class index via the class object. We can always fetch it or we can cache it in the class itself. It returns an indexed-btree.

`elephant:find-inverted-index` *class slot* **&key** *null-on-fail* [Generic Function]
This method finds an inverted index defined on the class described by an instance of persistent-metaclass.

`elephant:make-class-cursor` *class* [Generic Function]
Define a cursor over all class instances

`elephant:with-class-cursor` *var* **&body** *body* [Macro]
Bind the var argument in the body to a class cursor on the index specified the provided class or class name

`elephant:make-inverted-cursor` *class name* [Generic Function]
Define a cursor on the inverted (slot or derived) index

`elephant:with-inverted-cursor` *var* **&body** *body* [Macro]
Bind the var argument to an inverted cursor on the index specified the provided class and index name

5.3.3 Dynamic Indexing API

`elephant:enable-class-indexing` *class indexed-slot-names* **&key** *sc* [Generic Function]
Enable a class instance index for this object. It's an expensive thing to support on writes so know that you need it before you do it.

`elephant:disable-class-indexing` *class* **&key** *errorp sc* [Generic Function]
Delete and remove class instance indexing and any secondary indices defined against it

`elephant:disable-class-indexing` (*class persistent-metaclass*) **&key** (*sc* ***store-controller***) (*errorp* **nil**) [Method]
Disable any class indices from the database, even if the current class object is not officially indexed. This ensures there is no persistent trace of a class index. Storage is reclaimed also

`elephant:add-class-slot-index` *class slot-name* **&key** *update-class populate sc* [Generic Function]
Add a per-slot class index option to the class index based on the class accessor method

`elephant:remove-class-slot-index` *class slot-name* **&key** *update-class sc* [Generic Function]

Remove the per-slot index from the db

`elephant:add-class-derived-index` *class name derived-defun* **&key** *update-class sc populate* [Generic Function]

Add a simple secondary index to this class based on a function that computes a derived parameter. `warning:` derived parameters are only valid on persistent slots. An arbitrary function here will fail to provide consistency on transient slots or global data that is not stored in the persistent store. Derived indexes are deleted and rebuilt when a class is redefined

`elephant:remove-class-derived-index` *class name* **&key** *update-class sc* [Generic Function]

Remove a derived index by providing the derived name used to name the derived index

5.4 Persistent Sets

Persistent sets are a simple persistent collection abstraction. They maintain an unordered collection of objects. Unlike the normal list-oriented sets of Lisp, persistent sets use the equivalent of `pushnew` such that only one copy of any object or value is maintained using the serializer's `equal` implementation.

`elephant:pset` [Class]

Class precedence list: `pset, persistent-collection, persistent, standard-object, t`

An unordered persistent collection of unique elements according to serializer equal comparison

`elephant:insert-item` *item pset* [Generic Function]

Insert a new item into the pset

`elephant:remove-item` *item pset* [Generic Function]

Remove specified item from pset

`elephant:find-item` *item pset* **&key** *key test* [Generic Function]

Find a an item in the pset using key and test

`elephant:map-pset` *fn pset* [Generic Function]

Map operator for psets

`elephant:pset-list` *pset* [Generic Function]

Convert items of pset into a list for processing

`elephant:drop-pset` *pset* [Generic Function]

Release pset storage to database for reuse

5.5 BTrees

Persistent collections inherit from [Class elephant:persistent-collection], page 95 and consist of the [Class elephant:btree], page 95, [Class elephant:indexed-btree], page 95 and [Class elephant:btree-index], page 95 classes. The following operations are defined on most of these classes. More information can be found in Section 4.6 [Persistent BTrees], page 44 and Section 4.8 [BTree Indexing], page 46.

`elephant:make-btree` **&optional** *sc* [Function]

Constructs a new BTree instance for use by the user. Each backend returns its own internal type as appropriate and ensures that the btree is associated with the store-controller that created it.

`elephant:get-value` *key bt* [Generic Function]

Get a value from a Btree.

Values are written to a btree using the `setf` method on `get-value`.

`elephant:remove-kv` *key bt* [Generic Function]

Remove a key / value pair from a BTree.

`elephant:remove-kv` *key* (*bt btree-index*) [Method]

Remove a key / value from the `primary` by a secondary lookup, updating `all` other secondary indices.

`elephant:existsp` *key bt* [Generic Function]

Test existence of a key / value pair in a BTree

`elephant:drop-btree` *bt* [Generic Function]

Delete all key-value pairs from the btree and render it an invalid object in the data store

`elephant:map-btree` *fn btree* **&rest** *args* **&key** *start end value from-end collect* **&allow-other-keys** [Generic Function]

Map btree maps over a btree from the value start to the value of end. If values are not provided, then it maps over all values. BTrees do not have duplicates, but map-btree can also be used with indices in the case where you don't want access to the primary key so we require a value argument as well for mapping duplicate value sets. The collect keyword will accumulate the results from each call of fn in a fresh list and return that list in the same order the calls were made (first to last).

These functions are only defined on indexed btrees.

`elephant:make-indexed-btree` **&optional** *sc* [Function]

Constructs a new indexed BTree instance for use by the user. Each backend returns its own internal type as appropriate and ensures that the btree is associated with the store-controller that created it.

`elephant:add-index` *bt* **&key** *index-name key-form populate* [Generic Function]

Add a secondary index. The indices are stored in an eq hash-table, so the index-name should be a symbol. key-form should be a symbol naming a function, or a list which defines a lambda `--` actual functions aren't supported. The function should take 3 arguments: the secondary `db`, primary key and value, and return two values: a boolean indicating whether to index this key / value, and the secondary key if so. If populate = t it will fill in secondary keys for existing primary entries (may be expensive!)

`elephant:get-index` *bt index-name* [Generic Function]

Get a named index.

`elephant:get-primary-key` *key bt* [Generic Function]

Get the primary key from a secondary key.

`elephant:remove-index` *bt index-name* [Generic Function]

Remove a named index.

This function is only valid for indexes.

`elephant:map-index` *fn index* **&rest** *args* **&key** *start end value from-end collect* **&allow-other-keys** [Generic Function]

Map-index is like map-btree but for secondary indices, it takes a function of three arguments: key, value and primary key. As with map-btree the keyword arguments start and end determine the starting element and ending element, inclusive. Also, start = nil implies the first element, end = nil implies the last element in the index. If you want to traverse only a set of identical key values, for example all nil values, then use the value keyword which will override any values of start and end. The collect keyword will accumulate the results from each call of fn in a fresh list and return that list in the same order the calls were made (first to last)

5.6 Btree Cursors

Cursors are objects of type cursor (see [Class elephant:cursor], page 96) which provide methods for complex traversals of BTrees.

`elephant:with-btree-cursor` *var* **&body** *body* [Macro]

Macro which opens a named cursor on a BTree (primary or not), evaluates the forms, then closes the cursor.

`elephant:make-cursor` *bt* [Generic Function]

Construct a cursor for traversing BTrees.

`elephant:cursor-close` *cursor* [Generic Function]

Close the cursor. Make sure to close cursors before the enclosing transaction is closed!

`elephant:cursor-duplicate` *cursor* [Generic Function]

Duplicate a cursor.

Each of the following methods return multiple values consisting of `(exists? key value)`.

`elephant:cursor-current` *cursor* [Generic Function]
: Get the key / value at the cursor position. Returns has-pair key value, where has-pair is a boolean indicating there was a pair.

`elephant:cursor-first` *cursor* [Generic Function]
: Move the cursor to the beginning of the BTree, returning has-pair key value.

`elephant:cursor-last` *cursor* [Generic Function]
: Move the cursor to the end of the BTree, returning has-pair key value.

`elephant:cursor-next` *cursor* [Generic Function]
: Advance the cursor, returning has-pair key value.

`elephant:cursor-prev` *cursor* [Generic Function]
: Move the cursor back, returning has-pair key value.

`elephant:cursor-set` *cursor key* [Generic Function]
: Move the cursor to a particular key, returning has-pair key value.

`elephant:cursor-set-range` *cursor key* [Generic Function]
: Move the cursor to the first key-value pair with key greater or equal to the key argument, according to the lisp sorter. Returns has-pair key value.

`elephant:cursor-get-both` *cursor key value* [Generic Function]
: Moves the cursor to a particular key / value pair, returning has-pair key value.

`elephant:cursor-get-both-range` *cursor key value* [Generic Function]
: Moves the cursor to the first key / value pair with key equal to the key argument and value greater or equal to the value argument. Not really useful for us since primaries don't have duplicates. Returns has-pair key value.

`elephant:cursor-delete` *cursor* [Generic Function]
: Delete by cursor. The cursor is at an invalid position, and uninitialized, after a successful delete.

`elephant:cursor-put` *cursor value* **&key** *key* [Generic Function]
: Overwrite value at current cursor location. Currently does not properly move the cursor.

5.7 Index Cursors

Index cursors are made the same way standard cursors are, with a call to `make-cursor`, except with the index as the argument instead of a standard btree. In addition to the standard cursor operations, which provide the direct key and value of a `btree-index`, the following class of "p" cursors work on an index and allow you to get the primary value of the `indexed-btree` that the `btree-index` belongs to.

They each return multiple values `(exists? key primary-value primary-key)`.

`elephant:cursor-pcurrent` *cursor* [Generic Function]
: Returns has-tuple / secondary key / value / primary key at the current position.

`elephant:cursor-pfirst` *cursor* [Generic Function]
: Moves the key to the beginning of the secondary index. Returns has-tuple / secondary key / value / primary key.

`elephant:cursor-plast` *cursor* [Generic Function]
: Moves the key to the end of the secondary index. Returns has-tuple / secondary key / value / primary key.

`elephant:cursor-pnext` *cursor* [Generic Function]
: Advances the cursor. Returns has-tuple / secondary key / value / primary key.

`elephant:cursor-pprev` *cursor* [Generic Function]
: Moves the cursor back. Returns has-tuple / secondary key / value / primary key.

`elephant:cursor-pset` *cursor key* [Generic Function]
: Moves the cursor to a particular key. Returns has-tuple / secondary key / value / primary key.

`elephant:cursor-pset-range` *cursor key* [Generic Function]
: Move the cursor to the first key-value pair with key greater or equal to the key argument, according to the lisp sorter. Returns has-pair secondary key value primary key.

`elephant:cursor-pget-both` *cursor key value* [Generic Function]
: Moves the cursor to a particular secondary key / primary key pair. Returns has-tuple / secondary key / value / primary key.

`elephant:cursor-pget-both-range` *cursor key value* [Generic Function]
: Moves the cursor to a the first secondary key / primary key pair, with secondary key equal to the key argument, and primary key greater or equal to the pkey argument. Returns has-tuple / secondary key / value / primary key.

`elephant:cursor-next-nodup` *cursor* [Generic Function]
: Move to the next non-duplicate element (with different key.) Returns has-pair key value.

`elephant:cursor-next-dup` *cursor* [Generic Function]
Move to the next duplicate element (with the same key.) Returns has-pair key value.

`elephant:cursor-pnext-nodup` *cursor* [Generic Function]
Move to the next non-duplicate element (with different key.) Returns has-tuple / secondary key / value / primary key.

`elephant:cursor-pnext-dup` *cursor* [Generic Function]
Move to the next duplicate element (with the same key.) Returns has-tuple / secondary key / value / primary key.

`elephant:cursor-prev-nodup` *cursor* [Generic Function]
Move to the previous non-duplicate element (with different key.) Returns has-pair key value.

`elephant:cursor-prev-dup` *cur* [Generic Function]
Move to the previous duplicate element (with the same key.) Returns has-pair key value.

`elephant:cursor-pprev-nodup` *cursor* [Generic Function]
Move to the previous non-duplicate element (with different key.) Returns has-tuple / secondary key / value / primary key.

`elephant:cursor-pprev-dup` *cur* [Generic Function]
Move to the previous duplicate element (with the same key.) Returns has-tuple / secondary key / value / primary key.

5.8 Transactions

`elephant:with-transaction` **&rest &body** *body* [Macro]
Execute a body with a transaction in place. On success, the transaction is committed. Otherwise, the transaction is aborted. If the body deadlocks, the body is re-executed in a new transaction, retrying a fixed number of iterations. If nested, the backend must support nested transactions.

The following functions are an advanced use of the transaction system. They may be useful if, or example, you want to integrate Elephant transactions with non-Elephant side-effects that you explicitely make transactional.

`elephant:controller-start-transaction` *store-controller* **&key &allow-other-keys** [Generic Function]
Start an elephant transaction

`elephant:controller-abort-transaction` *store-controller transaction* **&key &allow-other-keys** [Generic Function]
Abort an elephant transaction

`elephant:controller-commit-transaction` *store-controller transaction* **&key &allow-other-keys** [Generic Function]
Commit an elephant transaction

5.9 Migration and Upgrading

Upgrade is a call to Migrate with checks for compatability. The migrate methods are included here in case you wish to develop a more specific "partial upgrade" or "partial migrate" of data from one store to another instead of using the top-level copy which migrates all live objects.

`elephant:upgrade` *sc target-spec* [Generic Function]

Given an open store controller from a prior version, open a new store specified by spec and migrate the data from the original store to the new one, upgrading it to the latest version

`elephant:migrate` *dst src* [Generic Function]

Migrate an object from the src object, collection or controller to the dst controller. Returns a copy of the object in the new store so you can drop it into a parent object or the root of the dst controller

`elephant:migrate` (*dst store-controller*) (*src hash-table*) [Method]

Migrate each hash element as the types are non-uniform

`elephant:migrate` (*dst store-controller*) (*src array*) [Method]

We only need to handle arrays of type 't' that point to other objects; fixnum, float, etc arrays don't need to be copied

`elephant:migrate` (*dst store-controller*) (*src cons*) [Method]

`warning`: This doesn't work for circular lists

`elephant:migrate` (*dst store-controller*) (*src structure-object*) [Method]

Walks structure slot values and ensures that any persistent references are written back into the slot pointint to the new store

`elephant:migrate` (*dst store-controller*) (*src standard-object*) [Method]

If we have persistent objects that are unindexed and `only` stored in a standard object slot that is referenced from the root, then it will only be copied by recursing through the slot substructure just as the serializer will, but copying any persistent objects found

`elephant:migrate` (*dst store-controller*) (*src indexed-btree*) [Method]

Also copy the inverse indices for indexed btrees

`elephant:migrate` (*dst store-controller*) (*src btree*) [Method]

Copy an index and it's contents to the target repository

`elephant:migrate` (*dst store-controller*) (*src persistent*) [Method]

Migrate a persistent object and apply a binary (lambda (dst src) ...) function to the new object. Users can override migrate by creating a function that calls the default copy and then does stuff with the slot values. A dynamic variable: *inhibit-slot-copy* can be bound in the caller to keep the new object from having it's slots copied `after` *elephant:migrate* (*dst store-controller*) (*src store-controller*) [Method]

This method ensures that we reset duplicate object detection over the store-controller

`before` *elephant:migrate* (*dst store-controller*) (*src store-controller*) [Method]

This method ensures that we reset duplicate object detection over the store-controller

`before` *elephant:migrate* (*dst store-controller*) (*src persistent*) [Method]

This provides some sanity checking that we aren't trying to copy to the same controller. We also need to be careful about deadlocking our transactions among the two gets/puts. Each leaf migration should be in its own transaction to avoid too many write locks.

`elephant:migrate` (*dst store-controller*) *src* [Method]

Default: standard objects are automatically migrated

`elephant:migrate` (*dst store-controller*) (*src store-controller*) [Method]

Perform a wholesale repository migration from the root. Also acts as a poor man's `gc` if you copy to another store of the same type!

6 Design Patterns

This chapter explores different ways that Elephant can be used to solve common problems in user programs. The term "Design Pattern" may be overkill as there is no formal specification of patterns. However the goals is similar to classical design patterns: provide a coherent description of how to approach ceratain common problems using Elephant as an enabling tool.

Most of this chapter falls short of a tutorial in the application of a pattern. Instead it provides a conceptual guide to implementing the pattern along with some code examples to show how Elephant features are invoked to support the pattern.

The authors hope that users of Elephant will find this a good source of inspiration for how to apply Elephant to their own programs and that they will be motivated to contribute design patterns of their own.

6.1 Persistent System Objects

The simplest design pattern supported by Elephant is the use of persistent objects in the place of standard objects. Typically you can just modify the old class definition to inherit the `persistent-metaclass`. Depending on your application, objects may need to have transient slots for performance reasons. We'll create a dummy class to illustrate:

```
(defclass system-object ()
  ((appname :accessor system-appname :initarg :name)
   (url :accessor system-url :initarg :url)
   (state :accessor system-state :initarg :state :initform 'idle))
  (:metaclass persistent-metaclass))
```

When starting up your application you need to recover references to any persistent objects that were created in a prior session or initialize a new one.

If you are storing system objects in parameters, you can just call an initialization function on startup:

```
(defparameter *system* nil)

(defun initialize-system (appname)
  (let ((system-object (get-from-root '*system*)))
    (setf *system
          (if system-object system-object
              (make-instance 'system-object :name appname)))))

*system*
=> #<SYSTEM-OBJECT ...>
```

And now you can use your parameter as you did before. If you want to avoid calling initialization functions, you can just accesss system objects through functions instead of parameters.

```
(defparameter *system* nil)

(defun sys-object ()
  (unless *system
    (let ((appname (get-application-name))
          (url (get-system-url)))
      (setf *system* (make-instance 'system-object
                                    :name appname
                                    :url url))))
  *system*)

(sys-object)
=> #<SYSTEM-OBJECT ...>
```

One constraint to keep in mind is that slot access will be slower as it has to synchronize to disk. This is usually not noticable for objects that are accessed on the order of seconds instead of milliseconds. For objects read constantly, but where you want to save any written values it helps to have a transient slot to cache values. You can override some methods to ensure that the persistent value is always updated, but that reads happen from the cached value and that the cached value is restored whenever the object is loaded.

```
(defclass system-object ()
  ((appname :accessor system-appname :initarg :name)
   (url :accessor system-url :initarg :url)
   (laststate :accessor system-laststate :initarg :state
              :initform 'idle)
   (state :accessor system-state :initarg :state :transient t)
  (:metaclass persistent-metaclass))

(defmethod (setf system-state) :after (state (sys system-state))
  (setf (system-laststate sys) state))

(defmethod initialize-instance :after ((sys system-state) &rest rest)
  (declare (ignore rest))
  (when (slot-boundp sys 'laststate)
    (setf (system-state sys) (system-laststate sys))))
```

And now you have an instant read cache for a slot value. This pattern is used several times within the Elephant implementation.

6.2 File System Replacement

One of the more annoying time-wasting activities in programming is saving and restoring data from disk. Data in configuration files, static data such as graphics and other formats take time and attention away from solving the main problem and are additional sources of bugs. Because Elephant's serializer supports most lisp types, Elephant can greatly simplify ease these concerns and allow you to work directly with your natural in-memory representations with almost no work to encode/decode formats or manage files in the file system[1].

The simplest way to accomplish this is to simply open a store controller and initialize a key-value pair in the root btree as a instead of a filename and file data in some system

[1] Example provided by Ian Eslick, April 2007

directory. Like the initialization process described for standard objects, you can hide some of the details like this:

```
(defvar *resources* (make-hash-table))

(defun get-resource (name)
  (multiple-value-bind (value foundp) (gethash name *resources*)
    (if foundp
        value
        (multiple-value-bind (value foundp) (get-from-root name)
           (if foundp
               value
               (error "Resource named ~A was not initialized" name))))))

(defun set-resource (value name)
  (add-to-root name value)
  (setf (gethash name *resources*) value))

(defsetf get-resource set-resource)
```

Another simple metaphor is to use Elephant btrees as persistent hash tables that persist key-value pairs for you. We'll wrap the Elephant btree in a simple class to provide a little conceptual isolation.

```
(defclass phash ()
  ((btree :accessor phash-btree :initarg :btree
          :initform (make-btree))))

(defun make-persistent-hash (name)
  (let ((btree (get-from-root name)))
    (if btree
        (make-instance 'phash :btree btree)
        (let ((phash (make-instance 'phash)))
           (add-to-root name (phash-btree phash))
           phash))))

(defun getphash (key phash)
  (get-value key (phash-btree phash)))

(defun setphash (value key phash)
  (setf (get-value key (phash-btree phash)) value))

(defsetf getphash setphash)
```

Of course to make a proper abstraction we'd want to provide some conditions that allowed restarts that initialized values or allowed users to update the hash in the background and continue computation.

6.3 Checkpointing Conventional Program State

Another challenge for many programs is saving some subset of program state. This could involve checkpointing an evolving computation, keeping track of state for the purposes of 'undo' or enabling crash recovery at key points in the program's execution.

One approach is to transform all our program state into persistent objects. However if the use of program state is slot-access intensive, this can have a significant performance impact. To improve the performance of the application, careful use of transactions is needed which further complicates program design and operation.

Can Elephant be used to provide a simple solution that retains the in-memory performance that we want? Can we do all this without having to put a ton of persistence assumptions into our main program code? The answer is yes, assuming you are willing to explicitly checkpoint your code and adhere to some simple constraints in accessing your program objects.

6.3.1 Assumptions

To maintain processing speed and convenience we would like all our objects to be standard lisp objects without special harnesses that would interfere with applying the full power of lisp. At some point during execution, we want to store the current state of a set of objects to disk and yet make it easy to reproduce the original state at a later point in time. For simplicity, we'll limit ourselves to collections of CLOS objects.

A complication is that many programs have sets of interdependant objects. These could be complex program graphs, the state of an ongoing search process or a standard OO system that uses a bunch of different program object types to run. This means that we need to persist not just object state, but also references to other objects.

Using CLOS reflection we can provide a general solution to capturing objects, slot values and references. However to reproduce references, we'll need to be able to find the object referenced and the only way to do that is to store it as well. Thus we want to create a snapshot of a closed set of self-referential objects.

The assumptions underlying the snapshot mechanism is:

- **Use standard CLOS objects and references to other CLOS objects.** We need reflection to
- **Use standard hash tables to keep track of sets of objects.** Your program should use the hash table as an entry point to find objects. When objects are restored, just replace an existing hash table with the new one and access your objects that way. Any parts of your program that have pointers into your objects but are not themselves snapshotted, will need to be able to refresh their pointers in some way.
- **Find your root object (s) and know what is "reachable" from them.** Ensure that you aren't referring to standard objects outside those you want to store as they will be stored too (persistent object references are fine though). Make sure your root refers to objects that refers to other objects and so on such that all objects you want to store can be reached by some set of pointer traversals. Looping references are fine.

6.3.2 Snapshot Set

The snapshot implementation is called a `snapshot-set`. The next section will go into detail, but a walkthrough will help make it clearer[2].

A snapshot set is quite easy to use. Load the complete code and play with this simple walk through. The code can be located in the Elephant source tree under `src/conrib/eslick/snapshot-set.lisp`.

The first step is to create a `snapshot-set` object,

```
(setf my-set (make-instance 'snapshot-set))
```

and add it to the root so we don't lose track of it.

```
(add-to-root 'my-set my-set)
```

Then we need some objects to play with.

```
(defclass my-test-class ()
  ((value :accessor test-value :initarg :value)
   (reference :accessor test-reference :initarg :reference)))

(setf obj1 (make-instance 'my-test-class :value 1 :reference nil))
(setf obj2 (make-instance 'my-test-class :value 2 :reference obj1))
(setf obj3 (make-instance 'my-test-class :value 3 :reference obj2))

(register-object obj3 my-set)
(snapshot my-set)
```

Now your set should have persistent versions of all three classes that are reachable from `obj3`.

```
(map-set (lambda (x) (print (test-value x))) my-set)
=>
3
2
1
```

Of course such fully connected objects are not always common, so we'll demonstrate using hash tables to create root indexes into our objects and sidestep registration calls entirely. We'll create a fresh set to work with.

```
(setf my-set (make-instance 'snapshot-set))
(add-to-root 'my-set my-set)

(setf obj4 (make-instance 'my-test-class :value 4 :reference obj1))
(setf obj5 (make-instance 'my-test-class :value 5 :reference nil))

(setf hash (make-hash-table))
(setf (snapshot-root my-set) hash)

(setf (gethash 'obj3 hash) obj3)
(setf (gethash 'obj4 hash) obj4)
```

[2] Example provided by Ian Eslick, April 2007

```
(setf (gethash 'obj5 hash) obj5)

(snapshot my-set)
```

To properly simulate restoring objects, we need to drop our old hash table as well as clear the persistent object cache so the snapshot set transient object is reset.

```
(setf my-set nil)
(setf hash nil)
(elephant::flush-instance-cache *store-controller*)
```

Now we'll pretend we're startup up a new session.

```
(setf my-set (get-from-root 'my-set))
(setf hash (snapshot-root my-set))
```

The cache is automatically populated by the implicit `restore` call during snapshot-set initialization, and our hash table should now have all the proper references. We'll pull out a few.

```
(setf o4 (gethash 'obj4 hash))
(setf o3 (gethash 'obj3 hash))
(setf o2 (test-reference o3))

(not (or (eq o4 obj4)
         (eq o3 obj3)
         (eq o2 obj2)))
=> t
```

The new objects should not be eq the old ones as we have restored fresh copies from the disk.

If you review the setup above, `obj3` references `obj2` which references `obj1` and `obj4` also references `obj1`. So if the objects were properly restored, these references should be `eq`.

```
(eq (test-reference o2) (test-reference o4))
=> t
```

And finally we can demonstrate the restorative power of snapshot sets.

```
(remhash 'obj5 hash)

(gethash 'obj5 hash)
=> nil nil

(restore my-set)
(setf hash (snapshot-root my-set))

(gethash 'obj5 hash)
=> #<MY-TEST-CLASS ..> t

(test-value *)
=> 5
```

This means that while our set object was not reset, the restore operation properly restored the old reference structure of our root hash object. Unfortunately, in this implementation you have to reset your lisp pointers to get access to the restored objects.

A future version could traverse the existing object cache, dropping new references and restoring old ones so that in-memory lisp pointers were still valid.

6.3.3 Snapshot Set Implementation

In this section we walk through the implementation of the snapshot set in detail as it provides:

- Insight into constraints in serialization and lisp object identity
- How to leverage Elephant for some more sophisticated applications than persistent indices and class slots.
- Helps you understand a useful utility (that we may add to an extensions release in the future)

To generalize the behavior discussed above, we will define a new persistent class called a snapshot set. The set itself is a wrapper around the btree, but provides all the automation to store and recover sets of standard objects.

```
(defpclass snapshot-set ()
  ((index :accessor snapshot-set-index :initform (make-btree))
   (next-id :accessor snapshot-set-next-id :initform 0)
   (root :accessor snapshot-set-root :initform nil)
   (cache :accessor snapshot-set-cache
          :initform (make-hash-table :weak-keys t)
          :transient t)
   (touched :accessor snapshot-set-touched
            :initform (make-array 20 :element-type 'fixnum
                           :initial-element 0 :fill-pointer t
                           :adjustable t)
            :transient t))
  (:documentation "Keeps track of a set of standard objects
    allowing a single snapshot call to update the store
    controller with the latest state of all objects registered with
    this set"))
```

The set class keeps track of IDs, a set of cached objects in memory, the on-disk btree for storing instances by uid and the current uid variable value. Notice the use of the transient keyword argument for the cache.

There are two major operations supported by sets `snapshot` and `restore`. These save objects to disk and restore objects to memory, along with proper recovery of multiple references to the same object.

Additional operations are:

- Registration: Adding and removing objects from a set
- Root operations: Easy access to a single root hash table or object
- Mapping: Walk over all objects in a set

To enable snapshots, we have to register a set of root objects with the set. This function ignores objects that are already cached, otherwise allocates a new ID and caches the object.

```
(defmethod register-object ((object standard-object) (set snapshot-set))
    "Register a standard object.  Not recorded until
     the snapshot function is called on db"
    (aif (lookup-cached-id object set)
         (values object it)
         (let ((id (incf (snapshot-set-next-id set))))
   (cache-snapshot-object id object set)
   (values object id))))

  (defun lookup-cached-id (obj set)
    (gethash obj (snapshot-set-cache set)))

  (defun cache-snapshot-object (id obj set)
    (setf (gethash obj (snapshot-set-cache set)) id))
```

A parallel function registers hash tables. One very important invariant implied here is that the cache always contains objects that are eq and mapped back to a serialized object in the backing btree. There is no need, however, to immediately write objects to the store and this gives us some transactional properties: snapshots are atomic, consistent and durable. Isolation is not enforced by snapshots.

This means that the transient cache has to be valid immediately after the snapshot set is loaded from the data store.

```
  (defmethod initialize-instance :after ((set snapshot-set) &key lazy-load &allow-other-
    (unless lazy-load (restore set)))
```

This also has consequences for unregistration. Removing a root object should also result in the removal of all objects that are unreachable from other roots. However, since side effects are not permanent until a snapshot operation, we merely have to garbage collect id's that were not touched during a snapshot operation. This makes unregistration simple.

```
  (defmethod unregister-object (object (set snapshot-set))
    "Drops the object from the cache and backing store"
    (let ((id (gethash object (snapshot-set-cache set))))
      (when (null id)
        (error "Object ~A not registered in ~A" object set))
      (drop-cached-object object set)))
```

But snapshots are a little bit more work.

```
  (defmethod snapshot ((set snapshot-set))
    "Saves all objects in the set (and any objects reachable from the
     current set of objects) to the persistent store"
    (with-transaction (:store-controller (get-con
                                          (snapshot-set-index set)))
      (loop for (obj . id) in
                (get-cache-entries (snapshot-set-cache set))
            do
    (save-snapshot-object id obj set))
      (collect-untouched set)))

  (defun save-snapshot-object (id obj set)
    (unless (touched id set)
      (setf (get-value id (snapshot-set-index set))
    (cond ((standard-object-subclass-p obj)
  (save-proxy-object obj set))
((hash-table-p obj)
  (save-proxy-hash obj set))
(t (error "Cannot only snapshot standard-objects and hash-tables"))))
      (touch id set))
    id)

(defun collect-untouched (set)
    (map-btree (lambda (k v)
        (unless (touched k set)
  (remove-kv k (snapshot-set-index set))))
        (snapshot-set-index set))
    (clear-touched set))
```

We go through all objects in the cache, storing objects as we go via `save-snapshot-object`. This function is responsible for storing objects and hash tables and recursing on any instances that are referenced. Any object that is saved is added to a touch list so they are not stored again and we can mark stored instances for the `collect-untouched` call which ensures that newly unreachable objects are deleted from the persistent store. Any newly found objects are added to the in-memory cache which, being a weak array, should eventually drop references to objects that are not referred to elsewhere.

It should be noted that garbage objects not garbage collected from the weak-array based cache may be stored to and restored from the persistent store. However this is merely a storage overhead as they will eventually be dropped across sessions as there are no saved references to them.

Now when we serialize a standard object, all the slot values are stored inline. This means that by default, a slot that refers to a standard object would get an immediately serialized version rather than a reference. This of course makes it impossible to restore multiple references to a single object. The approach taken here is to instantiate a *proxy* object which is a copy of the original class and stores references to normal values in its slots. Any references to hashes or standard classes are replaced with a reference object that records the unique id of the object so it can be properly restored.

```
(defun save-proxy-object (obj set)
    (let ((svs (subsets 2 (slots-and-values obj))))
      (if (some #'reified-class-p (mapcar #'second svs))
(let ((proxy (make-instance (type-of obj))))
    (loop for (slotname value) in svs do
        (setf (slot-value proxy slotname)
        (if (reify-class-p value)
  (reify-value value set)
  value)))
    proxy)
obj)))
```

The function checks whether any slot value can be reified (represented by a unique id) and if so, makes a new proxy instance and properly instantiates its slots, returning it to the main store function which writes the proxy object to the btree.

On restore, we simply load all objects into memory.

```
(defmethod restore ((set snapshot-set))
  "Restores a snapshot by setting the snapshot-set state to the last
snapshot.  If this is used during runtime, the user needs to drop all
references to objects and retrieve again from the snapshot set.  Also
used to initialize the set state when a set is created, for example
pulled from the root of a store-controller, unless :lazy-load is
specified"
  (clear-cache set)
  (map-btree (lambda (id object)
       (load-snapshot-object id object set))
     (snapshot-set-index set)))

(defun load-snapshot-object (id object set)
  (let ((object (ifret object (get-value id (snapshot-set-index set)))))
    (cond ((standard-object-subclass-p object)
   (load-proxy-object id object set))
  ((hash-table-p object)
   (load-proxy-hash id object set))
  (t (error "Unrecognized type ~A for id ~A in set ~A"
                 (type-of object) id set)))))
```

If an object has a reference object in a slot, then we simply restore that object as well. `load-snapshot-object` accepts null for an object so it can be used recursively when a reference object refers to an object (via the unique id) that is not yet cached. The `load` functions return an object so that they can used directly to create values for writing slots or hash entries.

```
(defun load-proxy-object (id obj set)
  (ifret (lookup-cached-object id set)
 (progn
   (cache-snapshot-object id obj set)
   (let ((svs (subsets 2 (slots-and-values obj))))
     (loop for (slotname value) in svs do
  (when (setrefp value)
    (setf (slot-value obj slotname)
  (load-snapshot-object (snapshot-set-reference-id value) nil set))))
   obj)))
```

6.3.4 Isolating multiple snapshot sets

A brief note on how to separate out the objects you want to store from those you don't may be useful. We want to snapshot groups of inter-referential objects without sucking in the whole system in one snapshot. These object sets must be closed and fully connected. If the program consists of a set of subgraphs, a root element of each graph should be stored in a hash table that is then treated as the snapshot root.

- **Manual registration:** Objects without external references are easy, just `register` or `unregister` them from the `snapshot-set` as needed and then map over them to get them back.
- **Implicit registration:** Just store objects in a hash that is the root of a `snapshot-set` and you are good to go.
- **Graphs:** Graphs are easy to store as they naturally consist of a closed set of objects. If the graph nodes reference other system objects that you don't want to store, you'll need to implement something akin to the indirection provided here. Just store the root of the graph in the snapshot set root and go from there.
- **All instances of a type:** Another easy way to create sets is to overload `make-instance` to store all new objects in a weak hash table that is treated as the root of a `snapshot-set` (NOTE: I have not verified that weak hashes are properly serialized and reproduced - I suspect they are not so you might have to copy after a `restore`).

For more complex applications, you can isolate these closed sets of objects by using `snapshot-set` root hash tables as an indirection mechanism. Instead of storing direct references in an object slot or hash value, isolation is ensured by storing keys and indirecting through a hash table to get the target object. This can be hidden from the programmer in multiple ways. The easiest way is just to make sure that when you store references you store a key and overload the slot accessor. A sketch of this follows:

```
(defparameter *island1-hash* (make-hash-table))
(defparameter *island2-hash* (make-hash-table))
(defvar *unique-id* 0)

(defclass island1-object ()
  ((pointer-to-island1 :accessor child :initform nil)
   (pointer-to-island2 :accessor neighbor :initform nil)))

(defmethod neighbor :around ((obj island1-object))
  (let ((key (call-next-method)))
     (when key (gethash key *island2-hash*))))

(defmethod (setf neighbor) :around (ref (obj island1-object))
  (cond ((subtypep (type-of ref) 'island2-object)
         (let ((key (find-object ref *island2-hash*)))
           (if key
               (progn
                 (call-next-method key obj)
                 obj)
               (progn
                 (setf (gethash (incf *unique-id*) *island2-hash*) ref)
                 (call-next-method *unique-id* obj)
                 obj))))
        (t (call-next-method))))

(defun find-object (obj hash)
   (map-hash (lambda (k v)
               (declare (ignore k))
               (if (eq obj v)
                   (return-from find-object obj)))
              hash))
```

The same template would apply to `island2` references to `island1` objects. You could further simplify creating these hash table indirections with a little macro:

```
(defmacro def-snapshot-wrapper (accessor-name
          (source-classname target-classname hashname uid))
  (with-gensysms (obj key ref)
   `(progn
      (defmethod ,accessorname :around ((,obj ,source-classname))
         (let ((,key (call-next-method)))
           (when ,key (gethash ,key ,hashname))))
      (defmethod (setf ,accessorname) :around
                 (,ref (,obj ,source-classname))
         (cond ((subtypep (type-of ,ref) ,target-classname)
                (let ((,key (find-object ,ref ,hashname)))
                  (if ,key
                      (progn
                        (call-next-method ,key ,obj)
                        ,obj)
                      (progn
                        (setf (gethash (incf ,uid) ,hashname) ,ref)
                        (call-next-method ,uid ,obj)
                        ,obj))))
               (t (call-next-method)))))))

(defclass island2-object ()
  ((pointer-to-island2 :accessor child :initform nil)
   (pointer-to-island1 :accessor neighbor :initform nil)))

(def-snapshot-wrapper neighbor
                      (island2 island1 *island1-hash* *unique-id*))
```

Of course this doesn't work for multi-threaded environments, or for separating more complex collections of types. I am also sure that more elegant solutions are possible. In most cases, we assume the user will have a natural collection of objects that can be closed over by types or references so such efforts are unnecessary.

6.4 Elephant as Database

As we move beyond replacing standard objects with persistent objects and using Elephant to save conventional lisp data, we can exploit Elephant's advanced class indexing features and the query system. With these facilities, Elephant can be used as a full-fledged object oriented database system

NOTE: Will finish this section after the query engine is done as it will be more coherent/complete

6.5 Multithreaded Web Applications

Web applications can exploit all of the patterns described in previous sections. Each server thread can have transactional access to objects encapsulating user data, commercial transactions, database data, etc. Users can formulate queries against objects and get html rendering of the result views.

The most important characteristic of Elephant in these settings is that instances and `store-controller` objects in versions 0.9 and greater are automatically thread-safe. The only consideration in these cases is transaction design.

NOTE: What are common wrappers for Elephant that come up in web applications? Presentation functions?

NOTE: Should this section be supplanted by a full application example which uses most of the patterns above?

6.6 Real-World Application Examples

This section contains a collection of case studies or overviews of read-world applications that have exploited Elephant.

6.6.1 Konsenti

Elephant is used by Konsenti(tm), a for-profit company of Robert L. Read, one of the maintainers of Elephant. It can be visited at `http://konsenti.com`.

Konsenti uses the Data Collection Management (DCM) package, found in the `src/contrib/rread directory`. DCM provides prevalence-style in-memory write-through caching. The most enjoyable feature about Elephant for this project is that new Business Layer objects can be created without having to deal with an Object-Relational Mapping, enabling extremely rapid development.

All Business objects are persisted via a `director` in DCM (which sits on top of Elephant.) Many of these business objects are in fact finite state machines decorated with functions. The functions are represented by lambda s-expressions stored in slots on the business objects. A complete Message Factory and double-entry accounting system are also implemented as DCM objects. Binary objects, such as uploaded PDFs, can be attached to objects as comments and are stored directly in Elephant. Konsenti is based on utf-8, and unicode characters outside of the ISO-8859-1 character set are routinely stored in Elephant. Konsenti uses Postgres as a backend for licensing reasons; but use of other data stores is possible.

6.6.2 Conceptminer

Conceptminer is an Elephant-based web-mining framework developed by Ian Eslick (`http://www.media.mit.edu/~eslick`) that performs large-scale text analysis over the web to identify semantic relationships such as "PartOf", "DesireOf" and "EffectOf" between English phrases.

Elephant's persistence capability is used to keep full records of all source material, extracted relationships and search queries so that it is always possible to trace the source of a learned relation and to avoid repeated queries to web search engines. Conceptminer used Elephant 0.6.0 and the development branch of Elephant 0.9 to perform months of analysis consisting of millions of pages and a page/query database of over ten gigabytes.

There are several interesting uses and extensions of Elephant in Conceptminer:

- Bulk storage of post-processed web data: Elephant was used to store hundreds of thousands of processed web pages as strings, associate pages with queries and store related metadata.
- Derived index: a custom string hash function over URLs was used to populate a class derived index, allowing fast identification of pages from their URL without requiring expensive eql comparisons.
- Inverted document index: a (not terribly efficient) data structure that efficiently maps words to documents allowing pages to indexed by the words contained in them. Allowed for phrase and conjunction searches.
- User association data structure: a data structure based on oids that supports general one-to-many mappings between classes. Had a custom migrate method to support migration of associations. Supplanted by persistent sets as of 0.9.

The most interesting use of Elephant was extending its transactional architecture to cover in-memory lisp operations. `PCOMP` (Process Components) is a framework for constructing and managing simple, dataflow-style multi-threaded applications in Common Lisp. The goal is to simplify the process sufficiently so that the ordinary user can hide from many of the details associated with aborting transactions. To this end, the model provides for safe, asynchronous communications among a set of components which may be scheduled together in a single process or communicate across separate threads (and potentially processes). Components are packaged into a system inside a Container object which schedules execution and mediates communications.

Communications between components can be in a dataflow style or using messages. Each component has a single port for receiving incoming data items. These items, if access is shared among components, should have the proper synchronization protections on mutating accesses. There is also an asynchronous communications method allowing you to send messages to components with particular names.

The basic building block is a component. Components are defined using the defcomponent form and contain several major elements, such as:

```
(defcomponent counter
  (:vars (count 0) end (increment 1))
  (:initialize (assert end))
  (:body
     (when (>= (incf count increment) end)
        (terminate))))
```

The arguments to defcomponent behave as follows:

- `:vars` - Values that the body wants to retain between invocations
- `:initialize` - A reserved message handler called at the begining of time
- `:body` - A body expression that is executed whenever data has arrived

The body and messages are evaluated in a very specific environment. Within the body certain variables and functions are bound:

- Variables:
    - `data` - The current data item

  - `self` - The component object
  - `"vars"` - all variables named in `:vars` are bound using `symbol-macrolet` and available as in a `let` statement. Any side effects to those vars are visible, but not saved to th component state until the component commits (see below).
- Functions:
  - `(terminate)`
  - `(send data)`
  - `(receive data)`
  - `(get-ctrl-msg target type data)`
  - `(pause)`
  - `(abort)`

Each component execution is bound in a transactional framework. No variables are written, messages consumed or messages sent until the body or control handler has exited normally. Users can tap into this transactional framework by overriding `start-transaction`, `commit-transaction` and `abort-transaction` methods for the component class. Transactional variables are implemented via `:after` methods on these generic functions.

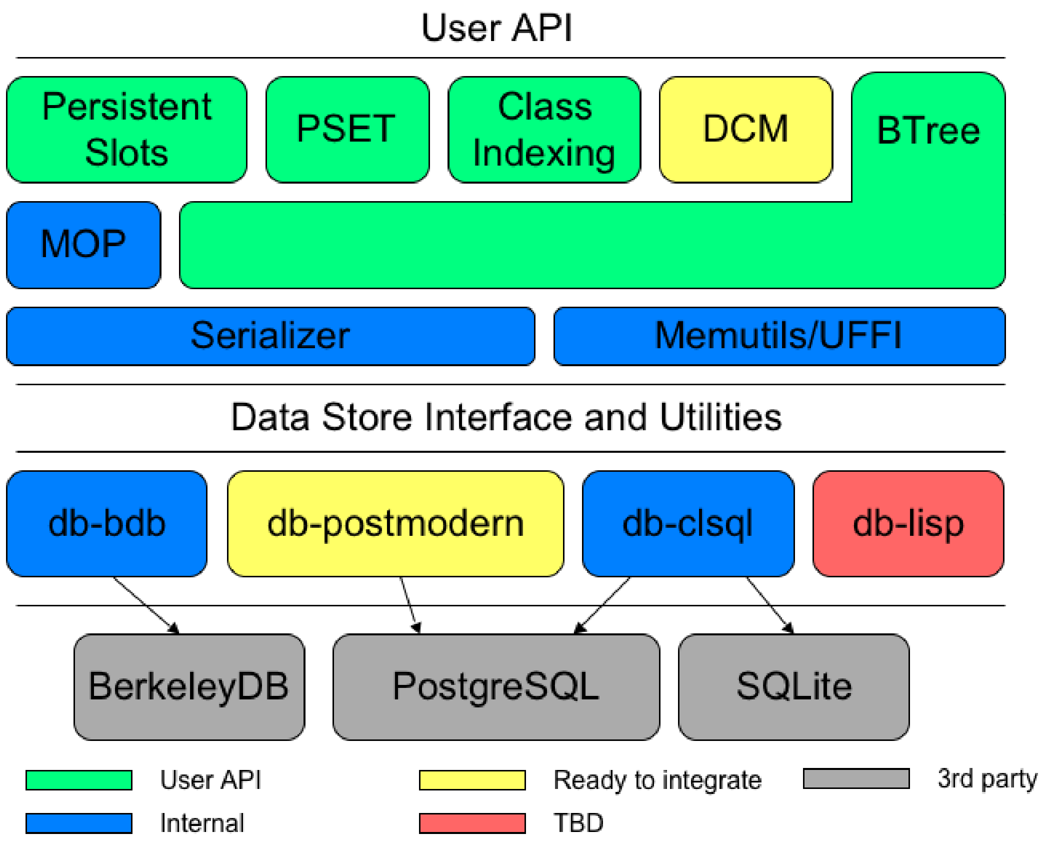
When signals are asserted by the body or a message handler, they are also wrapped in restart handlers called:

- **retry:** Try to execute the component again
- **retry n times:** Using retry you can retry the body or message again. Usually this works best at the REPL when you can test or repair the error and then keep the procesess running
- **replace:** Interactively or automatically enter an expression to replace the current data item with one of your choosing
- **ignore:** Drop the message or data input as if it never arrived
- **terminate:** Terminate execution of the current component

7 Elephant Architecture

Elephant's early architecture was tightly coupled to the Berkeley DB API. Over time we've moved towards a more modular architecture to support easy upgrading, repository migration, shared functionality between data stores and general hygene.

The architecture has been carefully modularized:



To get a feeling for what is happening inside elephant, it is probably best to walk through the various major protocols to see how these components participate in implementing them.

- Initialization of a store controller
- Creating a persistent object
- Operations on persistent slots
- Operations on persistent collections
- Implementing `with-transaction`

7.1 Initializing a store controller

When the main elephant `open-store` function is called with a specification, it calls get-controller which first checks to see if a controller already exists for that spec.

If there is no controller, it calls `build-controller` to construct one. If the data store code base is not present, `load-data-store` is called to ensure that any asdf dependencies are satisfied. The associations for asdf dependencies are statically configured in `*elephant-data-stores*` for each data store type supported by elephant.

While being loaded, the data store is responsible for calling `register-data-store-con-init` to register a data store initialization function for its spec type (i.e. :BDB or :CLSQL). For example, from bdb-controller.lisp:

```
(eval-when (:compile-toplevel :load-toplevel)
  (register-data-store-con-init :bdb 'bdb-test-and-construct))
```

This mapping between spec types and initialization functions is accessed by `lookup-data-store-con-init` from within `build-controller`. The function returned by `lookup-data-store-con-init` is passed the full specification and returns a `store-controller` subclass instance for the specified data store.

The new controller is stored in the `*dbconnection-spec*` hash table, associating the object with its specification. Finally Elephant calls open-controller to actually establish a connection to or create the files of the data store.

Finally, if the default store controller `*store-controller*` is nil, it will be initialized with the new store controller, otherwise the original value is left in `*store-controller*` until that store controller is closed using `close-store`.

The data store implementor has access to various utilities to aid initialization.

- `get-user-configuration-parameter` - Access symbol tags in my-config.sexp to access data store specific user configuration. You can also add special variables to variables.lisp and add a tag-variable pair to `*user-configurable-parameters*` in variables.lisp to automatically initialize it when the store controller is opened.
- `get-con` behavior when store is closed or lost
- `database-version` a store controller implements this in order to tell Elephant what serializer to use. Currently, version 0.6.0 databases use serializer1 and all later database use serializer version 2. This is to ensure that a given version of the Elephant code can open databases from prior versions in order to properly upgrade to the new code base.
- Symbol conversions. To aid in opening legacy databases, a symbol conversion facility is provided in controller.lisp to be applied to any symbols extracted from the legacy data store. (if, for instance, the type name of subclasses changed, such as sleepycat-btree becoming bdb-btree)

At this point, all operations referencing the store controller should be able to proceed.

At the end of a session,

7.2 Persistent Object Creation

The only thing that a data store has to do to support new object creation, other than implement the slot protocol, is implement the method `next-oid` to return the next unique object id for the persistent object being created.

Existing objects are created during deserialization of object references. The serializer subsystem is built-into the core of elephant and can be used by data stores. The serializer is abstracted so that multiple serializers can be co-resident and the data store can choose the appropriate one. The abstraction boundary between the serializer, the data store, and the core Elephant system is not perfect, so be aware and refer to existing data store implementations if in doubt.

A serializer takes as arguments the store-controller, lisp object and a `buffer-stream` from the memory utility library and returns the buffer-stream with the binary serialized object. The deserializer reverses this process. For all lisp objects except persistent classes, this means reallocating the storage space for the object and recreating all its contents. Deserializing a standard object results in a new standard object of the same class with the same slot values.

Persistent classes are dealt with specially. When a persistent object is serialized, it's oid and class are stored in the `buffer-stream`. On deserialization it uses the oid to check in the store-controller's cache for an existing placeholder object. If the cache misses, then it creates a new placeholder object using the class and oid as described in See Section 4.3 [Persistent Classes and Objects], page 37. The store controller contains a cache instance that is automatically initialized by the core Elephant object protocol.

Currently the serializer is selected by the core Elephant code based on the store controller's database version. See the reference section for details on implementing the store-controller database version method. It is a relatively small change to have the data store choose its own serializer, however we will have to tighten up and document the contracts between the Elephant core code, serializer and data store.

7.3 Persistent Slot Protocol

The core protocol that the data store needs to support is the slot access protocol. During object initialization, these functions are called to initialize the slots of the object. The four functions are:

- `persistent-slot-reader`
- `persistent-slot-writer`
- `persistent-slot-boundp`
- `persistent-slot-makunbound`

More details can be found in the data store api reference section. In short, these functions specialize on the specific `store-controller` of the data store and take instances, values and slotnames as appropriate.

Typically the oid will be extracted from the instance and be used to update a table or record where the oid and slotname identifies the value. A slot is typically unbound when no value exists (as opposed to nil).

7.4 Persistent Collection Protocols

The BTree protocol is the most extensive interface that data stores must implement. Data store implementations are required to subclass the abstract classes `btree`, `indexed-btree`, and `index` and implement their complete APIs. Each class type is constructed by Elephant using a `store-controller` that builds them. These methods are `build-btree`, `build-indexed-btree` and `build-index`.

The `get-value` interface is similar to the persistent slot reader and writer, but instead of using oid and slotname to set values, it uses the btree oid and a key value as a unique identifier for a value.

The BTree protocol almost requires an actual BTree implementation to be at all efficient. Keys and values need to be accessible via the cursor API, which means they need to be walked linearly in the sort order of the keys (described in Section 4.6 [Persistent BTrees], page 44).

An indexed BTree automatically maintains a hash table of the indices defined on it so that users can access them by mapping or lookup-by-name. The data store also has access to this interface.

A BTree index must also maintain a connection to its parent BTree so that an index value can be used as a primary tree key to retrieve the primary BTree value as part of the `cursor-pnext` and `cursor-pprev` family of methods.

The contract of `remove-kv` is that the storage in the data store is actually freed for reuse.

Persistent set implemenation is optional. A default BTree based implementation is provided by default

7.5 Implementing Transactions

One of the most important pieces of functionality remaining to discuss is implementing transactions. In existing data stores, transactions are merely extensions of the underlying start, commit and abort methods of the 3rd party library or server being used. The Elephant user interfaces to these functions in two ways: a call to `execute-transaction` or explicit calls to `controller-start-transaction`, `controller-commit-transaction` and `controller-abort-transaction`.

7.5.1 Implementing Execute Transaction

The macros `with-transaction` and `ensure-transaction` wrap access to the data store's `execute-transaction`. This function has a rich contract. It accepts as arguments the store controller, a closure that executes the transaction body and a set of keywords. Keywords required to be supported by the method (or ignored without loss of semantics) are `:parent` and `:retries`.

The semantics of `with-transaction` are that a new transaction will always be requested of the data store. If a transaction exists, `ensure-transaction` will merely call the transaction closure. If not it will function as a call to `with-transaction`.

`execute-transaction` is that it must ensure that the transaction closure is executed within a dynamic context that insures the ACID properties of any database operations (`pset`,`btree` or persistent slot operations). If there is a non-local exit during this execution, the transaction should be aborted. If it returns normally, the transaction is committed. The integer in the `:retries` argument dictates how many times `execute-transaction` should retry the transaction before failing.

Elephant provides some bookkeeping to the data store to help with nested transactions by using the `*current-transaction*` dynamic variable. In the dynamic context of the transaction closure, another call to `execute-transaction` may occur with the transaction argument defaulting to the value of `*current-transaction*`. The data store has to decide how to handle these cases. To support this, the first call to execute transaction can create a dynamic binding for `*current-transaction*` using the `make-transaction-record` call. This creates a transaction object that records the store controller that started the transaction and any data store-specific transaction data.

The current policy is that the body of a transaction is executed with the `*store-controller*` variable bound to the store-controller object creating the transaction. This is important for default arguments and generally helps more than it hurts, so is an implementation requirement placed on `execute-transaction`.

If two nested calls to `with-transaction` are made successively in a dynamic context, the data store can create true nested transactions. The first transaction is passed to the `:parent` argument of the second. The second can choose to just continue the current transaction (the CLSQL data store policy) or to nest the transaction (the BDB data store policy).

7.5.2 Interleaving Multiple Store Transactions

Finally, some provision is made for the case where two store controllers have concurrently active transactions in the same thread. This feature was created to allow for migration, where a read from one database happens in one transaction, and while active has to writes to another data store with a valid transaction.

The trick is that `with-transaction` checks to see if the current transaction object is the same as the `store-controller` object passed to the `:store-controller` argument. If not, a fresh transaction is started.

Currently no provision is made for more than two levels of multi-store nesting as we do not implement a full transaction stack (to avoid walking the stack on each call to handle this rare case). If a third transaction is started by the store controller that started the first transaction, it will have no access to the parent transaction which may be a significant source of problems for the underlying database.

8 Data Store API Reference

This reference includes functions that need to be overridden, classes inherited from or other action taken to implement support for a new data store. Included are the exported elephant functions that need methods defined on them as well as the data-store-only functions exported in data-store-api.lisp. Some functions here are utilities from the main elephant package that support store implementations, but are not required. Migration, class indices and query interfaces are implemented on top of the store API and require no special support by implementors.

Because the number of data store implementors is small, this is a minimal documentation set intended to serve as an initial guide and a reference. However, it is anticipated that some interaction will be needed with the developers to properly harden a datastore for release.

The sections each contain a short guide and a list of functions relevant to them.

8.1 Registration

Elephant looks at the first element of the specification list to determine which data store module to use. The master table for this information is `*elephant-data-stores*` in elephant/controller.lisp. This will need to be augmented for every data store with the specification keyword tag to be used (such as `:BDB` or `:CLSQL`) and the required asdf dependencies.

In addition, the data store source should use an eval-when statement to call the following function:

`elephant-data-store:register-data-store-con-init` *name* *controller-init-fn* [Function]

Data stores must call this function during the loading/compilation process to register their initialization function for the tag name in *elephant-data-stores*. The initialization function returns a fresh instance of the data stores store-controller subclass

If the data store requires any special user-specified configuration, augment the key types in config.sexp with what you need and use the following function to access.

`elephant-data-store:get-user-configuration-parameter` *name* [Function]

This function pulls a value from the key-value pairs stored in my-config.sexp so data stores can have their own pairs for appropriate customization after loading.

8.2 Store Controllers

Subclass store-controller and implement store and close controller which are called by open-store and close-store respectively.

`elephant:store-controller` [Class]

Class precedence list: `store-controller, standard-object, t`

Slots:

- `spec` — initargs: `:spec`

  Data store initialization functions are expected to initialize :spec on the call to make-instance

- `root`

  This is an instance of the data store persistent btree. It should have an `oid` that is fixed in the code and does not change between sessions. Usually it this is something like 0, 1 or -1
- `class-root`

  This is another root for class indexing that is also a data store specific persistent btree instance with a unique `oid` that persists between sessions.
- `instance-cache`

  This is an instance cache and part of the metaclass protocol. Data stores should not override the default behavior.
- `instance-cache-lock`

  Protection for updates to the cache from multiple threads. Do not override.
- `serializer-version`

  Governs the default behavior regarding which serializer version the current elephant core is using. Data stores can override by creating a method on initialize-serializer.
- `serialize`

  Accessed by elephant::serialize to get the entry point to the default serializer or to a data store specific serializer
- `deserialize`

  Contains the entry point for the specific serializer to be called by elephant::deserialize

Superclass for the data store controller, the main interface to any book-keeping, references to `db` handles, the instance cache, btree table creation, counters, locks, the roots (for garbage collection,) et cetera. Behavior is shared between the superclass and subclasses. See slot documentation for details.

`elephant-data-store:open-controller` *sc* **&key** *recover recover-fatal thread* **&allow-other-keys** [Generic Function]

Opens the underlying environment and all the necessary database tables. Different data stores may use different keys so all methods should &allow-other-keys. There are three standard keywords: :recover, :recover-fatal and :thread. Recover means that recovery should be checked for or performed on startup. Recover fatal means a full rebuild from log files is requested. Thread merely indicates to the data store that it is a threaded application and any steps that need to be taken (for example transaction implementation) are taken. :thread is usually true.

`elephant-data-store:close-controller` *sc* [Generic Function]

Close the db handles and environment. Should be in a state where lisp could be shut down without causing an inconsistent state in the db. Also, the object could be used by open-controller to reopen the database

`before` *elephant-data-store:close-controller* (*sc store-controller*) [Method]

Ensure the classes don't have stale references to closed stores!

`after` *elephant-data-store:close-controller* (*sc store-controller*) [Method]

Delete connection spec so store-controller operations on cached controller information fail

`elephant-data-store:connection-is-indeed-open` *controller* [Generic Function]

Validate the controller and the db that it is connected to

For upgrading and opening legacy databases it is important that a store be able to indicate which version of elephant was used to create it. This governs the chosen serializer, mappings between elephant symbols used in an old vs. new version, etc. Because this is called to initialize the serializer, it must directly implemented by the data store without using the serializer.

`elephant-data-store:database-version` *sc* [Generic Function]

Data stores implement this to store the serializer version. The protocol requires that data stores report their database version. On new database creation, the database is written with the *elephant-code-version* so that is returned by database-version. If a legacy database does not have a version according to the method then it should return nil

`around` *elephant-data-store:database-version sc* [Method]

Default version assumption for unmarked databases is 0.6.0. It is possible to check for 0.5.0 databases, but it is not implemented now due to the low (none?) number of users still on 0.5.0

There are some utilities for serializing simple data without a serializer using the memutil package.

`elephant-data-store:serialize-database-version-key` *bs* [Function]

Given a buffer-stream, encode a key indicating the version using the constant +elephant-version+

`elephant-data-store:serialize-database-version-value` *version bs* [Function]

Serializes a list containing three integers to the buffer stream bs

`elephant-data-store:deserialize-database-version-value` *bs* [Function]

Deserializes the 3 integer list from buffer stream bs

8.3 Slot Access

Persistence is implement with a metaclass and several required base classes.

`elephant:persistent-metaclass` [Class]

Class precedence list: `persistent-metaclass`, `standard-class`, `class`, `specializer`, `metaobject`, `standard-object`, `t`

Metaclass for persistent classes. Use this metaclass to define persistent classes. All slots are persistent by default; use the :transient flag otherwise. Slots can also be indexed for by-value retrieval.

`elephant:persistent` [Class]

Class precedence list: `persistent, standard-object, t`

Slots:

- `%oid` — initargs: `:from-oid`

  All persistent objects have an oid
- `dbconnection-spec-pst` — initargs: `:dbconnection-spec-pst`

  Persistent objects use a spec pointer to identify which store they are connected to

Abstract superclass for all persistent classes (common to both user-defined classes and Elephant-defined objects such as collections.)

`elephant:persistent-object` [Class]

Class precedence list: `persistent-object, persistent, standard-object, t`

Superclass for all user-defined persistent classes. This is automatically inherited if you use the persistent-metaclass metaclass. This allows specialization of functions for user objects that would not be appropriate for Elephant objects such as persistent collections

Persistent objects can be queries for their home store controller so that functions such as map-btree do not need a store-controller argument. (NOTE: Should this function be user visible?)

`elephant-data-store:get-con` *instance* **&optional** *sc* [Generic Function]

This is used to find and validate the connection spec maintained for in-memory persistent objects. Should we re-open the controller from the spec if it's not cached? That might be dangerous so for now we error

All objects require a unique object identifier. During new object creation the data store is asked to produce a unique id.

`elephant-data-store:next-oid` *sc* [Generic Function]

Provides a persistent source of unique id's

These functions are called by the metaclass protocol to implement the appropriate operations on persistent class slots. Unless protected by a transaction, the side effects of these functions should be atomic, persistent and visible to other threads on completion.

`elephant-data-store:persistent-slot-writer` *sc new-value instance name* [Generic Function]

Data store specific slot writer function

`elephant-data-store:persistent-slot-reader` *sc instance name* [Generic Function]

Data store specific slot reader function

`elephant-data-store:persistent-slot-boundp` *sc instance name* [Generic Function]

Data store specific slot bound test function `elephant-data-store:persistent-slot-makunbound` *sc instance name* [Generic Function]

Data store specific slot makunbound handler

8.4 Collections

To support collections, the data store must subclass the following classes.

`elephant:persistent-collection` [Class]

Class precedence list: `persistent-collection, persistent, standard-object, t`

Abstract superclass of all collection types.

`elephant:btree` [Class]

Class precedence list: `btree, persistent-collection, persistent, standard-object, t`

A hash-table like interface to a BTree, which stores things in a semi-ordered fashion.

`elephant:btree-index` [Class]

Class precedence list: `btree-index, btree, persistent-collection, persistent-object, persistent, standard-object, t`

Secondary index to an indexed-btree.

`elephant:indexed-btree` [Class]

Class precedence list: `indexed-btree, btree, persistent-collection, persistent, standard-object, t`

A BTree which supports secondary indices.

To create the data store-appropriate type of btree, the data store implements this method (and possibly related methods) aginst their store-controller.

`elephant-data-store:build-btree` *sc* [Generic Function]

Construct a btree of the appropriate type corresponding to this store-controller.

Most of the user-visible operations over BTrees must be implemented. Class indexing functions such as `map-class` and `get-instances-by-value` and related functions are all implemented using map-btree and map-index.

- [Generic-Function elephant:get-value], page 61 (and `(setf get-value)`)
- [Generic-Function elephant:existsp], page 61
- [Generic-Function elephant:remove-kv], page 61
- [Generic-Function elephant:get-index], page 62
- [Generic-Function elephant:remove-index], page 62
- [Generic-Function elephant:map-btree], page 61
- [Generic-Function elephant:map-index], page 62

Mapping over the indices of a btree is important to derived facilities such as class indexing and the query subsystem.

`elephant:map-indices` *fn bt* [Generic Function]

Calls a two input function with the name and btree-index object of all secondary indices in the btree

8.5 Cursors

Data stores must subclass these cursor classes and implement all the methods described in Section 8.5 [DSR Cursors], page 96 except [Macro elephant:with-btree-cursor], page 62.

`elephant:cursor` [Class]

Class precedence list: `cursor, standard-object, t`

Slots:

- `initialized-p` — initargs: `:initialized-p`

  Predicate indicating whether the btree in question is initialized or not. Initialized means that the cursor has a legitimate position, not that any initialization action has been taken. The implementors of this abstract class should make sure that happens under the sheets... Cursors are initialized when you invoke an operation that sets them to something (such as cursor-first), and are uninitialized if you move them in such a way that they no longer have a legimtimate value.

A cursor for traversing (primary) BTrees.

`elephant:secondary-cursor` [Class]

Class precedence list: `secondary-cursor, cursor, standard-object, t`

Cursor for traversing secondary indices.

8.6 Transactions

These functions must be implemented or stubbed by all data stores.

`elephant-data-store:execute-transaction` *store-controller txn-fn* **&rest** *rest* **&key &allow-other-keys** [Generic Function]

This is an interface to the backend's transaction function. The body should be executed in a dynamic environment that protects against non-local exist, provides `acid` properties for `db` operations within the body and properly binds any relevant parameters.

`elephant-data-store:controller-start-transaction` *store-controller* **&key &allow-other-keys** [Generic Function]

Start an elephant transaction

`elephant-data-store:controller-commit-transaction` *store-controller transaction* **&key &allow-other-keys** [Generic Function]

Commit an elephant transaction

`elephant-data-store:controller-abort-transaction` *store-controller transaction* **&key &allow-other-keys** [Generic Function]

Abort an elephant transaction

These are supporting functions and variables for implementing transactions.

`elephant-data-store:*current-transaction*` [Variable]

The transaction which is currently in effect.

`elephant-data-store:make-transaction-record` *sc txn* [Function]
: Backends must use this to assign values to *current-transaction* binding

`elephant-data-store:transaction-store` *txnrec* [Function]
: Get the store that owns the transaction from a transaction record

`elephant-data-store:transaction-object` *txnrec* [Function]
: Get the backend-specific transaction object

;; Designer considerations: ;; - with-transaction passes *current-transaction* or the user parameter to execute-transaction ;; in the parent keyword argument. Backends allowing nested transactions can treat the transaction ;; as a parent, otherwise they can reuse the current transaction by ignoring it (inheriting the dynamic ;; value of *current-transaction*) or rebinding the dynamic context (whatever makes coding easier). ;; - ensure-transaction uses *current-transaction* to determine if there is a current transaction ;; in progress (not null). If so, it jumps to the body directly. Otherwise it executes the body in a ;; new transaction by calling ... ;; - execute-transaction contract: ;; - Backends must dynamically bind *current-transaction* to a meaningful identifier for the ;; transaction in progress and execute the provided closure in that context ;; - All non-local exists result in an abort; only regular return values result in a commit ;; - If a transaction is aborted due to a deadlock or read conflict, execute-transaction should ;; automatically retry with an appropriate default amount ;; - execute-transaction can take any number of backend-defined keywords, although designers should ;; make sure there are no semantic conflicts if there is a name overlap with existing backends ;; - A typical design approach is to make sure that the most primitive interfaces to the backend ;; database look at *current-transaction* to determine whether a transaction is active. Users code can also ;; access this parameter to check whether a transaction is active.

8.7 Multithreading Considerations

This expands slightly on the multithreading discussion in Section 4.10 [Multi-threaded Applications], page 48.

Elephant provides a set of generic locking functions in `src/utils/locks.lisp` to help protect any shared structures. There are standard locking functions (`ele-with-lock`) and then a special locking interface called `ele-with-fast-lock` which on some lisps provides a faster locking option than the standard OS locks of the basic interface. (i.e. under Allegro this uses `without-interrupts` because Allegro still runs in a single OS process on all platforms, this is not true of SBCL).

See the sections on Transaction handling, particularly the dynamic behavior of `*current-transaction*`. Also read up on the store controller section in the User Guide to better understand the role of `*store-controller*`. At this time there are no other global variables to worry about.

8.8 Handling Serialization

Data stores must initialize [Class elephant:store-controller], page 91 with internal serializer functions. Packages `elephant-serializer1` and `elephant-serializer2` contains serialize and deserialize methods on buffer-streams as defined in `elephant-memutil`. The elephant functions `serialize` and `deserialize` dispatch on the appropriate slot values of the store-controller.

```
NOTE: This should perhaps become entirely the job of the data store to
decide how to serialize values and for a specific version, what
serializer to use.  The elphant main package can define serializers
for use by different data stores.
```

`elephant-data-store:serialize` *frob bs sc* [Function]

Generic interface to serialization that dispatches based on the current Elephant version

`elephant-data-store:deserialize` *bs sc* [Function]

Generic interface to serialization that dispatches based on the current Elephant version

These utility functions are useful if a data store does not have the ability to store variable length binary data. They are based on the `cl-base64` library.

`elephant-data-store:serialize-to-base64-string` *x sc* [Function]

Encode object using the store controller's serializer format, but encoded in a base64

`elephant-data-store:deserialize-from-base64-string` *x sc* [Function]

Decode a base64-string using the store controller's deserialize method

9 Copyright and License

9.1 Elephant Licensing

Elephant is a persistent metaprotocol and object-oriented database for Common Lisp. Detailed information and distributions can be found at `http://www.common-lisp.net/project/elephant`.

The program is released under the following license:

9.2 Elephant Manual Copyright and Licensing

9.3 3rd Party Libraries

Elephant depends on 3rd party lisp libraries. See their respective distributions for detailed copyright and licensing information. The following is a brief summary.

- **uffi**: By Kevin Rosenberg, no significant restrictions
- **cl-base64**: By Kevin Rosenberg, no significant restrictions
- **rt**: By Richard Waters, MIT License

9.4 Data Store Licensing Considerations

The Berkeley DB data store is based on the Berkeley DB C library, now owned by Oracle, but available as GPL'ed software. It is important to understand that applications using Berkeley DB must also be GPL'ed unless you negotiate a commercial license from Oracle. In most interpretations of the license, this includes a requirement to make code available for the entirety of any publicly visible website that is based on Berkeley DB. See

`http://www.oracle.com/technology/software/products/berkeley-db/htdocs/bdboslicense.html`.

The CL-SQL backend, depending on which SQL engine you use, may not carry this restriction and you can easily migrate data between the two. Since the Berkeley DB store is 4-5x faster than SQL, it may make sense to develop under BDB and transition to SQL after you've tuned the performance of the application. Licenses for various SQL engines can be found at:

- SQLite: Public Domain, see the SQLite license page
- Postgresql: BSD License, see the Postgresql license page
- MySQL: Dual licensing (similar to BDB), see the MySQL license page

Appendix A Concept Index

Appendix B Object Index

Appendix C Function / Macro Index

Appendix D Variable Index

Colophon

This manual is maintained in Texinfo, and automatically translated into other forms (e.g. HTML or pdf). If you're *reading* this manual in one of these non-Texinfo translated forms, that's fine, but if you want to *modify* this manual, you are strongly advised to seek out a Texinfo version and modify that instead of modifying a translated version. Even better might be to seek out *the* Texinfo version (maintained at the time of this writing as part of the Elephant project at `http://www.common-lisp.net/project/elephant/`) and submit a patch.