Jay Almarode

Tuning multi-user Smalltalk

AN IMPORTANT ACTIVITY before delivering any application is tuning the code to meet performance requirements. In single-user Smalltalks on the client machine, this activity typically involves using profiling tools to identify the methods where most time is spent. Once these methods are identified, several options are available, such as implementing the methods as primitives in C code, caching calculated values that are used repeatedly, or perhaps most important, producing a better design. This tuning activity might also involve analyzing the memory usage of the application, reducing the memory footprint of the application while it is running, minimizing the number of temporary objects that are created and then quickly garbage collected, and exercising more explicit control over garbage collection (especially in real-time systems).

These same tuning activities are applicable to multi-user, server Smalltalk as well. In addition, because server Smalltalk must accommodate concurrent transactions by many hundreds of users, and must handle many millions of objects being created and retrieved, there are additional ways in which applications can be tuned. In this column I discuss some of the techniques to tune multi-user, server Smalltalk applications.

A key component in tuning a large-scale, multi-user Smalltalk application is understanding and controlling the placement of objects on disk. Because the number and size of objects may prevent all that are being used in an application from being present in RAM at once, the proximity of objects may impact application performance. Obviously, the fewer disk pages to be accessed during the normal course of application execution the better performance. To tune the placement of objects on disk, server Smalltalk must allow developers to cluster objects that are frequently accessed together. In GemStone Smalltalk, objects are placed on disk based on their access patterns by default. More specifically, objects that are created or modified within the same transaction tend to be placed close together. In many cases, this default placement is sufficient.

However, GemStone Smalltalk does provide additional protocol to allow developers to discover where objects are placed and to move them closer together for more efficient access.

The first step in tuning an application's performance for accessing objects is to understand the reading and writing characteristics of the application while it is running. In GemStone Smalltalk, you can send the messages "pageReads" or "pageWrites" to class System to get the cumulative number of pages that were read or written since the session began (i.e., since you logged into the server). Typically, it is useful to measure the number of pages read immediately before and immediately after an extensive calculation or query to determine if clustering objects together might be of benefit. For example, the following code returns the number of pages that were read to execute the given query.

```
| initialNumberOfReads |
initialNumberOfReads := System pageReads.
SetOfPersons select: [ :person |
  " find each person younger than their spouse "
  person isMarried and: [ person spouse age > person age ].
^ System pageReads - initialNumberOfReads
```

Pages are written to disk for two reasons: first, when internal buffers become full and must make room for new objects to be created; second, when the transaction is committed. Measuring the number of pages written at various times during the life of a transaction can help determine if buffer sizes need to be increased, whereas measuring the number of pages written just before and after a transaction is committed may help determine if more explicit control over clustering may help.

Clustering related objects together solves a specific problem: poor performance because of too much disk activity. One way to check how objects are clustered is to determine which pages the objects are stored on. You can send the message "page" to any object to get an integer identifying the disk page on which the receiver resides. This integer is a logical identifier of the page, not a pointer to a storage location.

Objects are stored persistently in structures called *extents*. An extent is a disk file or raw partition on disk. The repository of all objects can be maintained in multiple extents, possibly distributed among several disk drives on several machines. In GemStone Smalltalk, there is a single object, named SystemRepository, that is an instance of

Using Smalltalk since 1986, Jay Almarode has built CASE tools, interfaces to relational databases, multi-user classes, and query subsystems. He is currently a Senior Software Engineer at GemStone Systems Inc., and can be reached at almarode@gemstone.com.

class Repository. In addition to defining protocol to perform online backups and restores, to dynamically add new extents, and to create replicates of extents for purposes of fault tolerance, class Repository also has methods to provide information about the extent in which a page is located and the file name for a given extent. The next example shows how one can determine the file name where an object is actually stored on disk.

```
| extendId |
"MyObject is the object whose location we are interested in."
extendId := SystemRepository extentForPage: MyObject
page.
^ SystemRepository fileNames at: extendId.
```

By analyzing the reading and writing behavior of your application for excessive disk activity and determining the number and location of pages where objects reside, performance may be improved by explicitly controlling how objects are clustered together. Conceptually, you can think of objects as being written to disk on a stream of disk pages. When a page is filled, another page is chosen and objects are written to the new page. The stream of pages used for writing is called a 'bucket'. GemStone Smalltalk provides the class ClusterBucket to give programmers control over which stream of pages objects are written. Every object is associated with an instance of ClusterBucket and all objects assigned to the same ClusterBucket will be clustered together. When objects with the same ClusterBucket are written to disk, they are written to contiguous locations on the same page, if they will fit, or contiguous locations on several pages if not.

A ClusterBucket can be associated with a specific extent. Each ClusterBucket has an instance variable extentId that specifies which file the stream of pages will be written. You can find out what extents are available by executing the expression SystemRepository fileSizeReport. This returns a string that describes the extent identifier, file name, file size, and space available for each available extent. An example of how to set the extent for an existing ClusterBucket is the expression aClusterBucket extendId: 3.

You can create a new ClusterBucket by executing the expression ClusterBucket newForExtent: 4. Initially, there are seven existing instances of ClusterBucket maintained in a global array named AllClusterBuckets.

Some of these are available for application developers, whereas others are used to cluster system objects, such as kernel methods or source code strings. When new instances of Cluster Bucket are created, they are added to this global array and a ClusterBucket's position in this array is known as its cluster Id.

This provides a way to reference any ClusterBucket that exists through its clusterId, for example, by performing the expression ClusterBucket bucketWithId: 7.

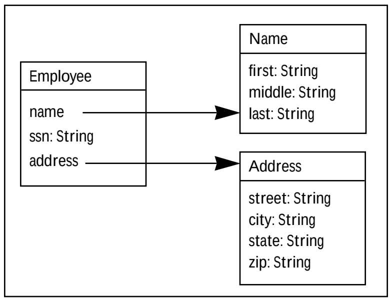

Figure 1. Employee schema.

To specify the ClusterBucket for a particular object, you can send the message "clusterInBucket: aClusterBucket". This will not immediately write the object to disk but indicates that when it is next written, the stream of pages in which it is written will be determined by the given ClusterBucket.

If you want to write the object to disk immediately, you can send the message "moveToDiskInBucket: aCluster Bucket". Sending the "clusterBucket" message to an object will return the ClusterBucket to which the receiver is currently assigned. GemStone Smalltalk provides some convenience methods to help cluster objects. You can send the message "cluster" to any object to assign it to the current default ClusterBucket. You can use this message to build specialized clustering behaviors for your application classes. One such method already provided is clusterDepthFirst, which traverses through the named and indexable instance variables of the receiver, sending the "cluster" message to each object. The cluster method returns a boolean indicating if the receiver has already been clustered during the current transaction. This is used to prevent infinite recursion. There are also convenience methods defined in class Behavior to cluster classes and related objects. The clusterBehavior method clusters a class and its method dictionary. The clusterDescription method clusters the objects that describe the structure of a class, such as its instVarNames array, class variables, instance variable constraints, and class history.

To illustrate how to control object clustering, imagine a set of Employee objects based on the simplified schema illustrated in Figure 1.

Suppose most applications that access an instance of Employee also access the name and ssn as well; so we would like to cluster instances of Employee with their corresponding Name and 'ssn' String objects.

The addresses of employees are accessed less frequently and are typically accessed for all employees at once, so we would like to cluster all Address objects together. The following code shows how we can cluster these objects so that employees and their frequently accessed subcomponents are stored contiguously and employee addresses are grouped together separately.

```
| empCluster addressCluster |

" get the bucket previously created for Employees "
empCluster := ClusterBucket bucketWithId: 8.
" get the bucket previously created for Addresses "
addressCluster := ClusterBucket bucketWithId: 9.

TheSetOfEmployees do: [ :anEmp | | name address |
  anEmp clusterInBucket: empCluster.
  anEmp ssn clusterInBucket: empCluster.

  " cluster the name and its components "
  name := anEmp name.
  name clusterInBucket: empCluster.
```

continued on page 32

GETTING REAL *continued from page 19*

```
name first clusterInBucket: empCluster.
name middle clusterInBucket: empCluster.
name last clusterInBucket: empCluster.

" cluster the address and its components "
address := anEmp address.
address clusterInBucket: addressCluster.
address street clusterInBucket: addressCluster.
address city clusterInBucket: addressCluster.
address state clusterInBucket: addressCluster.
address zip clusterInBucket: addressCluster. ].
```

This column has described how to determine if clustering objects might help application performance and how to cluster objects using ClusterBuckets. My next column will discuss how to measure overall system performance and steps for tuning multi-user Smalltalk for higher transaction throughput.

THE BEST OF COMP.LANG.SMALLTALK

continued from page 23

- Avoid commitment—This is another way of expressing the principle of postponing decisions but one that might strike a chord with younger or unmarried programmers.
- It's not a good example if it doesn't work—This one comes from David Buck (dbuck@magmacom.com), who's fed up with looking at example and test methods that haven't been properly maintained as the code evolved. I can't think of a way to apply this to life but it's good advice anyway.
- Steal everything you can from your parents—A principle for those trying to make effective use of inheritance or moving into their first apartment.
- Cover your a**—Like in a bureaucracy, the most important thing is to make sure that it isn't your fault. Make sure your code won't have a problem even if things are going very wrong elsewhere.

SEQUENTIAL KEY ALLOCATION

continued from page 26

```
        ifTrue: [ keyCache := self nextKeys: self
                  keyCacheSize ].

    key := keyCache first.
    keyCache removeFirst.
    ^key.
```

If you choose to make the array optimization in the nextKeys: method, this method must be changed to insert nil values into the array as each key gets returned rather than using the removeFirst selector.

Dayle Woolston and Chris Kesler have been working with Smalltalk for 4 years building client/server database applications. They can be reached at dayle_woolston@novell.com and chris_kesler@novell.com.