Large Persistent Collection Common Persistent Pattern

By Matthew Barker, Director of System Engineering, Versant Corporation

Also Known As

Scalable Persistent Collection

Intent

Persistent collection classes are a mainstay of ODBMS. Implementations of persistent arrays, sets, array lists, hash-maps, and other persistent collection classes are present in the various ODBMS products including Versant's Object Database. When these collections classes grow very large, various performance and concurrency problems arise.

The Large Persistent Collection pattern addresses these problems. The intent is to sub-divide a large persistent collection into smaller component collections using a balanced tree structure. The interface to the large collection class is identical to the regular persistent collection class so easy replacement is available as scalability and concurrency demands grow.

Motivation

1. Dramatically improve scalability:

    a. Updates to the LargePersistentCollection will incur only the writing of a small subset of the collection instead of the entire collection.

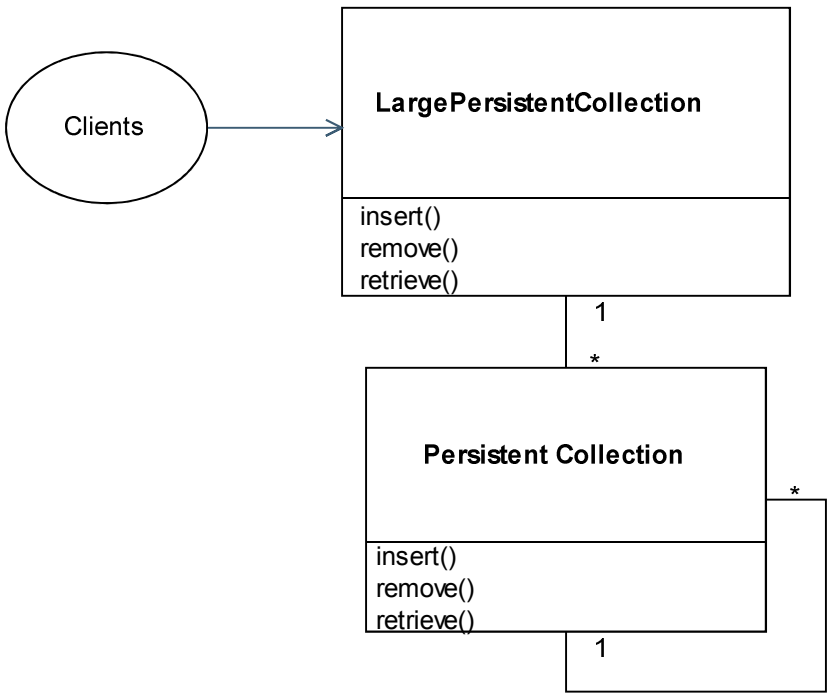
    b. Only the necessary component collection object is fetched from the server as portions of the LargePersistentCollection is accessed by one or more clients.

c. If a regular persistent collection has insert and delete performance of O(n) (such as with a persistent array), then the performance of inserts and deletes will improve to O(log n) with use of the LargePersistentCollection.

2. Improve concurrency:

a. read and writes collisions are minimized; instead of one large collection object, instead the collection is now comprised of many smaller collection objects which are locked separately when that particular portion of the collection class is accessed.

Structure



Discussion

Use of LargePersistentCollection may produce deadlocks where they would not otherwise occur. This may be avoided by adding a method on the parent LargePersistentCollection class that gives exclusive access to the entire LargePersistentCollection thus eliminating the improved concurrency discussed above but still providing the scalability improvements.

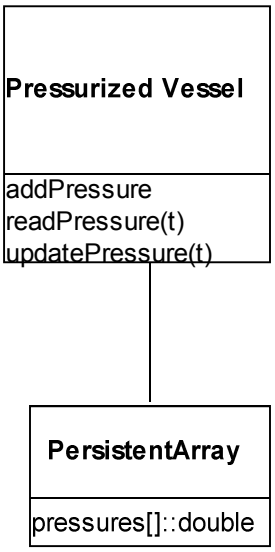
Keeping the tree balanced is another problem that has to be addressed but much research is available in this area.

Example

Consider the use of a persistent array to track the pressure changes of a vessel over time. Further assume that the time sequence could be over very long periods of time with small increments which could result in an array size of millions. Assume the beginning time is saved and the increment of time is fixed at one second so we can always determine the position in the array given a specific time.

A few typical use cases for such a system are:

1. Add a new pressure to end of array.
2. Read the pressure at a given time.
3. Update pressure at a given time (as a correction is deemed necessary).



Without the use of LargePersistentArray, we experience the following problems:

- *addPressure* and *updatePressure* requires we read in the entire PersistentArray and write the entire array back to the server at commit time slowing performance, increasing network traffic, and exhausting memory resources. We also lock out the persistent array from the other two operations until we have completed the *addPressure* operation.

- *readPressure* requires we retrieve the entire array from the server into client memory slowing performance, increasing network traffic, and exhausting memory resources. We also block the *addPressure* and *updatePressure* operations as we acquire a read lock on PersistentArray during the *readPressure* method.

If we replace the use of PersistentArray with LargePersistentArray, we get the following benefits:

- *addPressure* and *updatePressure* only requires we read in the top level object and a few small collection objects of the LargePersistentArray. When the operation completes and the object is written back to the server, we only have to write the one small collection back to the server. We also gain the benefit of not blocking out the other two operations except on rare occasions (when reading or updating the same small collection).

- *readPressure* only requires we read in the top level array and a few small collection objects. We also gain the benefit of not blocking out the other two operations except on rare occasions (when updating the same small collection as we are reading).

Sample Code

Here we show a C++ template implementation of a LargeMap class that supports elemental keys. It utilizes Versant's persistent map implementation "VEIDictionary<>" for the component small collection classes. For brevity sake, the LargeMapIterator class is not included.

```
// Large persistent map, only support elemental keys and values
template <class KEY, class VALUE>
class VEELargeMap : public PObject
{
	TYPEDEF_PClassObject(VEELargeMap)
	friend class VEELargeMapIterator<KEY,VALUE>;

private:
	typedef VEEDictionary<KEY, VALUE>    SmallMap;
	typedef VEIDictionary<KEY, SmallMap> MasterMap;
	KEY  mapSize;
	MasterMap masterMap;

public:
	VEELargeMap() : mapSize(0) { };

	virtual ~VEELargeMap()
	{
		VEIDictionaryIterator<KEY,SmallMap> iter( masterMap );

		while (!(!iter)) {
			(*iter).deleteobj();
			iter++;
		}
	}

	bool isEmpty() const { return size() == 0; }

	VALUE operator[]( KEY key ) const
	{
		// exception will be raised if either key is bad
		KEY masterKey =  key>>13;  // divide by 8192 (i.e. 2^13)
		SmallMap& smallMap = masterMap[masterKey];
		return smallMap[key];
	};

	void add( KEY key, VALUE element )
	{
		dirty();
		KEY masterKey =  key>>13;

		if ( !masterMap.valid_key(masterKey) )
		{
			Link<SmallMap> lNewSmallMap =
				O_NEW_PERSISTENT(SmallMap);
			lNewSmallMap->add( key, element );
			masterMap.add( masterKey, *lNewSmallMap );
```

```
            }
            else  // lookup in master map, add to found small map
            {
                  SmallMap& smallMap = masterMap[masterKey];
                  smallMap.add( key, element );
            }
            mapSize++;
      };

      void remove( KEY key )
      {
            dirty();
            KEY masterKey =  key>>13;

            SmallMap& smallMap = masterMap[masterKey];
            smallMap.remove( key );

            // DON't ALLOW empty small maps in this MasterMap
            if ( smallMap.size() == 0 ) {
                  masterMap.remove(masterKey);
                  smallMap.deleteobj();
            }
            mapSize--;
      };

      KEY size() const
      {
            return mapSize;
      }

};
```

Submitted by:

Matthew Barker
Director of System Engineering

Versant Corporation
255 Shoreline Drive, Suite 450
Redwood City, CA 94065
mbarker@versant.com