4 Benefits of Integrating a Language With a Database

By Dr. Jans Aasman:

One of the hallmarks of a truly modern programming language is the coupling of a language—for creating applications, devising algorithms and solving business problems—with an underlying database. When there is practically no separation between a database and its programming language, developers immensely increase their productivity, maximize the effectiveness of the code they write and provide unparalleled speed and flexibility in supporting business needs.

Predicating a programming language on a database delivers these advantages by enabling developers to treat data as if it is in-memory, liberating them from the burden of manipulating how data is represented in the underlying database. This one simple advantage exponentially increases the flexibility for addressing business problems and, when used with advanced logic languages, enables developers to write applications in a fraction of the time and with less effort than it would otherwise require.

Specific benefits of tightly coupling a database with its programming language include eliminating the time-honored mismatch between data structures and database representations, smart caching with transactional technologies for committing or rolling back database changes and automatically changing object definitions.

These benefits remove the database as an otherwise complicating factor in supporting business use of data by freeing developers to flexibly innovate solutions—and speedily implement them.

In-Memory Manipulations

Developers focus on manipulating data structures to solve business problems. Traditionally, this concern was hamstrung by the onus of translating those structures into how data are represented in the database, a time-consuming necessity detracting from programmers' attention to business problems. By fully integrating an object-oriented database or graph database with its programming environment, users can manipulate data objects in memory and they'll automatically persist to storage. Developers can solely focus on working with data structures for business objectives without spending just as much, if not more, effort wrangling data's representation in the database.

Read the full article at DevOps.

Fixed Indices Speed up Slot Access in Allegro CL

Instances of CLOS classes include a vector of slot values associated with the instance. A new feature in Allegro CL allows these slot values to be accessed more efficiently by specifying at **defclass** time the index of the slot values vector a particular slot will occupy.

1.0 Background

Object-oriented programming and specifically CLOS (the Common

Lisp Object System) provide a powerful programming tool but it comes at a cost: the more complex the hierarchy of classes and the provision of methods, the longer it takes for the system to determine exactly what should be done in a specific method call. When a method is called, the class of every required argument must be determined and all methods associated with the class pattern of the arguments must be found.

One common action is determining the value of a slot of a class instance. Slots are specified when a CLOS class is defined, with some slots being inherited from superclasses and some being added by the new class definition. Regardless of whether slots are inherited or new, when an instance of the class is created, a vector of slot values is associated with the instance.

Thus to obtain an instance’s slot value, the system must only determine the index of the particular slot in the vector of values. At its least optimized, finding the index involves accessing information in the class definition, which can be costly. This access is what takes time when getting a slot value: once the index is known, the value is obtained by a simple vector access.

2.0 Fixed indices

It is possible that the structure of the slot value vector (that is, which index corresponds to which slot) might change if a class is redefined. Therefore it is unsafe to assume that, say, if a slot value is at index 2 in one call it will be at that index in a later call. Slot accesses cannot, using standard Common Lisp techniques be fully optimized.

Allegro CL has added a new feature which improves upon this: fixed index values. Now when a class is defined, each slot can be assigned an index value and the system guarantees that the slot value will always be at that index of the slot value vector. This means that a slot value access can be reduced to

```
(svref [slot-value-vector] [slot-index])
```

which, with proper declarations and optimization levels, can be reduced to a few machine instructions. This can provide significant speed-ups, as the following example shows.

```
(defclass foo ()
((a :initarg :a excl:fixed-index 2 :accessor foo-a)
(b :initarg :b excl:fixed-index 3 :accessor foo-b)
(c :initarg :c :accessor foo-c)))
(defvar *foo-inst* (make-instance ‘foo :a 1 :b 2 :c 3))

;; *VEC* is the slot values vector for *FOO-INST*.
;; The value of the ‘A slot is index 2.
(defvar *vec* (excl:std-instance-slots *foo-inst*))

;; The slot value vector of *foo-inst* has four elements for three
;; values:
;;  0 – value if slot c
;;  1 – unused
;;  2 – value of slot a
;;  3 – value of slot b

;; Let’s time some accesses. We define test functions in a file:

;; file sv.cl
(in-package :user)

(defun p1 () (dotimes (i 10000000) (foo-a  *foo-inst*)))
(defun p2 () (dotimes (i 10000000) (slot-value *foo-inst* ‘a)))
(defun p3 () (dotimes (i 10000000) (svref *vec* 2)))
;; sv.cl end

;; We compile with speed 3:
Compiler optimize setting is
```

```
(declaim (optimize (safety 1) (space 1) (speed 3) (debug 0)
(compilation-speed 0)))
cl-user(39): :cl sv.cl
;;; Compiling file sv.cl
;;; Writing fasl file sv.fasl
;;; Fasl write complete
; Fast loading sv.fasl

;; And we run timings:
cl-user(41): (time (p1))  ;; USING THE FOO-A ACCESSOR:
; cpu time (non-gc) 0.127957 sec user, 0.000000 sec system
; cpu time (gc)     0.000000 sec user, 0.000000 sec system
; cpu time (total)  0.127957 sec user, 0.000000 sec system
; real time  0.127954 sec (100.0%)
; space allocation:
;  0 cons cells, 0 other bytes, 0 static bytes
; Page Faults: major: 0 (gc: 0), minor: 88 (gc: 0)
nil
cl-user(42): (time (p2)) ;; USING SLOT-VALUE:
; cpu time (non-gc) 0.274489 sec user, 0.000000 sec system
; cpu time (gc)     0.000000 sec user, 0.000000 sec system
; cpu time (total)  0.274489 sec user, 0.000000 sec system
; real time  0.274485 sec (100.0%)
; space allocation:
;  0 cons cells, 0 other bytes, 0 static bytes
; Page Faults: major: 0 (gc: 0), minor: 0 (gc: 0)
nil
cl-user(43): (time (p3)) ;; USING SVREF ON THE SLOT VALUE
VECTOR:
; cpu time (non-gc) 0.005561 sec user, 0.000994 sec system
; cpu time (gc)     0.000000 sec user, 0.000000 sec system
; cpu time (total)  0.005561 sec user, 0.000994 sec system
; real time  0.006555 sec (100.0%)
; space allocation:
;  0 cons cells, 0 other bytes, 0 static bytes
; Page Faults: major: 0 (gc: 0), minor: 0 (gc: 0)
```

Accesses using **svref** is thus 50 times faster than ordinary access.

You will note we specified the value vector index of a slot using the slot option excl:fixed-index in the **defclass** form. This is an Allegro CL extension and if used in code which will also be run in other Common Lisp implementation must be conditionalized for Allegro CL only.

2.1 Before, after, and around methods

In the example above, we reduced a slot access to a **svref** call into the slot value vector. This will produce the maximum speedup but users must keep in mind that before, around, and after methods defined on **slot-value** or on a slot accessor function (**foo-a** and **foo-b** in our example as slot **c** did not have a fixed index) will not be run in that case because the actual methods are not run, just the call to **svref**. Users who intend to write before, around, or after methods on slot accessors can get the desired behavior and much of the speedup with one additional level of indirection:

```
(defmethod generic-foo-a ((obj foo)) (foo-a obj))
```

Before, after, and around methods can then be written for **generic-foo-a** and will run as expected.

3.0 Documentation

Allegro CL documentation can be found at https://franz.com/support/documentation/. Fixed-index slots are described in the section Defclass optimizations: fixed-index slots and defclass embellishers in the implementation.htm document. Also described in that section are **defclass embellishers**, which we will discuss in a later blog item.