Sequential key allocation strategies in Smalltalk

Dayle Woolston & Chris Kesler

DATABASE APPLICATIONS COMMONLY require unique, sequential keys for inserting new records into tables. For example, a call tracking system may allocate a case number or a billing system may allocate a *customer id*. In some applications, the sequential key may be the preferred method of retrieval (as in locating cases in a call tracking system).

We present several client strategies for key allocation. One size does not fit all. System administrative policy, application features, and overall performance expectations must be considered carefully in selecting an algorithm for generating sequential keys. Accordingly, we examine the following factors:

1. Direct access versus stored procedure access
2. Single versus multiple key allocation
3. Error handling
4. Explicit locking versus browse mode concurrency management strategies

The code samples discussed use VisualWorks 2.0 Smalltalk from ParcPlace Systems and Sybase System 10 from Sybase. The client/server dialog is implemented using the basic mechanisms of the VisualWorks 2.0 Database Connect driver for Sybase.

DIRECT VERSUS STORED PROCEDURE ACCESS

By *direct access* we mean the application issues SQL to generate the next key. In this case, the application must be intimately knowledgeable of various schema management issues: Are the next key seed values all contained in the same table within different records, or does each data table have a corresponding key table? Also, how does the client arbitrate multi-application access of the key table? Should the table be locked during sequential key allocation or should the application select for browse?

Sequential key allocation is a good candidate for stored procedure implementation. The function is simple, highly dependent on the schema, and a focal point of client activity. Should one application implement key allocation improperly, it could wreak havoc on the database.

SINGLE VERSUS MULTIPLE KEY ALLOCATION

Conceptually, *single key allocation* is the simplest strategy. Each time a client requests a new key, it receives the next ordinal number. This ensures that selects ordered on the key parallel selects ordered on the insertion timestamp for each record. This may or may not be an application requirement, and assumes the key is allocated only after the record has been validated. If the client should abandon the insertion and terminate, the key would be lost. Conversely, this strategy may be affected by when it is necessary in the end-user dialog to make the key available. If the user must know the key prior to server-based validation, it is not possible to guarantee that the key will parallel the insertion timestamp.

There are many performance factors affecting key allocation implementation. Requesting keys is relatively expensive because the client must communicate with the server over the network. Also, because hundreds of clients may compete for the next key, there is a potential system bottleneck. A solution is to have the server provide a range of keys each time the client makes a request. How many keys is enough? The answer depends on the application. In fact, objects managed by an application most likely require different ranges. For example, in a call tracking system, case numbers may be required to be sequential in time, whereas new customer numbers may have no such requirement. Models should provide single and multiple key allocation strategies. We now examine three implementations of key allocation.

SIMPLE UPDATE AND SELECT KEY ALLOCATION

In this first example, the database contains a table named KEYS. There is a record in KEYS for each data table requiring a sequential key. The method is called with the table name as its parameter and returns an integer value.

```
nextKeyFor: aTableName
    |ans|
    self session
          begin;
          prepare: 'UPDATE KEYS SET serial = serial + 1',
                   ' WHERE tableName = ''', aTableName,''' ';
          execute;
          prepare: 'SELECT serial FROM KEYS where
                   tableName = ''', aTableName, ''' ';
          execute.
```

```
        ans := self session answer atEnd; next.
        self session commit.
        ^ans first first.
```

The method works because the first statement invokes a transaction that must place a write lock on the KEYS table. This first statement updates the serial value of the record whose tableName value contains the string aTableName. The next statement selects the serial value and returns the integer to the caller. The commit releases any locks; other clients may proceed to complete the same dialog with the server.

This code makes several assumptions. First, it does not explicitly check for any rollback condition. It may be that the client does not have rights to update the KEYS table. This method should encapsulate the execute statement with a handler that enforces a rollback on such conditions. The following code creates an example signal handler. In production code you will want to preallocate the signal handler.

```
    nextKeyFor: aTableName
        |ans noUpdateSignal|
        noUpdateSignal := Signal new notifierString:
                            'Unable to get the next key from ' ,
                            aTableName.
        noUpdateSignal
            handle:
                [:ex |
                self adminConnection rollback.
                Dialog warn: ex errorString. ex return]
            do:
                [self session
                begin;
                prepare: 'UPDATE KEYS SET serial = serial + 1',
                            ' WHERE tableName = ''', aTableName,''' ';
                execute;
                prepare: 'SELECT serial FROM KEYS where
                            tableName = ''', aTableName, ''' ';
                execute.
                ans := self session answer atEnd; next.
                self session commit].
        ^ans first first.
```

Second, there may be no record in the KEYS table containing aTableName as the value of its tableName column. (Perhaps some coding error misspelled the table name string.) In this case, the code would execute without error but always return the most recent key in use. The end result would be an insertion error if there were a unique index on the key column of the data table for which the key is intended. This condition can be detected by checking the rowcount attribute for the session after each statement gets executed. It should be 1 in each case.

The final assumption is that the key must be preincremented. This is the type of schema assumption that is often documented too casually, causing problems down the road. If this method services all applications accessing the database, then this assumption is probably adequately handled; however, this is very unlikely. The more likely case is that the table will be accessed by heterogeneous clients: 4GLs, Smalltalks, C, C++, etc. This preincrement policy makes this function a good candidate for stored procedure encapsulation rather than direct implementation.

STORED PROCEDURE KEY ALLOCATION

The next example suggests how the nextKey method may contract with a stored procedure called nextKey.

```
    nextKeyFor: aTableName
        ^useStoredProcedures
            ifTrue:
                [self session
                        prepare: 'nextKey ' , aTableName, ', ',
                                    aNumber asString;
                        execute.
                (self session answer atEnd; next) first first]
            ifFalse:
                [self embbededSQLNextKeyFor: aTableName].
```

In this example, the database framework has some control over whether it uses stored procedures. If so, the first clause gets executed, otherwise execution gets redirected to another method (a new name for the previous example).

CONCURRENCY ISSUES

One of the vital issues we've avoided so far is that of *concurrency control*. Two separate clients cannot execute the preceding examples at the same time. The two options are *locking* and *browse* mode. Executing locks in this type of method is a pessimistic form of concurrency control. It requires signal handlers to detect the lock condition and possibly repeat attempts until the competing client clears the lock.

In our remaining example, we prefer an optimistic concurrency control strategy using what is known as browse mode. In browse mode, the client issues no lock requests. Instead, the row contains a column of type timestamp. The database service updates the timestamp value each time it performs a select against the row. In requesting the serial value, the client also requests the timestamp value. It then uses the timestamp value as a where clause restriction when updating the row to the incremented serial value. Only the client with the most recent timestamp succeeds in updating the serial value. This success indicates to the method that it may return a valid key to its caller. All other clients executing the same method execute their signal handler and make another attempt at fetching a new key.

Browse mode requires that the select statement end with the words "FOR BROWSE." The target table must have a unique index and, as noted, a timestamp column.

MULTIPLE KEY ALLOCATION IN BROWSE MODE

Our final example demonstrates a rich set of services. The method returns a collection of one or more sequential integer keys for the table aTableName, enabling the client to (possibly) cache multiple key values. We also use browse mode to implement an optimistic concurrency management strategy.

The method is composed in four sections. The first section sets up the method by assigning a "1" to the attempts counter, allocating a stream to compose queries, and creating a signal instance to manage any exception conditions. The second section defines the exception handling clause. When an exception occurs, the connection must rollback the transaction and increment the attempts counter. If more than 10 attempts occur, notify the user with a dialogue box and error out. Otherwise, just try again. The third section composes and executes two SQL statements. The first statement retrieves the key and timestamp values; the second statement attempts to update the key value. This section checks the session rowcount value to verify that the update occurred. This value should be 1. The fourth and final section creates an ordered collection and assigns to it a sequence of anInteger integers, beginning with the first available key value.

```
nextKeyFor: aTableName incrementBy: anInteger
    | currentData timestamp noUpdateSignal attempts oc
    aStream |
    attempts := 1.
    aStream := (String new: 75) writeStream.
    noUpdateSignal := Signal new notifierString:
                    'Unable to get the next key from ' ,
                    aTableName.
    noUpdateSignal
        handle:
            [:ex |
            self adminConnection rollback.
            attempts := attempts + 1.
            attempts > 10 ifTrue: [Dialog warn: ex
                            errorString. ex return].
            ex restart]
        do:
            [self adminConnection begin.
            self adminSession
                    prepare:
                        'select key, timestamp from ',
                        aTableName , ' FOR BROWSE';
                    execute.
            currentData := self adminSession answer atEnd;
                        next.
            timestamp := currentData last.
            aStream
                nextPutAll: 'update ';
                nextPutAll: aTableName;
                nextPutAll: 'set key = Key+';
                print: anInteger;
                nextPutAll: ' where timestamp = ';
                sqlPrint: timestamp.
            self adminSession
                    prepare: aStream contents;
                    execute.
                    answer.
            self adminSession rowCount < 1 ifTrue:
                    [noUpdateSignal raise].
                    self adminConnection commit].
    oc := OrderedCollection new: anInteger.
    lastUsedId + 1 to: lastUsedId + anInteger do: [:i | oc
    add: i].
    ^oc.
```

Each of these examples is acceptable depending on assumptions that must be supported by the client. This last example, however, illustrates several important optimizations. First, the algorithm for fetching the next key and updating its row on the server is enapsulated by a signal handler. Second, the method checks the session rowcount value to ensure the update actually occurred. If it did not, the signal is raised and the operation attempted up to 10 times. Third, the SQL is constructed using a write stream; a faster strategy than successive string concatenation with commas.

Despite the optimizations, the code is written for clarity over performance. A production version would create the signal at initialization time and keep it in a class-side dictionary. It doesn't make sense to create a new signal during each request for a new key. Also, it is important to allocate a string for the write stream that is very nearly the size of the largest SQL statement. Performance profiling demonstrates that write stream creation time is dominated by the size of the string allocated for the stream. Don't allocate a 500-byte string and think the stream operations are saving you any time over string operations with the comma operator. The best solution is to create the string only once, creating the write stream on that same string over and over with each call to the method. Finally, the ordered collections provide convenient collection services. However, they are considerably slower than arrays. The final section should probably be implemented with an array.

It is not immediately obvious, but this strategy also allocates a separate connection/session for servicing next key requests. It may be that this method executes in the context of saving many objects to the database bounded by a single begin/commit pair. As each object gets saved, it must request the next available key, which (as illustrated) requires its own transaction control. This alternate administrative connection executes key retrievals within a separate transaction. (It is customary for browse mode strategies to employ two connections.)

CACHING KEYS

The remaining code illustrates how a client may cache multiple (sequential) keys. This strategy is appropriate in many cases; it can greatly reduce the number of database requests, thus enhancing overall client performance and reducing network traffic. Without such a strategy, each insert operation generates two database transactions, first to fetch a key and second to perform the insert.

```
nextKey
    | key |
    (keyCache isNil or:
[keyCache isEmpty])
```

continued on page 32

GETTING REAL *continued from page 19*

```
    name first clusterInBucket: empCluster.
    name middle clusterInBucket: empCluster.
    name last clusterInBucket: empCluster.

    " cluster the address and its components "
    address := anEmp address.
    address clusterInBucket: addressCluster.
    address street clusterInBucket: addressCluster.
    address city clusterInBucket: addressCluster.
    address state clusterInBucket: addressCluster.
    address zip clusterInBucket: addressCluster. ].
```

This column has described how to determine if clustering objects might help application performance and how to cluster objects using ClusterBuckets. My next column will discuss how to measure overall system performance and steps for tuning multi-user Smalltalk for higher transaction throughput.

THE BEST OF COMP.LANG.SMALLTALK

continued from page 23

- Avoid commitment—This is another way of expressing the principle of postponing decisions but one that might strike a chord with younger or unmarried programmers.
- It's not a good example if it doesn't work—This one comes from David Buck (dbuck@magmacom.com), who's fed up with looking at example and test methods that haven't been properly maintained as the code evolved. I can't think of a way to apply this to life but it's good advice anyway.
- Steal everything you can from your parents—A principle for those trying to make effective use of inheritance or moving into their first apartment.
- Cover your a**—Like in a bureaucracy, the most important thing is to make sure that it isn't your fault. Make sure your code won't have a problem even if things are going very wrong elsewhere.

SEQUENTIAL KEY ALLOCATION

continued from page 26

```
        ifTrue: [ keyCache := self nextKeys: self
                  keyCacheSize ].

    key := keyCache first.
    keyCache removeFirst.
    ^key.
```

If you choose to make the array optimization in the nextKeys: method, this method must be changed to insert nil values into the array as each key gets returned rather than using the removeFirst selector.

Dayle Woolston and Chris Kesler have been working with Smalltalk for 4 years building client/server database applications. They can be reached at dayle_woolston@novell.com and chris_kesler@novell.com.