Controlling image size when using GemStone

John Bentley

In most Smalltalk applications, controlling the image size is an important goal. Larger images require more memory for execution, which can degrade performance. When using an object-oriented database (ODB), controlling the image size can be challenging. This challenge comes from the manner in which objects are stored and retrieved in an ODB.

One of the main advantages cited for using an ODB is the ODB's capability to preserve the object relationship web. Each object is stored with its relationship web intact. When an object is retrieved from the ODB, the related objects are also retrieved without having to manually reconstruct the relationship web (see Fig. 1).

In contrast, in a relational database paradigm when an object needs to be retrieved the developer must reconstruct the object being retrieved from a table. Then, the developer must also find and reconstruct the related objects, and finally, re-establish the relationship web. So, when considering the cost of manually rebuilding the object web, it is easy to see that using an object-oriented database can save a great deal of effort.

When retrieving an object from an ODB, related objects are retrieved automatically. The application developer no longer has to know which related objects are needed. This also means that the application developer does not necessarily know how many objects are being retrieved. This could be a problem. For example, if the application's equivalent of the Smalltalk dictionary was retrieved, then all objects in the database would be copied into the image. It would be possible to retrieve most, if not all, of the objects in the database. This can be a real problem as the database usually holds more objects than an individual Smalltalk image can handle.

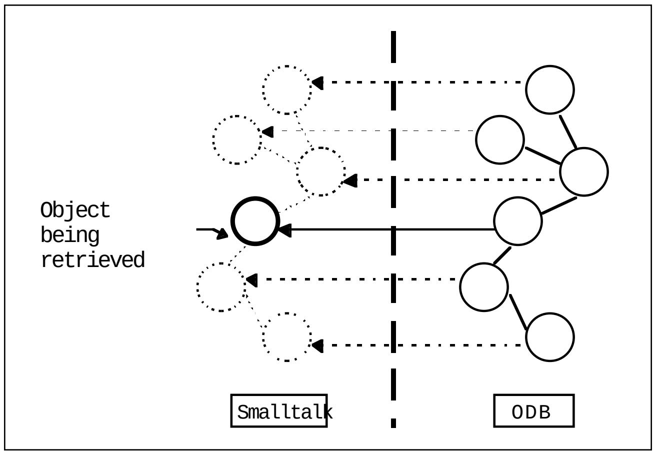

Figure 1. When using an OODb, related objects are automatically retrieved.

To prevent overloading the image with objects it is important to understand the mechanisms provided for the retrieval of related objects. While preventing the extreme case is vital, it is not the only reason. Another reason to control the retrieval of related objects is to keep the image size minimal. It is possible to create an application that does not retrieve any related objects until they are accessed. This set up would keep the image size small but could also make the application's performance bound to database access. It is important to retrieve enough related objects to perform the task at hand. The control mechanisms for retrieval gives the application developer the ability to balance image size versus database access.

Each ODB vendor provides different mechanisms for controlling object retrieval. For this reason, it is easiest to discuss the control mechanisms in the context of a particular object-oriented database implementation. In this case, the database used as an example is GemStone, from GemStone Systems, Inc. The Smalltalk listings are in reference to GemStone 4.0 and VisualWorks 2.0.

All listings refer to a typical employee payroll system. The payroll systems' object model is shown in Figure 2. The application calculates employees' salaries and addresses envelopes for distribution. The application also provides estimates at the company level for weekly and yearly payroll. The purpose of this example is to illustrate concepts and is not meant to be representative of typical ODB applications.

REPLICATION CONTROL

When an object is retrieved from GemStone, the object is "replicated" in the Smalltalk image. The replicated object, referred to as a replicate, is a copy of the GemStone object with a "link" to its database counterpart. The replicate maintains the link to its database counterpart so that updates may occur between the database and the image. In GemStone, this controlling of the retrieval of related objects is referred to as replication control.

The GemStone Smalltalk Interface (GSI) provides two techniques to control replication. One of these techniques is "replication tuning." Replication tuning allows the developer to control how objects and their relations are replicated in the image. The other technique is avoiding replication. In most cases, objects are replicated into a Smalltalk image to perform behavior. GemStone is an active database, allowing behavior to be defined and executed in the database. The behavior is defined using the GemStone Smalltalk dialect which is called Smalltalk DB. Using behavior defined in Smalltalk DB, database objects can respond to messages without being replicated into the client image thus eliminating the need for replication.

REPLICATION TUNING

There are three common approaches to tune replication using GemStone. The first approach is to change the GemStone to Smalltalk-replication level. This defines the level of relationship replication for the image. The second approach is class mapping. This provides a variety of ways to control how GemStone and Smalltalk instance variables relate. The final approach is to tune replication by changing the "no stub level." This controls relation replication levels on database updates.

When an object is initially retrieved from the database, it is represented by a *proxy*. A proxy is merely a Smalltalk object that references a database object. To replicate the object in Smalltalk, the message #asLocalObject is sent to the proxy. The result is a Smalltalk duplicate of the database object that maintains a reference to the database object. Listing 1 shows how the global MyCompany would be retrieved and replicated.

When the proxy replicates itself, it checks to see how many levels of relationships need to be replicated. This is known as the GemStone to Smalltalk replication level. If the level is two, the proxy replicates itself and the objects referenced by the object's instance variables. If the level is three, the objects referenced by the instance variables of the objects referenced by the primary object's instance variables are replicated as well. If the level is zero, all related objects are replicated. Any related objects not replicated are represented by stubs. A stub is a stand-in object that replicates itself when sent a message.

The GemStone to Smalltalk replication level is defined in an instance method in the GSSession class. The GSSession class defines behavior associated with a database session. The replication level is defined in the #defaultGSToSTLevel method. GSSession sets the replication level for the entire application and sets the default replication level for the application.

Defining the replication level in a method provides flexibility. For example, in one application, it might be appropriate for #defaultGSToSTLevel to unconditionally answer three. In another application, it might be appropriate to provide a different level depending on the platform being used. The #defaultGSToSTLevel could be defined as is shown in Listing 2.

In addition, there are special cases where a more spe-

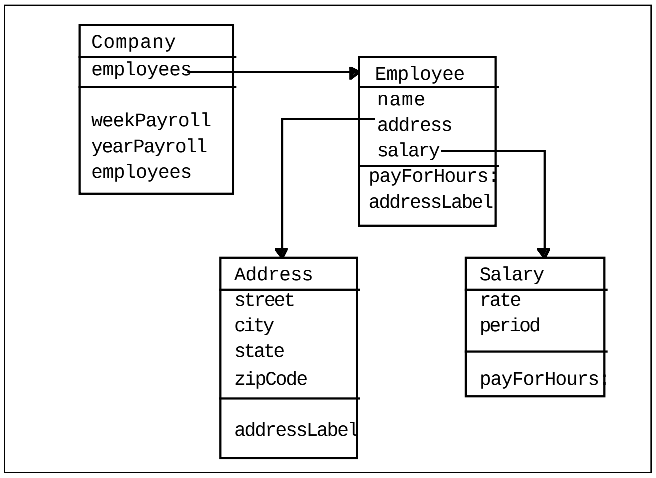

Figure 2. The payroll tracking system's object model.

Listing 1. Retrieving and replicating.

```
| proxy replicate |
"Retrieve the GemStone global MyCompany."
proxy := GSI currentSession at: #MyCompany.
"proxy references the GemStone global MyCompany."

"Replicate MyCompany."
replicate := proxy asLocalObject.
"replicate is a copy of the GemStone global
MyCompany.
It maintains a link to its GemStone counterpart."
```

Listing 2. Platform-dependent replication level.

```
"The following methods allow the default replication
level to be varied on a platform basis."

"Methods for GSSession."

defaultGSToSTLevel
"Answer the appropriate level for the current
platform."
    ^self platformLevels at: CurrentPlatform

platformLevels
"Answer the platform level dictionary. If unset,
initialize."
    (platform is Nil)
        ifTrue: [ (platform := (Dictionary new))
            add: #windows->3;
            add: #macintosh->2;
            add: #unix->5.
        ].
    ^platform
```

cific replication level is required. In these cases, #asLocalObjectToLevel: is used instead of #asLocalObject. The #asLocalObjectToLevel: is sent to a proxy to replicate the object to the level specified as the argument.

In Listing 1, MyCompany was replicated using the default level. If the default replication level was three then MyCompany, its employees collection and all contained employees would be replicated. However, if all of the contained employees are not needed, it might be better to only replicated two levels, as is shown in Listing 3. With the replication level set to 2, each employee object would not be replicated until accessed.

Thus far replication has been tuned at the instance basis. Replication can also be tuned on a class basis. This is done using the second replication tuning mechanism, class mapping. Class mapping allows Smalltalk and GemStone classes to have different structures. In the general case, Smalltalk and GemStone classes will have identical structure. In the example application, the Employee class has three instance variables: name, address, and salary. Both GemStone and Smalltalk would have Employee classes with these same three instance variables defined.

Listing 3. Replicating to a specific level.

```
| proxy replicate |
proxy := GSI currentSession at: #MyCompany.

replicate := proxy asLocalObjectToLevel: 2.
"This bypasses the default level and only replicates
 MyCompany and the collection. Employees are not
 replicated until accessed."
```

Listing 4. Custom mapping employee.

```
In GemStone

Object subclass: Employee
    instVarNames: #( 'name' 'address' 'salary' )
    classVars: #()
    poolDictionaries: #()
    inDictionary: Payroll
    constraints: #[]
    instancesInvariant: false
    isModifiable: false

In Smalltalk:

Object subclass: #Employee
    instanceVariableNames: 'name address salary'
    classVariableNames: '
    poolDictionaries: "
    category: 'Payroll'

stValues: anArray
"anArray contains four proxies for the GemStone
instance variables.
Note that only the first three, name, address, and
salary, are referenced."
    name := anArray at: 1.
    address := anArray at: 2.
    salary := anArray at: 3.
```

In some cases, there may be an instance variable in the GemStone class that should not appear in the Smalltalk class, or vice versa. For example, the GemStone Employee class might have an instance variable called allEmployees which references the collection of all employees in the system. This means when a single instance of Employee is replicated, every employee in the database would also be replicated. To prevent allEmployees from being replicated in the image, the Smalltalk class would not have the allEmployees instance variable. In this case, the Smalltalk class's instance variables would need to be mapped such that allEmployees was not replicated into Smalltalk.

The GSI adds methods to the class Object to handle class mappings. The methods, #stValues and #stValues:, handle the mapping of instance variables between Smalltalk and GemStone. #stValues is used when an object is being stored in GemStone. #stValues stores the instance variables into an array and answers that array. #stValues: is used when replicating an object being retrieved from GemStone. It takes an argument of an Array and copies this array's values into the instance variables. Both of these methods directly map the instance variables to an array, such that the instance variable stored at n is stored in the array at n.

Whenever a class' instance variables are defined differently in Smalltalk and GemStone, the #stValues and #stValues: methods must be redefined in the Smalltalk class to properly map the instance variables. For example, to remove allEmployees from the Smalltalk Employee class, the #stValues: would need to map name, address, and salary from the array, but not acknowledge allEmployees. The Smalltalk code for this is shown in Listing 4.

The final method of replication control addresses the case in which a replicated object is being updated to reflect changes made in its GemStone counterpart. When the database object is changed, the associated Smalltalk replicate object is "stubbed." The replicate object is then transformed into a stub instead of being reloaded. This "stubbing" provides a performance enhancement that allows *laissez-fare* loading of updated objects.

The behavior of a stub object is to replicate itself when accessed. The database object is replicated in its place. When replicated, it also replicates its related objects. Replication is still controlled using the #defaultGStoSTLevel defined in GSSession.

There are certain circumstances in which stubs are inappropriate. In this context, the most interesting circumstance deals with performance. If an object is stubbed, it will be replicated using the default replication level for the application. But, if an object was initially replicated using a custom level, this default behavior is undesirable. In this case, it is better to prevent the object from being stubbed.

Stubbing is controlled by the #noStubLevel method defined by GemStone in the class Object. The "no stub" level defines the levels at which stubbing is not allowed. A "no stub" level of zero, the default, indicates that the receiver can be stubbed. A "no stub" level of one, prevents the receiver from being stubbed. A "no stub level" of two prevents the receiver and the objects referenced by its instance variables from being stubbed, and so on.

In Listing 2, MyCompany was replicated to level two. This was done because the default replication level of three would have also replicated the employees. If MyCompany was stubbed, then on the next access, MyCompany would be replicated using the default replication level of three, replicating all contained employees. To prevent this, the #noStubLevel method is overridden in the Smalltalk Company class to answer 2, as is shown in Listing 5.

AVOIDING REPLICATION

In general, objects are retrieved from a database to interact in the client. In most cases, these objects can only answer messages when replicated in the client image. However, in GemStone, database objects can respond to messages without being replicated in the image. This removes the requirement for objects to be replicated into the client environment when needed to perform a task. These objects can respond according to the behavior defined in the database.

In GemStone, behavior can be defined in two places; the image and the database. This duality is often referred to as the two-space model. The two-space model is powerful but complex. Behavior in the image can be the same or different than the database. If the same method needs to exist in both the image and database, the developer must define the method in the image's dialect as well as in Smalltalk DB (see Fig. 3).

The two-space model is more complex than using a "non-active" database but does provide much greater flexibility to the application developer. For example, a query needs to be performed to find all the employee that make more than $800 a week. In a non-active database, the behavior for calculating pay would need to be in Smalltalk. To perform the query, every employee object would need to be replicated in the image to find those who make more than $800. An active database provides the ability to define salary calculation in the database. Only those employees that made more than $800 would need to be replicated.

Many non-active database provide the ability to perform queries against instance variables in the database. To optimize the query, an instance variable called weeksPay could be added to Employee to store the value.

Listing 5. Changing *Company's* "No Stub" level.

```
"Methods for Company class"

noStubLevel
"This prevents Company and the employees collection
from being stubbed."
    ^2
```

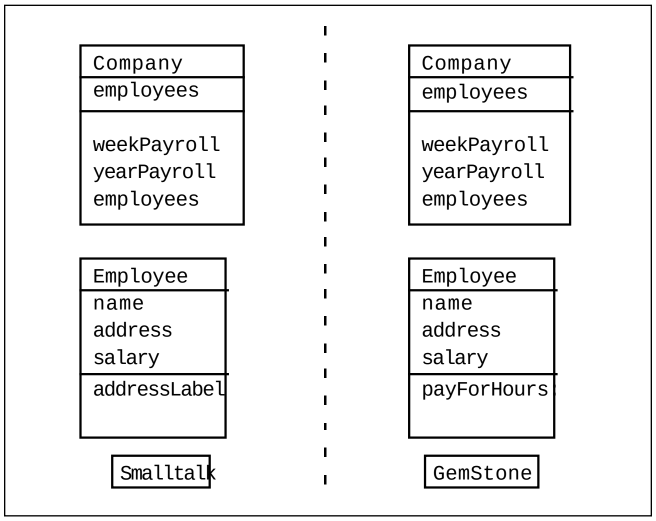

Figure 3. Behavior can be defined in both GemStone and Smalltalk.

However, in this example, if the query changed to use a month's pay, the database schema would need to be changed and the current instance of Employee would need to be migrated. In the active database, only changes to behavior are needed.

GemStone's Smalltalk DB is a Smalltalk dialect that has been specialized for database functionality. Smalltalk DB is similar to the VisualWorks, Visual Smalltalk, and IBM Smalltalk dialects but is not syntactically equivalent to any of these. Smalltalk DB provides the Common Language Data Types as defined in the original Smalltalk language specification. Smalltalk DB does not specify any user interface classes. However, Smalltalk DB does provide extensions to optimize queries and handle large collections.

To access behavior defined in Smalltalk DB, messages can be sent to a database object via its proxy. Proxies

Listing 6. Sending messages via proxies.

```
"Sending the message to a proxy using
remotePerform."
| proxy |
proxy := GSI currentSession at: #MyCompany.
proxy remotePeform: #yearPayroll

"Sending the message to a proxy using the gs prefix."
| proxy |
proxy := GSI currentSession at: #MyCompany.
proxy gsyearPayroll

"Sending the message to a replicate using the gs
prefix.
Convert the result to a replicate."
| replicate |
replicate := (GSI currentSession at: #MyCompany)
asLocalObjec
replicate as GSObject gsyearPayroll asLocalObject
"replicate asGSObject answers a proxy"
```

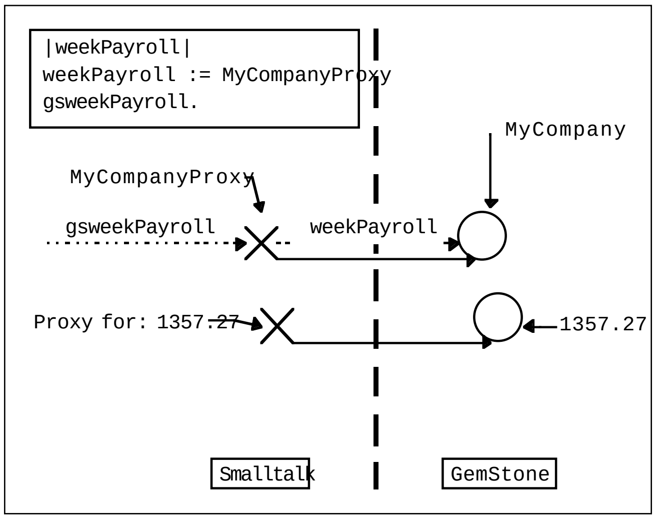

Figure 4. Messages can be sent to database objects via proxies. The resulting object is a proxy.

understand a set of messages similar to the #perform: set of messages defined in the class Object. These messages are #remotePerform:, #remotePerform:withArgs, #remotePerform:with:, #remotePeform:with:with:, etc. In addition to the #remotePerform: series, a proxy also sends messages prefixed with "gs" to its database object. When a message is sent to a database object via a proxy, the resulting object is also a proxy (see Fig. 4).

Database behavior is usually used when the receiver needs to collaborate with several other database objects that are not replicated in the image. Executing this behavior in the database eliminates the overhead of replicating objects when only the results are needed. In the example application, estimated payroll for the year could be calculated in the database. By using database behavior, all of employees for the company do not need to be replicated. Listing 6 contains the Smalltalk code needed to accomplish this.

Using #remotePerform: provides access to database behavior but at the price of dealing with proxies throughout the application. While proxies are Smalltalk objects, they are merely reference points to database objects. The GSI provides a more seamless method for accessing database behavior, namely forwarders.

Forwarders are Smalltalk objects that forward messages to their database object counterparts. When a message is sent to a forwarder, the forwarder sends a #remotePerform:withArgs: to the proxy for the database object. The behavior is executed by the database object and returns the result. By default, the result of a message sent to a forwarder is a replicate. Forwarders allow the application to access database behavior without having to deal with intermediary objects (see Fig. 5).

In general, forwarders are used to represent the principal access points of the system. These access points contain the major collections that are used to find specific instances of objects. First, the search message is forwarded into the database. Then, only the resulting set of objects are replicated, removing a lot of overhead.

Forwarders can be used to virtually eliminate the use of replicates. This can be very useful when dealing with underpowered client machines. By default, messages sent to a forwarder result in a replicate. By appending messages with "fw", the forwarder will answer with forwarders instead of replicates. In an application that only uses forwarders, the #doesNotUnderstand: message on the forwarder class could be modified to return forwarders by default.

In the example application, replication tuning was used to minimize the number of replicated objects related to MyCompany. Payroll messages are being sent to the database objects to avoid replication. Since MyCompany is only a container for employee objects, it is best to reference MyCompany with a forwarder. This prevents all of MyCompany's employees from being replicated. Payroll messages can be sent to MyCompany without regard to proxy conversions. Some sample code is shown is Listing 7.

When using forwarders, it is important to remember there is a trade off. A forwarder saves space in the image by avoiding replication. The messages often have to trav-

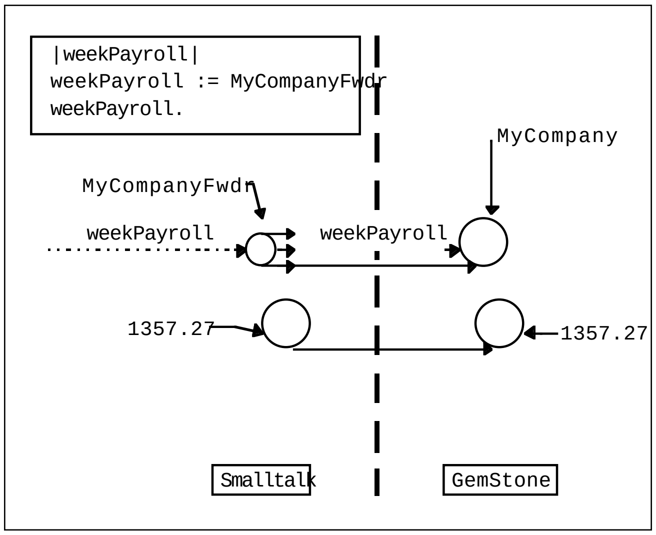

Figure 5. Messages sent to a forwarder are automatically sent to the database object. The resulting object is a replicate.

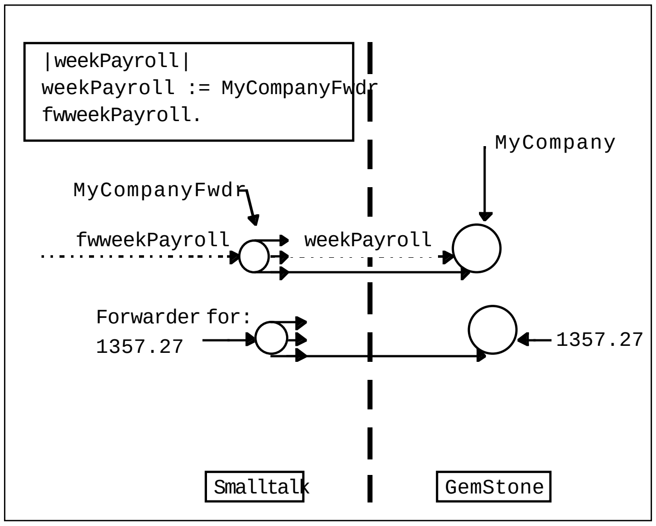

Figure 6. Messages sent to forwarders can be prefixed with a "fw" which causes the resulting object to be a forwarder.

el across the network to be answered. Network latency could cost more than replicating the object. Also, there is the concern of overloading the server with user requests. Essentially, forwarders are not a "silver bullet." As with anything else in software development, it takes trial and error along with educated guesses to determine the optimal solution for each implementation.

CONCLUSIONS

In general, controlling the size of the image is an important goal with respect to system performance. This is especially true when using on object-oriented database such as GemStone. Object-oriented databases provide mechanisms for controlling the flow of objects between database and image. In GemStone, image size control is provided by the ability to tune and avoid replication.

While limiting replication helps control image size, it is also important to consider database access time. Objects can be retrieved without replicating any related objects. This would prevent image growth but each subsequent access would require a database access, negatively impacting the user's performance. Replication control allows you to balance controlling image size while providing object caching for performance.

In summary, when building a Smalltalk application using GemStone, consider carefully the two-space model. When deciding where to execute behavior, look to see where the objects reside. If most of the objects needed to perform the operation are in the database, then define that behavior in the database. This minimizes the need for replication. If the object is needed for heavy interaction in client, replicate the object. This minimizes network latency.

Of course, there are no simple answers. Every application has a different object model and different hardware constraints. Plan to spend time trying out a variety of replication schemes as a part of performance optimization. The "right" solution is the one that works for you.

Listing 7. Forwarder examples.

```
"Get the estimate for the weekly payroll."
| forwarder |
forwarder := (GSI currentSessiona at: #MyCompany) as
Forwarder.
forwarder weeklyPayroll

"Find all employees that live in Raleigh.
Keep the resulting collection as a forwarder."
| forwarder |
forwarder := (GSI currentSessiona at: #MyCompany)
asForwarder.
forwarder employees fwselect: [ :emp| emp address
city = "Raleigh'
```

John Bentley is Member, Technical Staff at JumpStart Systems, Inc., Raleigh, NC. He can be reached via e-mail at jbentley@jmpstart.com or by phone at 919.832.0490.