Back-Pointer Managed Collection

Derek Laufenberg
Director of Product Management
Versant Corporation, dlaufen@versant.com

Abstract

Collection classes provide for very natural and efficient means of navigating a parent – children relationship graphs within an object database. They provide easy access to child objects and can often be exploited to minimize network traffic by reading the entire collection on a single server call. However there are some performance drawbacks when the size of the collection is large and the collection is updated frequently. This collision of size and frequent updates causes a large collection object to get written often. For large collections even the simple list of object IDs (OID) can slow the application. This pattern turns the relationship around and borrows from relational database patterns to eliminate the growth of the parent collection object. It exploits the OID indexing capability to create a synthetic collection with a simple efficient query that performs better when collection updates are frequent or multi-sourced.

Problem Detail

Most OODBs have a natural granularity at the object level and when an object is changed, that entire object is transmitted to the server at transaction commit. Generally, this isn't a problem, as many attributes of the object are likely changed during the transaction. But in the case of a simple collection object, (list, vector, or set) adding a single child to the collection causes a large OID list to be sent back to the server. This amounts to substantial network, CPU, and disk overhead as the collection object is updated in the database.

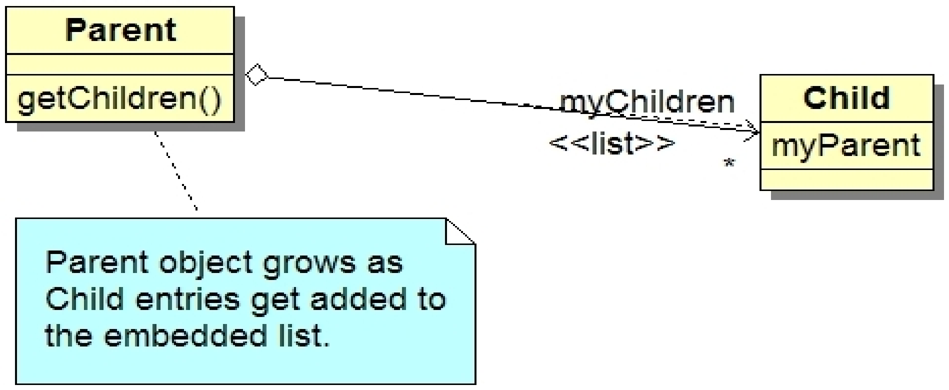
Parent
getChildren()
myChildren
<<list>>
*
Child
myParent
Parent object grows as Child entries get added to the embedded list.

Figure 1 – Typical Parent container with many Children.

Another bottleneck comes into play when multiple transactions need to update the collection. With only a single collection object, the Parent instance, write locks on the

Parent create a performance problem as they become a blocking point for other transactions that need to insert into the collection.

One way to work around this problem is to use a more complex collection container that is built up from multiple container objects. This multi-bucket approach works well when the collections are constant and the child objects don't change collections. I'm sure details for this approach will be covered in other patterns. This pattern takes another approach and uses an indexed query to replace the pre-computed collection.

Pattern Solution

The first part of the solution is to take the collection object out of the parent and place a reference back to the parent in the child class. Often this back reference is already there for navigational purposes anyhow. See the UML Figure 2 below. The child is added to the collection by setting its parent reference to the original parent object.   The next step is to place an index on the child's parent reference. The index is maintained by the server so when a child object is created or changed the collection is handled by the index.

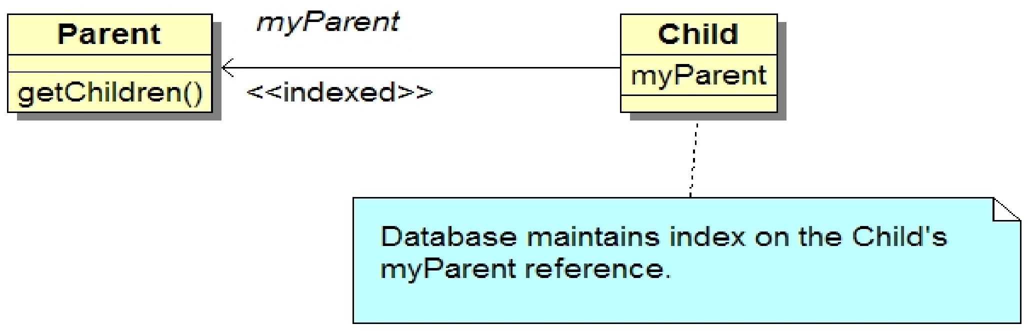

Figure 2 – Backwards reference from Child to Parent.

Now when the parent's GetChildren() method is called, a simple query is done against the database. The query in Versant QL would look like this:

```
select  selfoid from Child  where Child::myParent = $1
```

The query parameter $1 would be bound at execution time to the Parent OID of interest. When the query is executed, all the children which belong to the same parent will be returned. This creates the set that would have been maintained in the original Parent object's collection. Additional optimizations may be possible by setting query parameters that cause the loading of the Child objects as part of the query execution, thus eliminating further network calls. The index allows for fast execution of this query. If the database supports hash table indexing a hash should be used, otherwise a b-tree is sufficient. An index on the OID is critical because without it, this query would be order N to execute.

Example Use Cases

Target Tracking - observations from multiple sources of target's position/speed get updated frequently by the tracking system. The observation list for a given target is appended to and over time the performance would fall off as the collection grows. Using the server side index keeps the insert time near constant and allows multiple sources to commit observations without locking the target (parent) object.

MMOG Player Inventory – Players in MMO games have large collections of items that require inserts frequently or items getting traded between players. Keeping the number of objects involved in a trade or inventory update reduces the transaction complexity and improves system performance. The single back-reference on the item (Child) also helps eliminate duplication bugs where the item is in two inventories.

Benefits

This pattern simplifies the transaction when adding or changing the collection a Child belongs to. Since only a reference to the Parent object is required, the Parent doesn't need to be loaded by the transaction, as only the Parent's OID is needed. Thus, moving a Child from one Parent to another involves only the Child object which greatly improves the transaction efficiency and database concurrency.

Multiple transactions can insert objects into the collection without locking or blocking on the Parent. The database still h

Network and disk IO is saved by eliminating the transmission of the other children in the collection.

Drawbacks

This is very relational database solution so an OO purist may object to this pattern.

A Parent reference for the back-pointer in the Child class is required and may have to be added if it wasn't already needed for navigation.

This pattern moves the responsibility from the Parent to the Child with the help of the database's query system. The object database must support indexing on OID references and the cost to maintain the index is added to the transaction, but for large sets this cost is made up by the IO savings.

When the collection is updated by other transactions the GetChildren query needs to be re-executed to update the working collection. This may require some sort of messaging or event system to trigger.