Managing Multiple Record Entries
Part I

Lincoln Stoller, Ph.D.
Braided Matrix, Inc.

Contents of Part I

Contents of Part II

I. Introduction to multiple record entries

Multi-record transaction processing involves updating information stored in several files and is common in more complex applications. To do this securely in a multiuser environment requires locking multiple records at the same time. This two part article discusses four different strategies for managing multiple record transactions.

A multi-record entry stores a single entry in several records and possibly several files. Relational consistency requires either that all the data be saved or, if some of the related records are locked, none be saved and the entry rejected. This article reviews methods for insuring write access to a group of records.

While the primary goal in each of these methods is managing write access, their performance is also considered on the following criteria:

- Simplicity of code:

The technique must be easy to understand and implement. If it is necessary to make changes in the program, the code should be easy to modify.

- Speed of execution.

- Robustness over a network:

The database must remain fully functional in the event a user entering data crashes off the network.

- Impact on network function:

Write access should be as selective as possible so as not to lock other users out of any more records than necessary,

II. Techniques

Four basic approaches to checking and sustaining write access to multiple records in multiple files are considered — global transactions, hierarchical access, setting flags, and multiple record locking.

A. Global transaction

When a global transaction is initiated by one user it locks all others out of the database (and prevents them from switching out if the database under multifinder). All other database nodes are frozen until the transaction is completed. A user starts a global transaction by calling the routine **START TRANSACTION.**

In version 2 of 4th Dimension (4D v2) the user can start global or multitransactions by issuing either the **START TRANSACTION** or **START TRANSACTION**(*) command. In 4D v3 you can only use either global or multitransactions throughout the application. The START TRANSACTION command is used in both cases and the structure file must be set up to support either global or multitransactions exclusively. The structure file is set up using 4D Customizer to modify the v2/v3 compatibility resource. (cf. 4D Utilities Guide, p.1-27)

During the course of the transaction, all modifications are stored in local memory. Changes are written to disk only when the transaction is completed with a call to the routine

VALIDATE TRANSACTION. This initiates the recording of all changes accepted through an input layout or assigned procedurally and marked for saving with a **SAVE RECORD** command.

In 4D v2 validating a transaction causes it's effects to be written to disk immediately. 4D v3 supports caching of data in multiuser and normally will not write the changes immediately to disk even though the changes will immediately be seen by all users. If you want to force the changes to disk use the **FLUSH BUFFERS** command.

Alternatively, if some condition is not met during the transaction — typically, certain records being locked — the global transaction can be completed with a call to the **CANCEL TRANSACTION** routine. The database is unlocked without writing changes to disk.

Global transactions are useful in situations where global database maintenance is performed. Preventing access to the database is acceptable only if such exclusion is necessary for the purposes of the procedure being performed.

B. Hierarchical access

A hierarchical structure is one where there is a many file related to a one file and where the records in the many file are only accessed through the one file. In this case, access to a record of the many file is only allowed when the parent record in the one file is unlocked.

Here the one file performs two functions. It is acting both as repository for data and as a flag to control access to the related records. This method is easily extended to cases where several files exist in a many-to-one relation to a given parent file. Figure 1 shows a hierarchical structure.

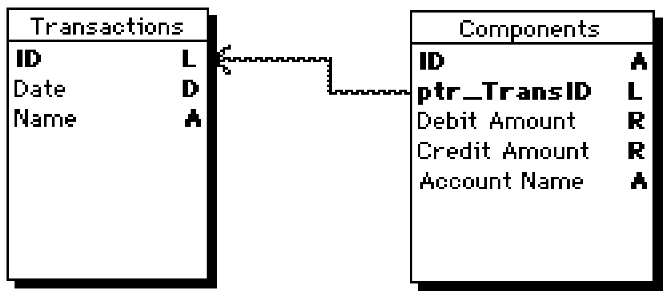

Figure 1.

Controlling access using this method has the benefit that it follows naturally from the database structure. Because of this, it excels in each of the four performance categories. However, this performance can quickly degrade if there are exceptions to the hierarchical access criteria. Consider the non-hierarchical structure for an accounting database shown in Figure 2 (derived from the previous structure by adding an Accounts file). In this example, the accounts file has a credit balance field that stores the running total of all credits minus debits.

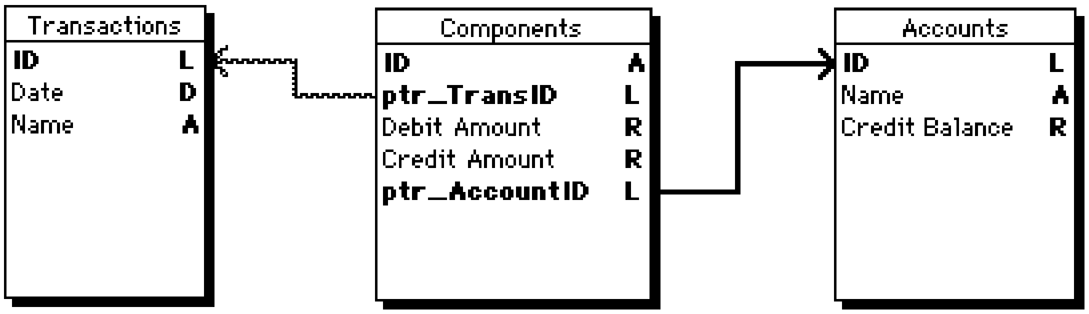

Figure 2.

If the previous hierarchical scheme is used to control access to the components file and one of the account records is locked, its credit balance cannot be updated. Another problem occurs if write access to components is needed through their accounts instead of through their transactions. That is, to modify the debit and credit records on an account-by-account basis. There are clearly ways to rectify these problems, but they require extensions to the hierarchical approach that compromise its simplicity.

C. Flags

A flag is anything that can be "raised" to signal users that some condition is in effect. Because it is process oriented, a single flag can control access to an arbitrarily large set of records

1. Semaphores

Semaphores are network-wide tags stored on the server, alongside the data, in a file referred to as the flags file. Semaphores take any text title and are set using the expression
Semaphore("a_string")
where "a_string" is any string of up to 8 characters in 4D v2 or 15 characters in 4D v3 (longer strings are truncated). In 4D v2 semaphores only work in multi-user mode; in 4D v2 single user mode the **Semaphore** function always returns false. In 4D v3 semaphores can always be set and clears.

If the semaphore does not exist, the **Semaphore** function creates it and returns false. In 4D v2 the semaphore function tests the statment "A semaphore by this name has already been created," and the function returns true or false. In 4D v3 the semaphore function tests the statement "A semaphore by this name has already been created *by another process*."

In both 4D v2 and 4D v3 a semaphore is created by the **Semaphore** command if none previously exists, that is if the function returns false. Version 3 also makes the distinction between local semaphores, those set by processes running on the local machine, and global semaphores, those set by any process running on the server. Local semaphores are semaphores which begin with a "$" (cf: 4D Language Reference 30-10).

A semaphore is erased by the statement
CLEAR SEMAPHORE("a_string")
In 4D v2, where semaphores don't have owners, any semaphore can be cleared from any node of the network. In 4D v3 only the process that set a semaphore can clear it. An important consequence of 4D v3's semaphore ownership is that all semaphores belonging to a process are cleared when that process ends either normally or abnormally (i.e. a user crashes off the

database). In contrast, semaphores in version 2 remained set even across periods where all users quit the database and then logged back on.

As an example, the statement
$AlreadySet := **Semaphore**("Transact")
could be executed in the during phase of a multi-record entry and then an **ACCEPT** command executed if $AlreadySet is false. If the semaphore is already set by a remote user or by another process, ($AlreadySet is true), the procedure can wait until the semaphore is free, or allow the user to cancel the entry.

2. Flag records

These are records used as flags and stored in a 4D file created especially for this purpose. Such records provide a semaphore-like system with some additional features:

- When the records are loaded and locked, they are associated with specific users.

- The records can convey special information.

- The programmer has full access to the contents of the file of flag records.

Such a file might either contain a fixed number of reusable records, or it may normally be empty. In the second case, records would be saved before an action and erased afterwards. These two approaches to managing flag records are quite different: creating and deleting customized records is more flexible than just loading and unloading preexisting flags but takes longer.

D. Individual record locking

Both of the following methods rely on checking and setting the write status of individual records. Locking only those records involved in a particular process can achieve a highly selective level of record management that has minimal impact on other users.

1. Stacking

In the memory of each user's machine there is a record stack for each file in the database. By pushing an unlocked record onto the stack, the local node places a copy of the record in memory and locks the record to all users. The lock persists through changes in the default file, the current record, and the current selection.

Stacking enables locking of any group of records. However, there is an important point to note when operating on a network:

An unlocked record loaded with write status is unlocked to the local user and locked to all others. However, an unlocked record that is loaded and then pushed onto the stack is **locked to everyone including the local user**. (This does not hold off the network where pushed records remain unlocked to the local user, also note the bug in version 3.0.1 described below.)

No changes can be made to a stacked record until it is popped off the local user's stack. As a previously unlocked record is popped, it is unlocked to the local user, though it remains locked to all others. Only at this point can changes be made and saved. (If the record was locked when it was pushed, it will remain locked after it is popped.)

Since the stacked records are not modifiable, variables need to be used for the input and temporary storage of modifications. As the records are popped, new field assignments can be made and saved to disk.

Since the stack is accessed sequentially, records must be unlocked according to stack order. There is no random access to the records, nor is there a way to tell what records or how many records are on the stack (unless they have been counted). There may be circumstances where it's easier to pop records until the stack is empty (as shown in the following code segment) rather than popping the number of records indicated by a counter.

Note that due to a bug in 4D Version 2, if a user with write access to records pushed on their stack returns to the splash screen without popping those records, they will be unable to unlock them. The records will remain locked until the user quits the database. This has been fixed in 4D v3 so that returning to the splash screen clears the record stacks.

The first release of the new version of 4D, version 3.0.1, exhibits the behavior that unlocked records pushed onto the stack appear locked to all process EXCEPT the local process, to which the records continue to appear unlocked. This unusual behavior is slated to disappear in the next update.

The code in Figure 3 locks unlocked records, calls procedures to get and assign changes, saves those records that have been modified, then unloads the selection. Clearing the stack relies on the behavior of **POP RECORD** — it does not alter the current selection if the stack is empty.

```
$AllUnlocked:=True
READ WRITE    ([File_1])
USE SET   ([File_1])
FIRST RECORD    ([File_1])
For ($j; 1; Records In Selection      ([File_1]))                 `Lock the selection.
        If  (Not  (Locked   ([File_1])))
              PUSH RECORD    ([File_1])                            `Record is now locked.
        Else
              $AllUnlocked:=False
        End if
        NEXT RECORD    ([File_1])
End for
`
If ($AllUnlocked)               `Check if all were accessible.
        MakeTempAssgnments                                         `Call procedure to store changes.
        Repeat
              POP RECORD    ([File_1])                             `Record is accessible
              $RecNum:=Record Number     ([File_1])
              If  ($RecNum>0)
                   DoRecordAssignments                             `Assign new values.
                   If  (Modified Record     ([File_1]))
                        SAVE RECORD     ([File_1])
                   End if
              End if
              UNLOAD RECORD     ([File_1])                         `Record is unlocked.
        Until  ($RecNum = -1)
Else
        ALERT  ("Some records locked; assignments could not be saved.")
        Repeat
              POP RECORD    ([File_1])
              $RecNum:=Record Number     ([File_1])
```

```
            UNLOAD RECORD    ([File_1])                      `Record is unlocked
      Until  ($RecNum = -1)                                  `if it was originally unlocked.
End if
```

Figure 3.

2. Multi-transactions

During a multi-transaction, changes made to the database by a local user are kept in memory; other users continue to have access to the database. If the multi-transaction is validated, all saved changes are written to disk; if it is canceled, the changes are discarded. In 4D v2 multi-transaction is started by the statement

START TRANSACTION (*)

and is either validated or canceled with calls to **VALIDATE TRANSACTION** or **CANCEL TRANSACTION**. In 4D v3 all transactions are either multi-transactions or global transactions as set using 4D Customizer. Which ever case is in effect the transaction is started by a **START TRANSACTION** command without any parameters.

Some aspects of record locking change during a multi-transaction. During a multi-transaction, an unlocked, loaded record that is saved before being unloaded is seen as unlocked to the local user, but remains locked to remote users until the multi-transaction is completed. Or to put it more simply: records that are saved remain locked to other users even after being unloaded. This applies to records saved procedurally. It also applies to records saved through an input layout either by issuing an **ACCEPT** command or by pressing an **ACCEPT** button.

If the multi-transaction is canceled, all record locks are erased and all records are unloaded. If the multi-transaction is validated, all previously unlocked records are again unlocked except for the current record, which remains locked and in memory.

This record locking behavior can be used to lock a set of records. Do this by starting a multi-transaction and looping through the selection issuing **SAVE RECORD** for each unlocked record. No modification is needed in order for the **SAVE RECORD** command to be effective in locking each record. In contrast with records that have been pushed on to the stack, the current set of records remains unlocked to the local user.

This selection of modified or newly created records can be randomly accessed, searched and sorted. In 4D v2 records created and saved after a multi-transaction had started could not be added to a set until the transaction was validated (if you tried to add them to a set they just wouldn't appear as members). In 4D v3 such records can be used in sets before the transaction is validated.

New field values can be assigned to the desired record directly. When assigning changes procedurally, another **SAVE RECORD** will be needed before unloading each record and before validating the transaction. After the multi-transaction is validated, all unlocked, saved records are written to disk without regard to whether any changes have actually been made.

The code in Figure 4 locks the records in the UserSet to all other users until the multi-transaction is complete.

```
START TRANSACTION     (*)                            `Omit the "(*)" in 4D v3 applications.

$Abort:=False
USE SET  ("UserSet")
```

```
READ WRITE   ( [File_1])
FIRST RECORD   ( [File_1])
For  ($j; 1; Records In Selection      ( [File_1))
          If  (Not ( Locked  ( [File_1))                    `Test for Read/Write Access
                SAVE RECORD   ( [File_1)
          Else
                $Abort:=True
          End if
          NEXT RECORD   ( [File_1)
End for

If  (Not ($Abort))                                          `Proceed only if all were unlocked.
          FIRST RECORD   ( [File_1)
          For  ($k; 1; Records In Selection      ( [File_1))
                MakeAssgnments
                SAVE RECORD   ([File_1)
          End for
          VALIDATE TRANSACTION
Else
          CANCEL TRANSACTION
End if
```

Figure 4.

In the next part of this article I'll compare these four methods on the criteria of simplicity of code, speed of execution, robustness over the network, and impact on users in a multiuser application.