Persistent Store for Source and Derived Databases

Jonathan Bachrach @ August 23, 1995 4:53 pm

1 Overview

This document proposes a persistent store (PS) design (and its simulation) to be used as the persistence substrate for source and derived databases in Harlequin's release zero Dylan development environment. The goal of this design is to be as simple as possible while at the same time addressing the requirements of these databases.

It is beyond the scope of this design to design a comprehensive database management system (DBMS), but it is hoped that this design will provide the substrate upon which a general purpose database can be built in the future. In particular, this design does not address DBMS issues such as access control, queries, transactions, schema evolution, file systems, etc.

2 Source and Derived Database Overview

Full description of these databases in their white papers.

2.1 Source Database Overview

The source database is a database that models all program source, including actual source code, requirements, specifications, and design rationale, user documentation, test suites and examples, bug tracking information, and

project management entities, etc. It supports a fine granularity based on the notion of a section roughly corresponding to one definition in a programming language. It supports various kinds of data including text, graphics, audio, and so forth. It supports hypertext links between sections of arbitrary types. It provides source version control functionality such as branching, versioning, conflict resolution, change histories, and snap shotting. It provides basic support for system configuration management in the form of support for grouping of files into projects and snapshotting particular configurations of source files and projects.

2.2 Derived Database Overview

3 Source and Derived Database Requirements

The source and derived databases are restricted in their scope and thus simplify the problem. This section motivates the design by analyzing the requirements of these databases.

3.1 Immediate Requirements

The following requirements are meant to be ones which need to be addressed in order that workable databases can be constructed.

3.1.1 Easy Integration

It should be as easy as possible to write programs that process persistent objects. First of all, the persistent store should support all of the power of Dylan. Furthermore, it is highly desirable to simplify the conversion of existing applications to use persistent objects. In particular, it is important to save the programmer from writing code that translates between disk resident representations and their representations during execution. Furthermore, it is important to save the programmer from specially treating persistent accesses and specially typing persistent objects.

3.1.2 High Performance

Because the source and derived databases are so fundamental to the performance of the compiler and environment in general, it is crucial that the persistent store executes swiftly. In particular, these databases require complex manipulations and queries that process large amounts of richly interconnected object-oriented data. Furthermore, derived databases will be sufficiently large to preclude the use of batch oriented persistent stores such as DOSS. The initial start up time is deemed too large to be acceptable. This argues for a demand loaded system with random access.

3.1.3 Minimal Memory Consumption

Source and derived databases contain a large amount of data and yet it is a requirement of the development environment to run in a reasonable memory footprint. In order to minimize the memory consumption, memory consumption should not exceed the size of the information that is actually consulted.It is assumed, though, that the in memory size of a pool will suffice for disk storage and that further.

Furthermore, it is desirable that extra code and data is not required to support persistent objects. In particular, it would be wise to avoid duplicate classes and functions to represent and manipulate persistent versions of transient objects. For example, it is undesirable to clone off a special version of the collection classes and their associated functions in order to support persistent collections. It is undesirable from a code / data bloat perspective and from a reusability perspective. Finally, it is also important to minimize the amount of bookkeeping information required to support the persistent store.

3.1.4 Incremental

Source and deriveds databases must support incremental redefinition of objects within a particular database and objects bound to exported persistent variables.

3.1.5 Single User

The release zero Dylan system will not support multiple users nor team programming and as such does not require multiple user support in the persistent store.

3.1.6 No Transactions

Transaction support is not required at this time because, a derived database can be reconstructed by recompiling a library, and a source database's reliability will be handled elsewhere for starters.

3.1.7 No Versioning

Versioning will be implemented in the source database itself.

3.2 Limited Cross Pool Pointers

It will be sufficient to require all cross pool accessable data to be explicitly exported / imported before usage. Furthermore, cross pool pointer consistency will be managed elsewhere as this is integral to the derived database's namespace component.

3.2.1 Architecture Dependent

Release Zero DylanWorks will operate only on a single platform, namely Win32 Intel, and thus, there is no need for heterogeneity.

3.2.2 Usual GC

There is no reason to believe that a special GC mechanism (above and beyond the one provided by the memory manager) will be required. There might be a need for special treatment of cross pool pointers.

3.2.3 No huge Objects

Large objects will no require special treatment and in general the size of a given persistent pool will not exceed the size of VM.

3.2.4 No Persistent Meta Objects

Classes and functions can be usefully stored in programs and can be symbolically referenced from persistent stores. No schema evolution is necessary as derived databases can be regenerated from sources. This might pose a problem for source databases.

3.3 Future Enhancements

The following enhancement would increase the functionality and performance of source and derived databases but are not absolutely required.

3.3.1 Simple Transactions

The source database will probably require reliable updates and it would be advantages to recover failures in midst of compilation without a full recompilation of the active libraries.

3.3.2 Shadowing / Revert

It would be desirable for the derived database to allow the ability to back out of the updates caused by a library compilation.

3.3.3 Clustering and Anticlustering

It is conceivable that the database will be too slow without some mechanism of placing objects with respect to pages. Performance data will be needed to make this decide whether this is important for the derived database.

3.3.4 Managed Cache Size

It is desirable to page out less often accessed pages to improve memory performance. Performance data will be needed to make this decide whether this is important for the derived database.

3.3.5 Reinitialization Protocol

It could be that transient slots in persistent objects will need to be reinitialized upon restoring from disk.

3.3.6 Batch Schema Evolution

Source databases need some upgrade mechanism and as such, might get by with a rudimentary batch-style schema upgrade mechanism.

4 General Design

The basic design is similar to object store or texas pstore, where a persistent store is a memory mapped disk file which is intelligently demand paged. Upon saving and restoring, the pointers within these pages are adjusted to reflect the appropriate base address. This process is called *page conversion*. The reader is referred to ostore or pstore documentation for more details.

The persistent store has been tested using a coarse grain simulation of a virtual memory system. Because standard Dylan doesn't have the ability to override pointer operations, a new meta object system and run-time were defined in order to force transparent references to objects to go through the simulated virtual memory system. This separate object system is just the sort of thing the design is purported to avoid. The upshot of this is that the user must define special classes (or use special builtin ones) in order to create persistent instances. The convention is that names of the special meta objects (including class and function names) are prefixed by '@'.

5 Getting Started

The following is example of the simulation:

```
require: ps;

restart-pools();
fluid-bind (*pool* = open-persistent-pool(#"dylan"))
  let todo = apply(@string, "dig");
  add-export!(*pool*, #"todo", todo);
end fluid-bind;
close-persistent-pool(#"dylan");
fluid-bind (*pool* = reopen-persistent-pool(#"dylan"))
  let todo = export-value(*pool*, #"todo");
  as-string(todo) = "dig"
end fluid-bind;
close-persistent-pool(#"dylan");
```

Note that in the simulation the user must use distinguished classes and functions and must use explicit coercions between objects of these classes and their counterparts outside the simulation.

6 Concepts and API

This section describes persistent store concepts and its highest level API.

6.1 Memory Manager

It is assumed that there is a *memory manager* which administers a collection of memory *pools*. Each pool has its own policy which governs pool behavior such as allocation and garbage collection.

<memory-manager> *Class*

shutdown-pools (mm :: <memory-manager>) *Method*

Clean up memory manager including associated VM and all active pools.

restart-pools (mm :: <memory-manager>) *Method*

shutdown-pools and then reinitialize memory-manager.

6.1.1 Allocation

Memory allocation occurs from a particular pool determined by the current setting of the following fluid variable:

***pool* :: <pool>** *Variable*

Controls the allocation of memory through **primitive-allocate.**

```
fluid-bind(*pool* = *jumbo-pool*)
  @pair(1, 2);
end fluid-bind;

fluid-bind(*pool* = *shrimp-pool*)
  make-big-goofy-dude-like-zombie-object(); // dynamic scoping
end fluid-bind;
```

```
fluid-bind(*pool* = *bobs-pool*)
  deep-copy(big-moby-object); // copy between pools
end fluid-bind;
```

The use of a fluid variable to control memory allocation is deemed controversial. It is beyond the scope of this paper to prescribe a more comprehensive solution.

6.2 Pools

Pools are areas from which objects can be created and managed. For the purposes of this document, there are two types of pools, *persistent* and *transient* pools, where persistent means that the pool is at least shadowed in a permanent storage medium, and transient means that the contents of the pool live only while the application is running. A persistent store is implemented in terms of a persistent pool, and in that respect, the two terms will be used synonomously in this paper.

`<pool>` *Class*

`<transient-pool> (<pool>)` *Class*

`<persistent-pool> (<pool>)` *Class*

`dump (pool :: <pool>, #key cached? = #f)` *Method*

Dump a *pool* as a sequence of pages to **standard-output**. If *pool* is persistent, permit either dumping directly from disk or from cache using *cached?* keyword parameter. Pointers to values corresponding to exported variables are dumped symbolically.

6.2.1 Importing / Exporting

Objects in a pool can be accessed through exports. These serve as the roots to the data within a given pool.

`add-export!` *Method*

`(pool :: <pool>, name :: <symbol>, value)`

Add an export named *name* to *pool* with the given *value*.

remove-export! (pool :: <pool>, name :: <symbol>) *Method*

Remove exported variable named *name* from *pool*.

export-value (pool :: <pool>, name :: <symbol>) *Method*

Read the named exported value.

export-value-setter (value, pool :: <pool>, name :: <symbol>) *Method*

Change the named exported value.

Persistent cross pool pointers have very limited support and restricted functionality. They are meant to be as simple as possible and require manual intervention. In particular, the exports of a given pool must be defined before being used in another pool. Furthermore, all imports of a given pool must be defined before being used in that pool. Finally, user code is responsible for all consistency checking of the imports and exports.

One repercussion of this is that exported values are not garbage collected. The pool must ensure that if exported values are moved that all references to them are patched to ensure up to date values for exported objects in other pools. In this respect, exported values are constants in the typical Dylan sense.

It is useful to consider the potential types of cross pool pointers and their storage:

transient -> transient = transient

transient -> persistent = transient

persistent -> transient = persistent if meta-object

persistent -> persistent = persistent if properly exported / imported

It will be necessary to add support for persistent pointers to transient meta objects such as classes and functions. This will allow classes and functions to live in programs instead of being stored persistently. This also finesses the booting problem for the dylan library.

All non-persistent cross pool pointers are saved as unbound and the user is responsible for their reinitialization.

add-import! *Method*

(pool :: <pool>, pool-name :: <symbol>, name :: <symbol>)

Add an import to *pool* from pool named *pool-name* and variable named *name*.

`remove-import!` *Method*

`(pool :: <pool>, pool-name :: <symbol>, name :: <symbol>)`

Remove an import to *pool* from pool named *pool-name* and variable named *name*.

`add-all-imports! (pool :: <pool>, pool-name :: <symbol>)` *Method*

Add all imports exported from pool named *pool-name*.

`remove-all-imports! (pool :: <pool>, pool-name :: <symbol>)` *Method*

Remove all imports exported from pool named *pool-name*.

6.2.2 Saving / Restoring

This section describes manual functions for saving persistent pools to disk. The pool manager will take care of incrementally restoring pages from disk.

`flush-dirty-pages (pool :: <persistent-pool>) => ()` *Method*

Write out pages that have changed since last save.

`dump-persistent-pool (pool :: <persistent-pool>) => ()` *Method*

Save persistent *pool* to disk.

6.2.3 Opening / Closing

In order for the memory manager to administer persistent pools, it maintains a named pool registry which contains a mapping from names to named pools. Currently, pool names and their associated pathnames are oversimplified. In the future, resource locators will be used in place of symbols for identifying persistent pools.

`open-persistent-pool` *Method*

`(name :: <symbol>) => (pool :: <persistent-pool>)`

Create a new persistent-pool of given *name* where its filename is the *name*'s string. The *pool* is then added to the named pool registry.

reopen-persistent-pool *Method*

(name :: <symbol>) => (pool :: <persistent-pool>)

Open an existing persistent-pool of given *name* where its filename is the *name*'s string. The *pool* is then added to the named pool registry. The entries in the import table are resolved at this time.

maybe-reopen-persistent-pool *Method*

(name :: <symbol>) => (pool :: <persistent-pool>)

Either find existing *pool* in the named pool registry or **reopen-persistent-pool**.

abandon (pool :: <pool>) => ()

Clean up *pool* unregistering pages, closing *pool's* stream, and removing *pool* from the named pool registry.

shutdown (pool :: <pool>) => ()

abandon *pool* and **dump-persistent-pool** to disk.

7 Implementation

This section describes the implementation details. The simulation is meant to be a proof of concept and at times is overly simplistic. The simulation assumes that all elements in memory are word width. For example, there is no support for objects of size greater than the size of a page. This was deemed to be beyond the scope of the simulation.

7.1 Run-Time

primitive-allocate-from-pool *Function*

(pool :: <pool>, size::<integer>)

Allocates *size* words from *pool*.

`primitive-allocate (size::<integer>)` *Function*

Allocates *size* words from **pool**.

`primitive-element (pointer, offset :: <integer>)` *Function*

Reference a word *offset* words from the base *pointer*.

`primitive-element-setter (value, pointer, offset :: <integer>)` *Function*

Set a word *offset* words from the base *pointer* to *new-value*.

7.2 Memory Manager

`memory-manager-vm (mm :: <memory-manager>) => (_ :: <vm>)` *Method*

The memory manager maintains pool registries in order administer the active pools.

7.2.1 Pool Registry

A memory manager maintains a pool registry (in addition to a named pool registry) which contains a collection of all active pools.

`register-pool! (mm :: <memory-manager>, pool :: <pool>) => ()` *Method*

Adds *pool* to the pool registry.

`unregister-pool! (mm :: <memory-manager>, pool :: <pool>) => ()` *Method*

Removes *pool* from the pool registry.

7.2.2 Named Pool Registry

`find-named-pool` *Method*

```
      (mm :: <memory-manager>, name :: <symbol>)
   => (pool :: false-or(<pool>))
```

Looks up a pool by *name* in the named pool registry.

register-named-pool *Method*

```
(mm :: <memory-manager>, pool :: <pool>) => ()
```

Registers *pool* in the named pool registry.

unregister-named-pool *Method*

```
(mm :: <memory-manager>, pool :: <pool>) => ()
```

Removes *pool* from the named pool registry.

7.3 Pool

<page> *Class*

$default-pool-page-size :: <integer> *Variable*

page-size (pool :: <pool>) => (_ :: <integer>) *Method*

Returns the number of words per page.

make-page *Method*

```
(pool :: <pool>, pool-page-number :: <integer>) => (_ :: <page>)
```

vm-page-number *Method*

```
(pool :: <pool>, pool-page-number :: <integer>)
=> (_ :: <integer>)
```

Maps from pool to VM page numbers.

pool-page-number *Method*

```
(pool :: <pool>, vm-page-number :: <integer>)
=> (_ :: <integer>)
```

Maps from VM to pool page numbers.

7.3.1 Exporting / Importing

A pool's exports are represented as an association list mapping variable names to their values, where each such mapping is called a binding. The variable names are stored as persistent strings.

A pool's imports are represented as an association list mapping pool names to association lists of variable names to their imported values. Thus there is one second level association for each imported pool.

Cross pool pointers are saved to disk symbolically by storing a reference to the appropriate binding in the import table and resolved by way of that import table binding upon restoration from disk.

The following functions are useful for managing a pool's imports.

`import-value` *Method*

`(pool :: <pool>, pool-name :: <symbol>, name :: <symbol>)`

Read imported value for *pool* from export pool named *pool-name* and exported variable named *name*.

`import-value-setter` *Method*

`(value, pool :: <pool>, pool-name :: <symbol>, name :: <symbol>)`

Change imported value for *pool* from export pool named *pool-name* and exported variable named *name*.

7.4 Persistent-Pool

In order to support a simple page conversion mechanism, data is stored on disk using a tagged representation using the following table:

`00` Pointer

`01` Integer

`10` Character

`11` Cross Pool Pointer

Pointers are stored on disk as disk offsets. Cross Pool Pointers are stored as pointers to the import table binding corresponding to the imported variable.

7.5 VM

This interface describes a simple minded interface to a VM. It is imagined that this will be strengthened in order to support real VM back-ends.

<vm> *Class*

shutdown (vm :: <vm>) => () *Method*

Clean up vm deallocating VM pages and performing other bookkeeping

number-pages (vm :: <vm>) => (_ :: <integer>) *Method*

Number of pages allocated for *vm*.

$default-page-size :: <integer> *Variable*

page-size (vm :: <vm>) => (_ :: <integer>) *Method*

Returns the number of words per page.

<vm-page> *Class*

7.5.1 Page Allocation

reserve-page (vm :: <vm>, page-number :: <integer>) => () *Method*

Reserve VM address space for page.

allocate-page *Method*

(vm :: <vm>, page-number :: <integer>) => (page :: <vm-page>)

reserve-page and backing store for page.

7.5.2 Page Mapping

find-page *Method*

```
(vm :: <vm>, page-number :: <integer>)
=> (page :: false-or(page :: <vm-page>))
```

Looks up page by *page-number* or returns false if not found.

as-pool-page (page :: <vm-page>) => (page :: false-or(<page>)) *Method*

Maps from VM to pool pages.

7.5.3 Triggers

`write-trigger? (page :: <vm-page>) => (_ :: <boolean>)` *Method*

Is page set to call **`handle-write-trigger`** on write to page?

`write-trigger?-setter` *Method*

```
      (new-value :: <boolean>, page :: <vm-page>)
   => (trigger? :: <boolean>)
```

Change write-trigger setting for page.

`handle-write-trigger (page :: <vm-page>) => ()` *Method*

Write-trigger handler which is called before write to page.

`read-trigger? (page :: <vm-page>) => (_ :: <boolean>)` *Method*

Is page set to call **`handle-read-trigger`** on read to page?

`read-trigger?-setter` *Method*

```
      (new-value :: <boolean>, page :: <vm-page>)
   => (trigger? :: <boolean>)
```

Change read-trigger setting for page.

`handle-read-trigger (page :: <vm-page>) => ()`

Read-trigger handler which is called before read to page.

7.5.4 Pointers

A pointer points to a word within a vm pages.

`<pointer>` *Class*

Under the hood smart pointer.

`make-pointer` *Method*

```
      (vm :: <vm>, page :: <vm-page>, offset :: <integer>)
   => (pointer :: <pointer>)
```

Create a pointer for given *page* and *offset*.

\= (p1 :: <pointer>, p2 :: <pointer>) => (_ :: <boolean>) *Method*

Comparison for smart pointers. This really should work for **==**, but does not because of limitations of the emulator.

7.6 MOP

The mop supports single inheritance and single argument dispatch. The following functions permit the creation of the mop in a given pool:

@boot-mop (pool :: <pool>) *Method*

Constructs MOP in *pool*.

@export-mop (pool :: <pool>) *Method*

Constructs exports for MOP in *pool*.

@reboot-mop (pool :: <pool>) *Method*

Copies exports for MOP in *pool* to Dylan variables of same name.

The following classes and their functions are supported. Not much detail is given but they are modeled very closely off their Dylan counterparts.

7.6.1 Core MOP

<@object>	*Class*
<@integer>	*Class*
<@character>	*Class*
<@boolean>	*Class*
$@true	*Variable*
$@false	*Variable*
<@class>	*Class*
@subtype? (type-1, type-2)	*Function*
@instance? (object, type)	*Function*
<@function>	*Class*
<@method>	*Class*
@make-method (code, specializers)	*Function*
<@generic-function>	*Class*
@choose-applicable-method (generic-function, argument)	*Function*
@add-method (generic-function, method)	*Function*

7.6.2 Collections

<@vector>	*Class*
@make-vector (size :: <integer>)	*Function*

@vector (#rest elements)	*Function*
@element (vector, index :: <integer>)	*Function*
@element-setter (new-value, vector, index :: <integer>)	*Function*
@add (vector, element)	*Function*
<@list>	*Class*
@head (list)	*Function*
@head-setter (new-value, list)	*Function*
@tail (list)	*Function*
@tail-setter (new-value, list)	*Function*
<@pair>	*Class*
<@empty-list>	*Class*
<@string>	*Class*
@string (#rest characters)	*Function*

7.6.3 Import / Export Support

<@binding>	*Class*
@key (binding)	*Function*
@key-setter (new-value, binding)	*Function*
@value (binding)	*Function*
@value-setter (new-value, binding)	*Function*