ODBMS
Persistent Implementation Pattern
The Versioned Graph

Robert Greene
Vice President
Versant Corporation
rgreene@versant.com

Pattern Name and Classification:

Persistent Versioned Graph Pattern

Intent:

A persistent versioned object graph pattern for time sensitive retrieval of deep object graphs.

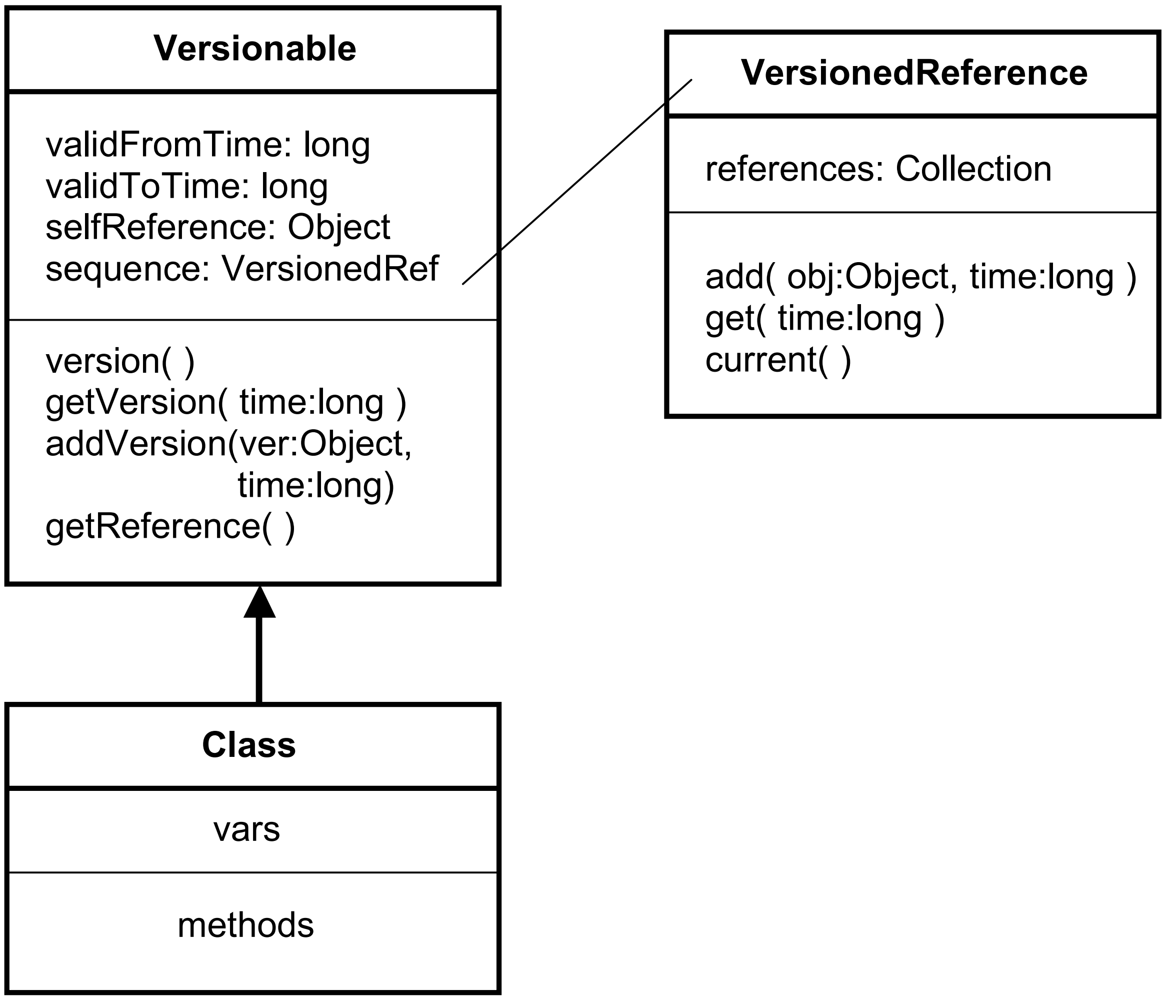
Motivation:

Efficient retrieval of point in time state for changing complex object graphs.

Applicability:

Many industries have problem domains where object state in deep graph structures are changing over time and answers to important questions become relevant to a particular time window. It is not sufficient to simply update an object when it changes, but instead to maintain object state throughout time and when a time sensitive question is asked of the domain, a slice of the object graph relevant to that time period is extracted for analysis. Implementation of this pattern provides for efficient point in time object graph analysis and can also form the foundation for time series related operations.

Structure:



Participants:

Concept Classes: Versionable, VersionedReference.

The concept of Versionable acts as the super class from which all versioned instances would inherit. It provides the basic information regarding the time slot with which a particular instance is valid. It provides a means to access any instance in the system based on a time window via query. It understands how to retrieve a particular version of itself relative to time via model navigation mechanisms, the real value and differentiator which makes this pattern difficult in a relational implementation.

The concept of VersionedReference acts as a reference type for versioned instances of a Versionable object. It is in essence a time sorted collection of references to versioned instances for a given object. When a time sensitive request is made of a Versionable object, this reference type is used to resolve the proper instance.

Collaboration:

Versionable instances are created as are any non-Versionable instance. However, when a Versionable instance is changed, clones are made and added to an associated VersionedReference instance. When message sends are made to a Versionable instance, they can either be directed to a default reference version or they can be directed at a particular time slot via the associated VersionedReference.

Consequences:

The side effects of using this pattern are that normal object references between Versionable classes become time sensitive. This means that normal getter methods need to have an overridden signature that takes in a representation of time and redirects to return the desired instance via the VersionedReference..

There is also one level of indirection between objects when loading a graph for a particular window of time. Therefore, things like network overhead need to be optimized accordingly. In practical situations, since the goal is to load an entire graph for a particular time slot, optimizations will be done to efficiently load levels of referenced objects in aggregation.

Another side effect is that the VersionedReference instance will nearly always need to be loaded with the 1st Versionable instance created. Therefore, multiple disk seeks will be required unless the database has the ability to store both of these types in the same disk block (clustered).

Implementation:

The same basic pattern has been found in two distinct implementations. One in which reference types are handled as a VersionedReference and another in which reference types are Versionable types that manage their own access via a self contained VersionedReference.

In either case, the implementation of the Versionable class itself can be seen to have a similar structure. The difference then resides in whether behavior to access reference objects is delegated to the referenced Versioned instance or handled locally.

One of these possible implementations of the pattern, where the reference type is handled as a VersionedReference, is found in the following text. The implementation and sample code is specific to the Versant ODBMS implementation, but the general approach will be evident for any ODBMS implementation.

An implementation of VersionedReference concept using Versant's Java Interface with trivial methods removed:

```java
package timeHash;

import java.io.*;
import com.versant.fund.Handle;
import com.versant.trans.TransSession;


public class TCollection implements Serializable{

   /*Separates index and object, so that when we're
     evaluating an index, the objects will not be retrieved from
     server until the object is found.  This is because by default,
     OODB's will lazy load and instead we want to do this efficiently.
     So, we are using some low level stuff, isn't strictly necessary.*/

    int elementSize;

    long[] index;  //index of entry
    long[] objs;  //object id of referenced object

   public TCollection()
   {
      objs = new long[8];
      index = new long[8];
      elementSize = -1;
   }


  /**
    * Add a Versionable instance to the collection
    */
   public void add(TransSession session, Object o, long version)
   {

      if( elementSize == (index.length -1) )
          growArrays();

      index[elementSize +1] = version;
      objs[elementSize +1] = session.getOidAsLong(o);
      elementSize++;

   }

   /**
    * Retrieve a time sensitive instance of Versionable
    */
   public Object get(TransSession session, long version)
   {
    if( elementSize == -1 ) return null;

    Handle ret = null;
    if( version <= index[0] )
    {
       ret = session.newHandle( objs[0] );
    }else if (version >= ( index[elementSize ] )){
      ret = session.newHandle( objs[ elementSize ] );
    } else{
      int i=retrieve(version,0,elementSize);
        ret = session.newHandle( objs[i] );
    }
    return session.handleToObject(ret);
}

/**
 * Efficiently retrieve the index of an entry
 * This method recursively divides the problem in half.
 * until the entry is resolved.
 */

private int retrieve(long ver,int start,int end)
{
    int result=0;
    int mid=(start+end)/2;
    int prev=mid-1;
    int next=mid+1;
    long midVer=  index[mid];
    long prevVer= index[prev];
    long nextVer= index[next];

    if((prevVer<=ver) && (ver<=midVer))
    {
      /*if ver is between preVer and midVer*/
      if((ver-prevVer) < (midVer - ver) )
      {
        result=prev;
      }else
      {
        result=mid;
      }
    }else
    if((midVer<=ver) && (ver<=nextVer))
    {
      /*if ver is between midVer and nextVer*/
      if((ver-midVer) < (nextVer - ver) )
      {
        result=mid;
      }else
      {
        result=next;
      }

    }else
    if(ver<midVer)
    {
          result=retrieve(ver,start,mid);
        }else
        {
          result=retrieve(ver,mid+1,end);
        }
        return result;
    }

     private void growArrays(){

        // DO THE  RIGHT THING
     }
};
```

An implementation of the Versionable concept with trivial methods removed:

```
import timeHash.TCollection;
import com.versant.trans.TransSession;

public class Versionable implements Cloneable{

  TCollection sequence;
  long  validFromDate;
  long  validToDate;
  long catalogNumber;
  long serialNumber;
  int reference;
  static long time = 1000*1000*1103760;
  String name;


  public Versionable() {

    this.sequence = new TCollection();
    this.validFromDate = time;
    this.validToDate = Versionable.endOfTime();
    this.reference = -1;
    this.name = "Version";
  }
  public Versionable( long _catNum ){
    this();
    this.catalogNumber = _catNum;

    }


  public Object getVersion( TransSession session, long _dateHash ){
    return  this.sequence.get( session, _dateHash );
  }
  public Handle getVersionHandle(TransSession session,long _dateHash ){
    return this.sequence.getHandle( session, _dateHash );
  }

  public void addVersion( TransSession session, Versionable _version ){

    TCollection t = this.sequence;

    if( t.getElementSize() >= 0 )
    {
      Versionable lastInst = (Versionable)t.lastObject(session);
      lastInst.setValidToDate( time );
    }
      _version.setValidFromDate( time );
      _version.setValidToDate( Versionable.endOfTime() );
      this.sequence.add( session, _version, _version.getSlot() );
      time += 10000;
  }

   public void addVersion( TransSession session, Versionable _version ,
long _time){

    TCollection t = this.sequence;

    if( t.getElementSize() >= 0 )
    {
      Versionable lastInst = (Versionable)t.lastObject(session);
      if( lastInst != null )
      lastInst.setValidToDate( _time );
    }
      _version.setValidFromDate( _time );
      _version.setValidToDate( Versionable.endOfTime() );
      this.sequence.add( session, _version, _version.getSlot() );
    }

  public void setReference( long _time ){
    this.reference = this.sequence.getIndex( _time );
  }

  public static long endOfTime(){
    return 1000*1000*2522880;
  }

  public Object version(TransSession session){
      return version( System.currentTimeMillis() );
  }

  public Object version( TransSession session, long _time ){

    Versionable obj = null;
    try{
        obj = (Versionable)this.clone();
        obj.setValidFromDate( _time );
        obj.setValidToDate( Versionable.endOfTime() );

      // If self referencing, you would also populate the sequence
```

```
        // addVersion( session, obj, _time );

      }catch( java.lang.CloneNotSupportedException  e ){
          e.printStackTrace();
      }
      return obj;
      }
```

Sample Code:

The following code snippets illustrate how instances of Versionable are used at runtime. Note that in this particular case, each of the classes Instrument and TradableInfo inherit from Versionable which means they each in turn have a "sequence" representing a VersionedReference. In this case, Instruments have a sequence of TradableInfo rather than a sequence of its own version history. Again, this is a choice leading to two different implementations of the same pattern.

Example of defining Versionable classes;

```
public class Instrument extends Versionable{

  InstrumentText  text;
  Parameter curveParameter;
  long  underTICatalogNumber;
  static long time = 1000*1000*1103760;



public class TradableInfo extends Versionable{

  MarketData  mData;
  static long time = 1000*1000*1103760;
  long instrumentID;
```

Example of creating a new version

…."time" and "session" are initialized elsewhere…

```
      /**
       * generate 5 versions of TradableInfo and add to Instrument
       */
      public static void populateVersion(Versionable instr, Versionable
      tinfo)
      {
            for(int i = 0; i<5; i++)
            {
                  long key = (time + 1000*i);
                  Versionable o = (Versionable)tinfo.version( );
                  session.makePersistent(o);
                  o.setSerialNumber( session.getOidAsLong( o ) );
                  if(instr != null)
                      instr.addVersion( session, o, key  );
            }
      }
```

……. Commit changes

Example of retrieving a particular version

The simplist example would look something like:

….retrieve an Instrument via query and access related TradableInfo for a specific time.

```
qry = new VQLQuery( session, "select * from Instrument where
validFromDate < $1 AND validToDate >= $2" );
qry.bind( new Long(timeFrom) );
qry.bind( new Long(timeTo));
Iterator itr = qry.execute( );

//use iterator to access an Instrument and load a specific version of
//TradableInfo from the underlying sequence.
While ( itr.hasNext() ){
      TradableInfo t = (TradeableInfo)(itr.next().get( session,
_timeOfInterest ));
 ……. Do something with t
}
```

The following is a more complex implementation, from an Accessor class implementation which aggregates and optimizes versioned graph loading …… please note the complete code with tests will be given freely on request, just send an e-mail.

```
  /**
   * Note the HandleVector is obtained from entries in a TCollection
   * relative to a requested time slot.  So, in this case we have
   * a collection of Instruments previously loaded sensitive to time.
   * Each of those in turn have a sequence of TradableInfo which in
   * turn need to be loaded relative to time.  So, the "handles" are
   * added to a collection and they are read into the cache in a
   * single RPC.
   */

public static HandleVector loadObjectsOfTime( TransSession session,
HandleVector objs, long _timeTo ){

    HandleEnumeration enumer = objs.handles();
    HandleVector hVector = session.newHandleVector();

//Grab the handles for the correct time period
    while( enumer.hasMoreHandles() ){
      Versionable v =
      (Versionable)session.handleToObject(enumer.nextHandle());
      hVector.addHandle( v.getSequence().getHandle( session,_timeTo )
);
    }
//Read all time sensitive objects in single RPC into cache
    hVector.groupReadObjects( session.database(), 0, Constants.NOLOCK
);

    return hVector;


}
```

Known Uses:

This pattern has been used in the cache implementations of several online trading systems for portfolio management and risk trading analytics. The pattern has also been used in the analysis of reaction-reaction modeling for the study of chemical compound interactions in life sciences.   It is also know to have applicability in areas of artificial neural network analysis and other problem domains of a time sensitive nature.