Serializing CA-Visual Objects

by Bill Cross

CA-World 2000
eBusiness Solutions in Internet Time
JP055SN

Introduction

In an application with more than a single data table, a basic problem is the linking of the two or more data tables together. In the DBF file format, the responsibility for maintaining the linkage is clearly the programmer's. This means that a unique serial identification or string must be used to identify each record. This may be the name of the entity being described, such as "Smith, John". But this is typically not a reliably unique identifier. An alternative is to have an ID field that the user would enter manually. However, then you must ensure that the user entry is maintained as unique, and you are thus reliant upon the imagination of the user to come up with unique names. An alternative method is the use of some type of mnemonic where the first letter of a particular field is used followed by a number; this is then checked against the database to ensure that the new key is unique and to increment it if it is not. Another method would be to maintain an index on the ID field and then go to the bottom of the table and generate a new key by incrementing on that last record. Each of these has the problem that another user on the network could be doing the same thing at the same time, and thus getting the same number. Another issue with this method is its need to navigate through the table to which you are trying to append a record. A better way would be for the application to generate a unique ID for each record automatically.

This paper discusses the implementation of a unique key ID generator for use in VO. This generator implementation ensures that each key ID is unique and that its storage is efficient.

Background

Purpose for Serial Numbers

As mentioned above, each record's having a unique serial number is critical for several reasons. Each serial number serves the purpose of ensuring correct identification of a record. This allows a stable linking of database records as opposed to other more suspect methods, such as user entry. This serial number must be unique for the database it is assigned to, it ideally should never be used twice, and it should be efficient in its use of storage space. The last means that while we could use a simple numbering system, we would need to first anticipate the size of our future database in records and ensure that our KeyID field is at least that long. Allowing for the maximum manageable file size for clipper/VO files of 35,000,000 records, we would then need to allow a field size of 8 characters. This would equate to 280MB just for the Key IDs of our records. So any reduction in the field size for the Key ID would be appreciated.

How It Would Be Used

Each table that required a unique Key ID would require an index tag on the KeyID field. This use of an index of unique Key IDs enables quickly retrieving any record specifically from the table. Any item about the record can be displayed to the user as a long string display, but the KeyID can be used to fetch and retrieve the specific record based upon the KeyID. This operation can be used in the many situations where a picklist such as a combobox or listbox is presented to the user. From this picklist the expectation is that a relationship with the record or object will be established. The GUI can then use the KeyID (the value in the combobox) as the stored item. With this unique Key ID we have an assurance that the current record is precisely linked to the specific record that we want and no others.

Implementation

Overview

Implementing this system requires a few basic items.

We need a method to generate the unique KeyIDs for each table in use by the application.

- This method must ensure the KeyIDs generated are persistent.
- It must ensure that the ID is unique in its use, where no two records in a table have the same ID. Preferably each ID would be used only once, ever.
- There must be enough IDs to cover not only the records that are created, but also any records that are subsequently discarded. The storage of these IDs needs to be efficient and not create an undue burden on the size of the table overall. An essential requirement of the generated KeyID is that it function well within CA-Visual Objects' index files.
- And finally this process should be automatic, minimizing the impact on the rest of our VO code.

ID Server

To solve the first requirement, I created a table to store the KeyIDs assigned to each table in the application. This KeyIDServer has a simple structure of two fields (see Table 1) that allows a KeyID for each table being used in the application. I could have had this KeyIDServer table contain only a single record with only a single field for the last KeyID. Such a setup would however mean that I would need a larger field for the KeyID, since all records in the application would be using the same series of IDs. This would mean that even tables with a small number of records would need a large field for storing the key.

Table 1: Structure for database: Key_ID.DBF

Field	Field Name	Type	Len	Dec
1	SERVERNAME	C	25	0
2	LASTKEY_ID	C	5	0
		Total:	30	0

It can be noted that for each server name the "last" KeyID is stored. The alternative of storing the "next" KeyID is not implemented for a specific reason. I wanted to ensure that two tables could not access the KeyIDServer and get a KeyID at the same time. If getting the next KeyID were simply a matter of reading the KeyIDServer table, then duplicate IDs might creep into the system. As will be shown later, a method to get the next ID ensures that only one calling table at a time gets a KeyID. I also want to be in complete control of the record locking process, so I turned off the concurrency control (as shown in Source Code 1).

The ServerName field above will store the same value returned by calling the dbServer:FileSpec:FileName access attribute. This means that the file path and file extension should not be entered in the ServerName field.

This Key_ID.dbf must also have a single index created for it, allowing the application to rapidly seek on the ServerName. The indexing code block should be {|| Upper(ServerName)}.

```
CLASS HHKeyIDServer INHERIT  dbServer

METHOD Init( oFileSpec, lShareMode, lReadOnlyMode, cDriver, acRDDs )  CLASS HHKeyIDServer
        SUPER:Init( oFileSpec, lShareMode, lReadOnlyMode, cDriver, acRDDs )
        // turn off auto record locking
        SELF:ConcurrencyControl := CCNONE

ACCESS ID CLASS HHKeyIDServer
        RETURN ( SELF:FIELDGET( #LASTKEY_ID ) )

ASSIGN  ID( str ) CLASS HHKeyIDServer
        RETURN ( SELF:FIELDPUT( #LASTKEY_ID, str ))
```

Source Code 1: KeyIDServer class declarations and Init()

Assigns and accesses were also created to make it simpler to access the LASTKEY_ID field.

The above table provides the means to store the KeyIDs persistently outside the application, so now we need to look at how a calling table is assigned a new KeyID. This is done by making a call to the KeyIDServer's nextID() method.

nextID()

Getting a KeyID is performed by making a call to the KeyIDServer asking for the nextID() and passing the file name of the calling table. This method will return the next KeyID that would then be used by the calling table. The calling table will then assign it to a newly created record as its KeyID.

Ensuring that the KeyIDs are unique is the function of the KeyIDServer. This uniqueness is enforced by only returning a new KeyID after a record lock has been established. As Source Code 2 shows, the name of the calling table is passed in as a string parameter. This name is used by the KeyIDServer to perform a seek to find the single record for that table. If the record is found then an endless attempt is made to lock that record. Only if the record is successfully locked does the KeyIDServer create a new KeyID, by calling the "increment" function. And then, before releasing the lock, the KeyIDServer stores the newly created KeyID into the record as the LastKeyID. A call to commit the new information is made to ensure that the new KeyID is immediately written to the Key_ID.dbf file.

At this point two important requirements have been met:

- Only a single calling table had access to get a new KeyID. This was accomplished by requiring a record lock prior to performing any action.
- Any other tables attempting to get a new KeyID will be blocked from getting the necessary record lock until after the LastKeyID field has been updated - precluding assignment of a duplicate KeyID.

The last thing I do is to unlock the record and return the new KeyID value back to the calling table.

```
METHOD NextID( cTableName ) CLASS HHKeyIDServer
	// return a unique ID number for the passed table name, passed as string
	LOCAL cTempKey AS STRING
	LOCAL cText:="Server name is blank, check source-CODE" AS STRING

	SELF:Seek( Pad(cTableName,25), FALSE )			// no softseek search
	IF SELF:Found
		// we have turned off concurrency control by VO
		// so we do it our selves, hopefully faster.

		WHILE ! ( SELF:LockCurrentRecord() )
		ENDDO
		// nothing is done until a record lock is achieved,
		// this ensures that only one calling table record gets a KeyID at a time.
		cTempKey := Increment( SELF:FIELDGET( #LASTKEY_ID ) )

		// we update the last KeyID before releasing the record.
		SELF:FIELDPUT( #LASTKEY_ID, cTempKey )
		SELF:Commit()

		SELF:UnLock( SELF:RecNo )

		RETURN ( cTempKey )

	ELSEIF !Empty( cTableName )
		cText :=   "Server "+cTableName +" missing, not found in Key_ID.dbf"

	ENDIF

	textbox{ , "Server Not Found", cText }:show()

	RETURN NULL_STRING
```

Source Code 2: KeyIDServer nextID() method

The nextID() method performs the necessary actions to ensure that only one table record gets a KeyID at a time. The next issue is how to make the KeyID as efficient as possible. Inside the nextID() is a call to the function Increment(), which is discussed next.

Incrementing in ASCII

The increment function is stored statically within the source of the KeyIDServer, ensuring that only this server is capable of calling this action. This function has the responsibility of taking a passed string parameter, which is the last KeyID previously assigned, and generating a new KeyID from the parameter. This function considers the KeyID to be a series of ASCII characters. The use of ASCII characters was chosen to maximize the number of unique IDs within a short field length. Using only digits allows for 10 values per space; adding letters increases that number to 62 (lower and upper case). By adding the other printable characters, the number can be increased to 95 for each character space of the KEY_ID field used by the UniqueIDdbServer class, discussed below. These characters range from ASCII number 32 up to 127. This range was chosen to ensure that indexing in Visual Objects returned consistent results. As shown in Table 2, by allowing for 95 values in each space, an extremely large number of unique KeyIDs can be stored in a relatively small field.

Table 2: Maximum capacity for unique KeyIDs dependent upon the length of the KeyID field

Possible Values	# of Spaces	# of Spaces	# of Spaces	# of Spaces	# of Spaces
95	6	5	4	3	2
Max Key IDs:	735,091,890,625	7,737,809,375	81,450,625	857,375	9,025

Note: To ensure the correct behavior of the ASCII index, the application should set collation to Clipper, by calling the "setCollation(#CLIPPER)" function in the start() method of the application.

```
STATIC FUNCTION Increment( str AS STRING ) AS STRING // may be empty
        // returns an incremented string, some thing like "255, 0, 0, 0" although its in ASCII
        // characters
        LOCAL nLen := Len ( str ), i, nCount, nPos AS INT
        LOCAL c, cStuff AS STRING
        LOCAL n AS WORD

        // first figure out where to increment
        i := nLen
        IF Asc( SubStr( str, nLen, 1 ) ) < 127

           c := SubStr( str, nLen, 1 )      // we start at the right end
           n := Asc( c )                    // convert the character to asc
           c := CHR( ++n )// we did an increment so reconvert back to character, and
                                            // stuff it back in.
           str := Stuff( str, nLen, 1, c ) // stuff it here
           RETURN Str

        ELSE
          // can not increment the last element so move backwards (to the left)

           nCount:=1
           nlen--
           FOR i := nLen TO 1 STEP -1
                IF Asc( SubStr( str, i, 1 ) ) < 127      // Chr( 127 ) and higher yields unpredictable
                                                          //results
                        nPos := i
                        EXIT
                ENDIF
                nCount++
           NEXT
        ENDIF

        c := SubStr( str, i, 1 )
        n := Asc( c )          // convert the character to ASCII
        ++n

        c := CHR( n )          // we did an increment so reconvert back to character, and stuff it back in.
        cStuff := c + Replicate( CHR(33), nCount ) // put leading spaces

        IF nPos == 0

            WarningBox{ , "Data-Structure Error", "Key ID for this database has been exceeded!" + ;
                "Database structure needs to increase KEY_ID to a larger field size" }:show()
            RETURN ""

        ELSE

           str := Stuff( str, nPos, 1+nCount, cStuff )

        ENDIF
        RETURN Str
```

Looking at the increment function, it begins by looking at the last entry of the passed string. If the ASCII value of that character is less than 127 it is incremented by one, converted back into a character and then stuffed back into its original position. The result is then returned. The reason for the 127 limit is that characters with ASCII values higher than 127 are each interpreted the same by CA-Visual Object's indexer. Indexing thus becomes corrupted.

If the ASCII value of the parameter's last character is equal to 127 however, the method must skip one character to the left, until a character is found with an ASCII value less than 127. If a value less than 127 is found, it is incremented as before. The character of that incremented ASCII value is then stuffed back into the original position, and the new KeyID is returned. A check is performed to be sure that the field length of the KeyID has not been exceeded. If it has, a warning is presented and a blank string is returned.

So far we have seen how the KeyIDServer functions and the single method to be called to make it return a new KeyID. However, our final intent of this implementation is to make the process of assigning unique KeyIDs as automatic and painless as possible. The following shows how the class of UniqueIDdbServer is created.

Unique ID dbServer

The final step in implementing the serialization of CA-Visual Objects is to create a new class of dbServer that will have as its basic capability unique KeyIDs. This class inherits from the dbServer class. This class should be high in the inheritance tree of your dbServer classes. However, if a buffered dbServer class exists, this UniqueIDServer class should inherit from the buffered class. This is so that calls to commit() by the UniqueIDServer do not preempt the buffering action of the buffered dbServer class.

Two attributes of the UniqueIDdbServer are created. The first, "symKeyFieldName", stores a protected reference to the symbol of the field that will be used as the KeyID for the table. This allows the programmer to override the default symbol of "KEY_ID". Access and assigns allow subclassed objects to reassign new field names to this instance variable.

The second attribute is an object reference to the KeyIDServer object that we have discussed previously. Each UniqueIDdbServer will contain its own reference to the KeyID generator. This reference is opened in the UniqueIDdbServer's Init() method. The notification function of the KeyIDServer is turned off, since I do not want any other windows or servers to be notified of record motion by the KeyIDServer. Finally, the Close() method of the UniqueIDdbServer adds the lines to reset notification and to properly close the KeyIDServer.

```
CLASS HHUniqueIDdbServer INHERIT  dbServer
        // this server ensures a unique ID for each record
        PROTECT symKeyFieldName AS SYMBOL  // stores the name of the key_id field as a symbol
        PROTECT oKeyIDServer AS KeyIDServer
```

Source Code 3: UniqueIDdbServer class declaration

```
ACCESS KeyField CLASS HHUniqueIDdbServer
        RETURN symKeyFieldName

ASSIGN KeyField( symName ) CLASS HHUniqueIDdbServer
        symKeyFieldName := symName
        RETURN symName
```

Source Code 4: UniqueIDdbServer accesses and assigns

```
METHOD Init( oFileSpec, lShareMode, lReadOnlyMode, cDriver, acRDDs    ) CLASS  ;   HHUniqueIDdbServer

        SUPER:Init( oFileSpec, lShareMode, lReadOnlyMode, cDriver, acRDDs      )

        // our own copy of the KeyID server
        SELF:oKeyIDServer := KeyIDServer{}
        SELF:oKeyIDServer:SuspendNotification()
        // default to this symbol name
        SELF:symKeyFieldName := #KEY_ID

        RETURN SELF
```

Source Code 5: UniqueIDdbServer Init() method

```
METHOD Close( ) CLASS HHUniqueIDdbServer
        LOCAL lSuccess AS LOGIC
        IF ( lSuccess := SUPER:Close( ) )
                // close our own copy of the KeyID server
                SELF:oKeyIDServer:ResetNotification()
                SELF:oKeyIDServer:Close()
        ENDIF

        RETURN lSuccess
```

Source Code 6: UniqueIDdbServer Close() method

The preceding has set the groundwork to implement automatic calls for assigning unique IDs to the UniqueIDdbServer. This is done within the append() method.

Append()

The best time to assign a serial ID to a record is as soon as it is created and before any other action is performed on the record. This ensures that any subsequent actions that depend upon the presence of the KeyID will function correctly. Therefore, the last step in this implementation is performed within the Append() method of the UniqueIDdbServer class.

After a successful append is performed by the super class, the UniqueIDdbServer still has a successful record lock on the current newly appended record. The KeyIDServer is then asked to generate a NextID. The UniqueIDdbServer passes in its own file name. As shown above in Source Code 2, the KeyIDServer takes the passed file name and uses it to seek to the record for that table where it holds the last KeyID assigned. The KeyIDServer then generates a new KeyID and passes it back to the calling table.

After getting the NextID, the UniqueIDdbServer puts the new KeyID value into the field identified as the Key field.

We then return to the calling application with a successfully serialized record.

```
METHOD Append(lreleaselocks) CLASS HHUniqueIDdbServer
	LOCAL lSuccess AS LOGIC
	LOCAL cNextID AS STRING

	// append a blank record, releasing locks on other records as appropriate
	IF ( lSuccess := SUPER:append(lreleaselocks) )
		// if successful, then create a unique ID
		// gets file name without path and extension
		cNextID := SELF:oKeyIDServer:NextID( SELF:FileSpec:FileName  )

		// put the newly created ID into the proper KeyID field before anything else happens
		SELF:FIELDPUT(  SELF:symKeyFieldName, cNextID )

	ENDIF

	RETURN lSuccess
```

Source Code 7: UniqueIDdbServer Append() method

Example in Jasmine - Java

Jasmine does not need a way to generate unique IDs in order to establish the basic linkage among objects with the database. This is performed automatically by Jasmine through the creation of Object IDs (OIDs). However, these OIDs are not readily accessed by external applications for use as identifiers. While developing a Jasmine application where the GUI was to be written in Java, the issue of communication between the GUI and the Jasmine database arose.

We wanted the GUI to retrieve a list of possible choices from the Jasmine database. An example would be where a listing of staff was being presented to the user. This listing would have to include several attributes about every person in order for users to be sure they had correctly identified the person they wanted. The basic process created was for the Jasmine application to provide a collection of strings making up the list of staff, with each string being a concatenation of the various attributes about each person object. This aspect worked well; however, once the user made a selection, we had to then perform a scan on the class of people using all the attributes that were used to identify the person.

What we desired was a serial number that we could append to the concatenated string used to make up the picklist. We could then pass this serial number back into Jasmine and perform a scan using this relatively smaller string. We could even build and maintain an index on this attribute to enhance performance.

We therefore attempted to mirror the process I described above for CA-Visual Objects.

The plan was to create a serial class high in the inheritance tree that would provide the basic functionality of a serial class, see Figure 1. The idea was that we would create a class level attribute to store the LastKeyID value. Similarly there would be class level methods to get the nextKeyID(). These two properties would effectively take the place of the KeyIDServer discussed above in the CA-Visual Objects version. The last step was to refine the basic new() method to automatically assign the KeyID to the newly created object, again mimicking the process I used in CA-Visual Objects.

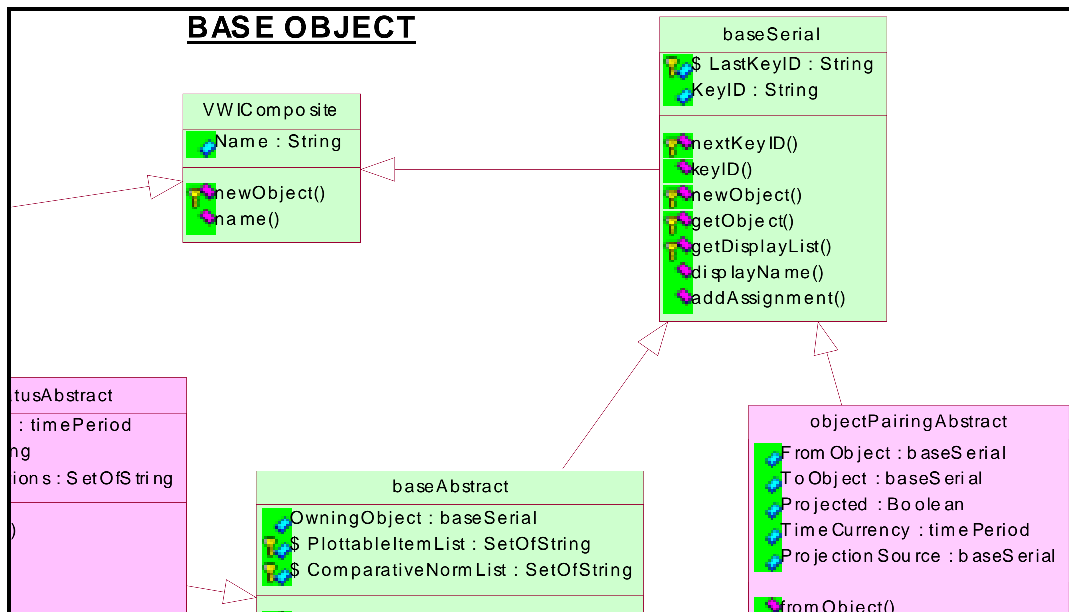

Figure 1: UML model of the top of the tree in a Jasmine application

The difficulty was that inheritance and refinement are not as clean and simple as in CA-Visual Objects. The problem is that Jasmine does not allow the new() method to be refined. This method is the base class method that can accept any "name=value" pairs as parameters. These parameters are typically used to pass values to mandatory attributes. However, this method may not be directly refined.

The solution approach we took was to create our own newObject() method that would wrap around the super.new() method. One problem that we did not resolve, however, was the rules on refinement of methods. Jasmine requires that when refining methods, the number of parameters be the same between the original method and the refined method. The parameter types must also be consistent with those as originally defined. Since this serial class was intended to be one of the base classes from which numerous other classes would inherit, we had the problem where we could not anticipate all the parameters that we might need. The only parameters that must be passed when creating a new object are those for mandatory attributes. We therefore decided to not declare any attributes as mandatory to Jasmine.

The following code sections show how we implemented the newObject() and nextKeyID() methods. The class level method to retrieve an object from a class based upon a passed KeyID is also shown.

```
defineProcedure baseSerial baseSerial::class:newObject()
{
  /* call nextKeyID() to get a unique KeyID for object.  call super.new() to create a new
     object.  assign new KeyID to new object.  return new object with KeyID.  */

            $defaultCF VWI;
            $String id;
            $id = baseSerial.nextKeyID();
            $return(self.new(KeyID := id));
};
```

Source Code 8: Jasmine newObject() method

```
defineProcedure String baseSerial::class:nextKeyID()
{
  /* Lock value of LastKeyID so no one else can use it.  Increment LastKeyID and return it */
  $defaultCF VWI;
  $String returntemp;
  $String holder;
  $Integer lkeyID;
  $returntemp = baseSerial.LastKeyID;
  $lkeyID = baseSerial.LastKeyID.convertToInteger();
  $lkeyID = lkeyID + 1;
  $holder = lkeyID.convertToString();
  $baseSerial.LastKeyID = holder;
  $return(returntemp);
};
```

Source Code 9: Jasmine nextKeyID() method

```
defineProcedure baseSerial baseSerial::class:getObject(String cKeyID)
{
	/* refine this for all sub-classes */
	$defaultCF VWI;
	$Bag<baseSerial> basebag;
	$baseSerial temp;
	$basebag = baseSerial from baseSerial;
	$scan(basebag,temp) {
		$if(temp.keyID() == cKeyID)
		{
			$return(temp);
		};
	};
	$return(NIL);
};
```

Source Code 10: Jasmine getObject() method

Summary & Conclusion

This paper discussed the need for creating unique serial IDs for records in a DBF table. It showed how to create a system to automatically generate and assign such Key IDs to each record at the time of appending. This included using the full range of ASCII characters when creating these IDs and showed how a limited field size could support a large number of unique IDs. Finally, it discussed how such a unique and accessible ID could be generated in Jasmine.

The preceding source is available on www.Hungry-Hippo.com

Biography

Bill Cross is a Senior Functional Analyst and Health Facility Planner for VW International, Inc. an engineering and management service firm located in Alexandria, Virginia. As part of his career in medical facility design, he has acquired skills in xBase programming including 8 years of Clipper and 5 years using CA-Visual Objects. He has developed applications in Clipper and CA-Visual Objects and is currently working on updating a Clipper application in use by a Texas Utility company. He has also led object analysis and design efforts in support of Jasmine projects.