Living Without Record Locking

By David Adams

Technical Note 06-38

Overview

4th Dimension and 4D Client/Server make it easy to ignore many of the complexities typically involved in developing a multi-process/multi-user database. Notably 4th Dimension and 4D Client/Server automatically manage record locking at the database engine level. This feature makes it possible to develop and deploy a multi-user/multi-process safe 4D system with virtually no special coding. However, there are a variety of situations when 4th Dimension programmers can't rely on the native record locking system alone, such as:

- Web- and SOAP-based record modification systems.
- Array and variable-based record modification systems, such as 4D, 4D Client, or 4D Open systems that uses AreaList, 4D View, or ListBoxes to display copies of values from records.
- File-based systems that combine, update, or otherwise modify records, such as records flowing from a series of remote offices back to a central database.

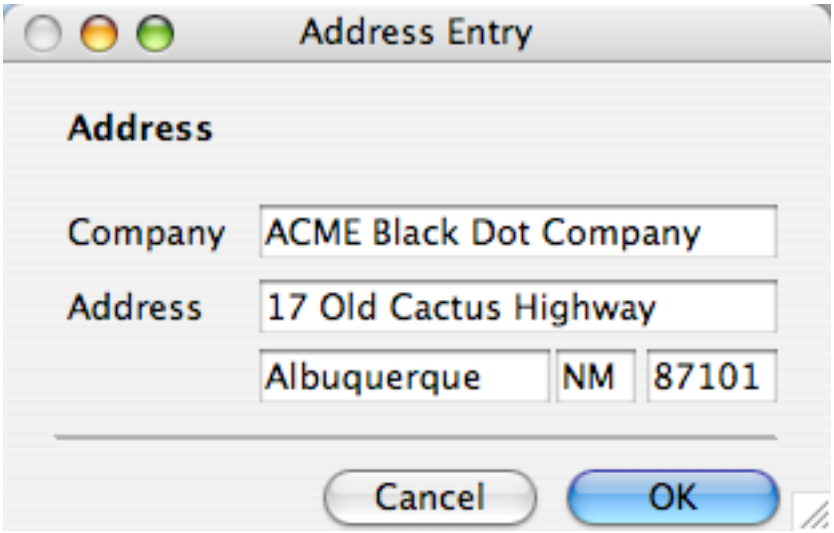
In these contexts, and others like them, records are not held locked centrally by the database engine. Therefore, another user or process may have deleted or updated the original record. Given these possibilities, custom record management code needs to test for, and address, three basic questions before updating data:

- Is the record locked right now but another process or user?
- Is the original record missing because it was deleted?
- Is the incoming record a stale copy, or is it based on the latest version of the original record?

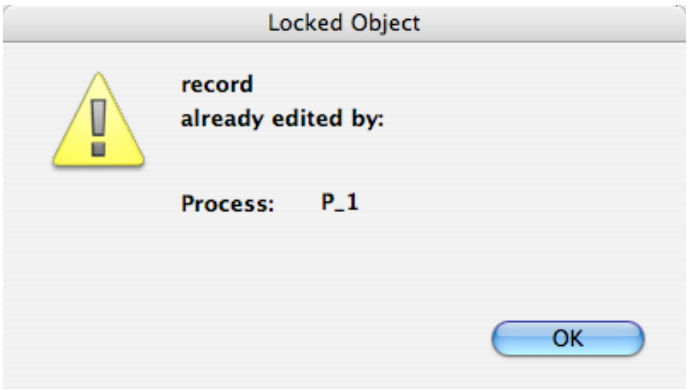
This technical note explains how to answer these questions and discusses how to implement a clear, consistent, reliable, and straightforward record update policy. By the end of this technical note, you'll see that answering these questions and writing the appropriate code is surprisingly simple. Before looking at these questions or their solutions in more detail, let's review how automatic record locking works in 4th Dimension. This background clarifies the custom solution and should help you avoid wasting time trying to emulate traditional record locking.

Review: Automatic Record Locking Features

Record locking within 4th Dimension and 4D Client/Server is fully automated when standard 4D commands and processes are used. Throughout this technical note we'll use a simple, imaginary address table as an example. A sample address record is pictured below:



Before displaying a record for editing, 4th Dimension and 4D Client/Server check if the record is already being edited by another process or user. If there is a conflict, 4D displays an alert, similar to the one pictured below:



As long as records are displayed in the User environment or with a standard command like **MODIFY SELECTION**, no custom code is needed to implement this behavior. Record locking in 4th Dimension is so simple and automatic that it is easy to take it for granted. However, the entire system depends on the carefully managed illusion that 4th Dimension or 4D Server display records. They don't, as we'll explain next.

4th Dimension Never Shows Records

Records can be displayed in 4D and 4D Client through forms in the user environment, or with **MODIFY SELECTION** or **DISPLAY SELECTION**. Regardless of how the data is presented, 4th Dimension only displays *copies* of records. In most cases there is no practical difference between showing records and copies of records. For example, in a single-process, single-user database, the distinction is purely academic. However, the difference between records and copies of records becomes critical any time more than one copy of a source record is active at once. This applies to triggers under 4D Server or any interface based on explicitly copies/translations of records, the subject of this technical note.

Record Locking and Record Copying

4th Dimension/4D Server manage record locking at the database engine level through 4D processes. When a process displays a record, the database engine loads the original values and sends a copy out for display or manipulation through code. The original record remains in the database engine. If the network connection is cut to a copy of 4D Client with unsaved record modifications, the original values are not changed, and the database itself is not harmed. Remember, 4D Client only works with copies of the master record, not the record itself. Making this system work depends entirely on 4D processes. When a record is held as locked, it is held within the context of a specific process. Once the process finishes, any records held locked by the database engine are released, and any local copies are discarded. In a 4D Client/4D Server or 4th Dimension setting, there is always a process to hold records. This is not the case with SOAP or Web connections. In those environments, once a request for data has been satisfied, the original process is either reused or closed. Therefore, there is nowhere to hold locked records open. Likewise, if values are copied into arrays or variables and records unloaded in a standard 4D/4D open/4D Client form, the original records are no longer locked.

Writing a Custom Record Locking System

When developers realize that record locking depends on maintaining a specific process, there is a temptation to try to build a custom process to manage record locking for SOAP, Web, and other situations that lack automatic record locking. 4th Dimension's own contextual Web serving mode, for example, is based largely on this idea. Except in the case where a custom system would provide special benefits, I'd strongly recommend not spending any time on writing a custom record locking system. Apart from being awkward to implement, it's entirely unclear how long records should be artificially locked for. One minute? Five minutes? One day? No matter what interval is selected, it will be too long in some cases and too short in others. Additionally, it isn't necessary to implement a custom record locking system. We'll look at why in a moment. First, I'll add a few words for anyone who still wants to write their own record locking system.

Sidebar: Tools for Writing A Custom Record Locking System

Not everyone readily accept that writing a custom record locking system is generally a poor idea. If you fall into this category, it's worth trying to write one. Even if the results are not satisfactory, attempting to write a custom record locking system is educational. After such an exercise, it's far easier to understand and appreciate how 4D works internally. Below are a few suggestions for anyone undertaking this task:

- Create a small, special-purpose database before starting. It's much quicker and easier to work with a simple system than a large existing code base.
- **PUSH RECORD** locks records for all processes, including the current process. This built-in feature can be used in the attempt to implement record locking.
- Records touched in a transaction remain locked until the transaction is cancelled or validated, so it is possible to lock ranges of records as a group.

Next, let's return to the example address database discussed earlier to see why custom record locking systems aren't needed.

Record Modification Example

As an example, we'll use the data from the imaginary example address database pictured earlier:

ACME Black Dot Company
17 Old Cactus Highway
Albuquerque, NM 87101

Records in this database can be edited at a remote warehouse through a Web interface, at the home office through a 4D Client interface, and from the road through a PocketPC-based SOAP system. Imagine that both a warehouse worker (Web-based) and a sales person in the field (SOAP-based) pull this record off the server for editing. The values at each location are shown below:

Warehouse	**Sales Rep**	**Server**
ACME Black Dot Company	ACME Black Dot Company	ACME Black Dot Company
17 Old Cactus Highway	17 Old Cactus Highway	17 Old Cactus Highway
Albuquerque, NM 87101	Albuquerque, NM 87101	Albuquerque, NM 87101

The values are identical. Now imagine that Joan, a remote sales person, is at ACME Black Dot making a sales call. She learns that the address should be changed to *28 New County Road*. Now, her copy of the record is different from the others:

Warehouse	**Sales Rep**	**Server**
ACME Black Dot Company	ACME Black Dot Company	ACME Black Dot Company
17 Old Cactus Highway	*28 New County Road*	17 Old Cactus Highway
Albuquerque, NM 87101	Albuquerque, NM 87101	Albuquerque, NM 87101

Joan saves the record, updating the original on the server and retrieving a fresh copy locally. Now her copy and the server copy agree, and the warehouse copy is out-of-date:

Warehouse Copy	**Sales Rep Copy**	**Server Copy**
ACME Black Dot Company	ACME Black Dot Company	ACME Black Dot Company
17 Old Cactus Highway	*28 New County Road*	*28 New County Road*
Albuquerque, NM 87101	Albuquerque, NM 87101	Albuquerque, NM 87101

Meanwhile, out at the warehouse, Ernest finishes a shipment for ACME. He reviews the shipping papers, sees that the address matches what's on his screen, and saves it back to the server. This resets the address Joan just changed back to the earlier value. Now Ernest's copy and the server agree, but Joan's changes have been lost:

Warehouse Copy	**Sales Rep Copy**	**Server Copy**
ACME Black Dot Company	ACME Black Dot Company	ACME Black Dot Company
17 Old Cactus Highway	*28 New County Road*	17 Old Cactus Highway
Albuquerque, NM 87101	Albuquerque, NM 87101	Albuquerque, NM 87101

This is a simple example of how multiple concurrent edits can go wrong, but it's more than adequate as a basis for discussion. More complex cases are easy to imagine but, ultimately, come down to the same basic behavior: there may be multiple versions of a single record open simultaneously in a multi-process/multi-user database. The core problem, then, is easy to see: when there are multiple copies of a record out for editing, changes can get written on top of each other in the wrong sequence. With this example in mind, let's return to the basic questions that must be addressed when reconciling edits without automatic record locking.

Managing Multiple Edits

Below are the three questions identified at the beginning of this note, along with a short description of how to program an answer:

- **Is the Record Locked?**
  If the record is currently in use by another process/client in a traditional 4D manner, the **Locked** function returns **True**.
- **Has the Record Been Deleted?**
  If the record can't be found, it has been deleted. Here a **QUERY** based on unique record values from the original returns zero records. This should emphasizes the importance of having a unique and unambiguous key for each table.
- **Has the Record Been Updated?**
  If the record has already been updated by another user, the current values are based on stale data, as in the situation with the ACME Black Dot address changes example described earlier. Detecting this situation requires a bit of planning and custom code. We'll look at how to approach this next.

Detecting Stale Copies of Records

In order to detect stale records, we need to know if the basis for the current copy of the record matches the original values in the record. Returning to the problem case in the example scenario above, the warehouse sends in an out-of-date copy of a record with the following values:

Warehouse Copy	**Server Copy**
ACME Black Dot Company	ACME Black Dot Company
17 Old Cactus Highway	*28 New County Road*
Albuquerque, NM 87101	Albuquerque, NM 87101

One way to detect that the warehouse copy is stale is to send the entire contents of the original record along with the update. Then, the original values can be compared with the master copy in the database before any update is applied. This strategy effectively requires sending two copies of each record with each update request. Apart from being awkward to implement at best, this approach is grossly inefficient. Conceptually, however, the idea is correct: the combined contents of the original fields are a unique signature for a specific copy of a record. Imagine the original values from the warehouse copy and the server copy combined and compared:

> ACME Black Dot Company **17 Old Cactus Highway** Albuquerque NM 87101
> ACME Black Dot Company **28 New County Road** Albuquerque NM 87101

It's obvious from looking at the concatenated values that the two copies differ. However, this scheme is poor because it requires sending and processing too much data. A better strategy is to create a signature that is small and easily calculated, such as a hash or a sequential version number.

Creating Signatures with a Hash

Hashing is a standard way to create compact signatures from blocks of data. Instead of transferring complete values, a hash is created from the original values and transferred instead. If the hashes don't match, the source values used to generate the hash also don't match. While this is a reasonable solution to the problem of distinguishing copies of records, it is unnecessarily expensive at runtime. Generating the hash requires scanning through all of the source data and applying various mathematical operations to generate the final hash. Instead, we can use a numerical counter as a version number to distinguish copies of records simply, reliably, and inexpensively.

Note *For more information on hashing, see technical notes 05-43* **The HashTools Component***, and 05-44* **Optimizing Searches with Hashes***.*

Adding Version Numbers to Records

If record copies needed to be distinguished, a simple and reliable solution is to add a version number field to the table. With this extra field, each time a record is saved, its version/update number can be incremented by one and saved with the record. Let's

look at the sample scenario again. This time, the record includes a version/update number, as listed below:

Warehouse Copy	**Sales Rep Copy**	**Server Copy**
ACME Black Dot Company	ACME Black Dot Company	ACME Black Dot Company
17 Old Cactus Highway	17 Old Cactus Highway	17 Old Cactus Highway
Albuquerque, NM 87101	Albuquerque, NM 87101	Albuquerque, NM 87101
Update #1	Update #1	Update #1

The values are identical. Now imagine that Joan, a remote sales person, is at ACME Black Dot making a sales call. She learns that the address should be changed to *28 New County Road*. Now, her copy of the record is different:

Warehouse Copy	**Sales Rep Copy**	**Server Copy**
ACME Black Dot Company	ACME Black Dot Company	ACME Black Dot Company
17 Old Cactus Highway	*28 New County Road*	17 Old Cactus Highway
Albuquerque, NM 87101	Albuquerque, NM 87101	Albuquerque, NM 87101
Update #1	Update #1	Update #1

Joan saves the record, updating the original on the server and retrieving a fresh copy locally with a modified update count. Now, her copy and the server copy agree and the warehouse copy is out-of-date:

Warehouse Copy	**Sales Rep Copy**	**Server Copy**
ACME Black Dot Company	ACME Black Dot Company	ACME Black Dot Company
17 Old Cactus Highway	*28 New County Road*	*28 New County Road*
Albuquerque, NM 87101	Albuquerque, NM 87101	Albuquerque, NM 87101
Update #1	Update #2	Update #2

Meanwhile, out at the warehouse, Ernest finishes a shipment for ACME. He reviews the shipping papers, sees that the address matches what's on his screen, and saves it back to the server. The server checks the incoming record detects that the record is based on an out-of-date update value. (Ernest's old copy has an update value of 1 while the server holds Joan's changes and an update value of 2.) The server-side code can easily return an error to inform Ernest that he is working with an out-of-date copy of the record.

If you think this system through, you'll find that despite the simplicity of the mechanism, it can handle record conflict scenarios of any complexity. So long as the update/version number is maintained accurately and consistently on the server side, it doesn't matter how end users received their copies of the record, how long they have held the copies, how they have modified them, or how they have submitted their changes. Changes derived from record export/import, SOAP, Web, or any other access scheme can all be managed successfully with the help of a simple record update tracking counter.

Thanks to Mark Burgess of LEAP Legal Software in Sydney for suggesting this solution.

Maintaining Update Values

The simplest way to maintain update values is in a trigger, as in the sample listed below:

```
Case of
   : (Database event=On Saving New Record Event )
     [Address]Update_Number:=1

   : (Database event =On Saving Existing Record Event )
     [Address]Update_Number:=[Address]Update_Number+1

End case
```

Detecting Stale Records: Where to Put the Code

At this point, it's fair to ask where to add the code to detect out-of-date records. In practice, you can put the code anywhere you manage record updates, be it from SOAP connections, Web requests, record imports, or custom code inside 4th Dimension. Alternatively or additionally, you can include the code in a trigger, like below:

```
C_LONGINT($0;$ErrorCode)
$ErrorCode:=0

Case of
   : (Database event=On Saving New Record Event )
     [Address]Update_Number:=1

   : (Database event=On Saving Existing Record Event )

     If (Old([Address]Update_Number)#[Address]Update_Number))
           ` The updated record either has no update version value (0)
           ` or an out-of-date version number. Reject the save.
       $ErrorCode:=-16000    ` Custom trigger errors should use values -32,000 through -15,000 .
     Else
       [Address]Update_Number:=[Address]Update_Number+1
     End if

End case

$0:=$ErrorCode
```

Even if you check for out-of-date records in custom code before the record is saved, it's a good idea to check in the save existing record trigger a fail-safe.

Variations on a Theme

The example discussed here uses an integer field to track record version numbers. Alternatively, you can use a date-time stamp, or any other scheme that supports distinguishing copies of records chronologically/sequentially. Unless there is a compelling reason not to use a simple counter, start with an integer.

Summary

4th Dimensions automatic record locking features make developing multi-process/multi-user systems remarkably easy. In those situations where the automatic features can't be used, the code required to manage multiple concurrent edits properly turns out to be simple. Code needs to check that the record still exists, load the record for writing, and verify that the update values are not based on an out-of-date copy of the record. Detecting stale records requires nothing more complicated than storing, maintaining, and testing a custom integer field.